Event-aware precise dynamic slicing for automatic debugging of Android applications

Hsu Myat Win a,b, Shin Hwei Tan b,c,∗, Yulei Sui a,∗∗

a Faculty of Engineering and Information Technology, University of Technology Sydney, NSW 2007, Australia
b Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, Guangdong, 518055, PR China
c Research Institute of Trustworthy Autonomous System, Southern University of Science and Technology, Shenzhen, Guangdong, 518055, PR China

A R T I C L E   I N F O

Article history:
Received 20 April 2022
Received in revised form 29 December 2022
Accepted 1 January 2023
Available online 7 January 2023

Keywords:
Android
Delta-debugging
Slicing

A B S T R A C T

Dynamic slicing aims to find the program statements that affect the values computed at some point of interest (i.e., a particular statement or variable) under a given program input. It is an enabling technique for many software engineering tasks (e.g., program understanding and debugging). Due to Android’s event-driven nature, dynamic slicing for Android is more challenging than that for traditional Java programs. Its asynchronous events drive the execution of an app through inter-component communications. These non-deterministic user events often yield a large search space when applying existing dynamic slicing techniques, which introduce redundant statements into the resulting slice.

We present ESDroid, an Event-aware dynamic Slicing technique for Android applications. The novelty of our approach lies in the combination of segment-based delta debugging and backward dynamic slicing to narrow the search space to produce precise slices for Android. Our experiment across 38 apps shows that ESDroid can help with slicing buggy code from exception program points. We compare the effectiveness of ESDroid with the state-of-the-art dynamic slicing tools (AndroidSlicer and Mandoline). ESDroid outperforms both tools by reporting up to 72% fewer spurious statements than AndroidSlicer, and 50% fewer than Mandoline in the resulting slice (the number of instructions to be examined).

© 2023 Elsevier Inc. All rights reserved.

1. Introduction

Program slicing (Weiser, 1984) collects the program statements that affect the values computed at some point of interest (i.e., a particular statement or variable, often referred to as a slicing criterion). While static slicing evaluates all possible program paths leading to the slicing criterion, dynamic slicing concentrates on one concrete execution for the given input (Agrawal and Horgan, 1990). Due to Android’s event-driven nature, slicing for Android is more challenging than that for traditional Java programs. Its asynchronous events drive the execution of an app through Inter-Component Communication (ICC). In addition, the Android framework supports the event queue mechanism to schedule and execute a user event. Due to arbitrary user interactions, adding an event to and dispatching another from the queue is non-deterministic. Such an event-driven system makes debugging and fault localization more complicated than traditional Java programs.

Static slicing techniques perform on a program dependence graph (PDG); the nodes of the PDG represent statements or a basic block, and the edges correspond to data or control dependences between nodes (Horwitz et al., 1988). Specifically, a directed data dependence edge $s_i \rightarrow s_j$ means any computation performed in $s_i$ depends on the computed value at node $s_j$. A control dependence edge $s_i \rightarrow s_j$ indicates that the decision to execute $s_i$ is made by $s_j$, that is, $s_j$ contains a predicate whose outcome controls the execution of $s_i$. The dynamic PDG, which is a subgraph of the static PDG (Ferrante et al., 1987), consists of only those nodes and edges that are exercised during a particular run. Precisely, a dynamic slicing tool first collects an execution trace of a program by instrumenting the program. Then, the tool checks the control and data dependences of the trace statements, determining statements that affect the slicing criterion and omitting the rest. The dynamic slices are more compact than static ones, making them suitable for debugging activities (Agrawal and Horgan, 1990; Agrawal et al., 1991; Korel and Laski, 1988), program understanding (Wang and Roychoudhury, 2008; Weiser, 1984), change impact analysis (Alves et al., 2011), regression test suite reduction (Gupta et al., 1992), and fault localization (Agrawal et al., 1995). However, dynamic slicing may include redundant...
statements if we do not consider input events, especially in Android apps with an event-driven nature. Specifically, redundant events with executed statements that do not affect the point of interest can lead to bigger slice with redundant statements.

**Existing Efforts and Limitations.** Basically, a backward slice identifies those statements that affect the point of interest (i.e., a particular statement or variable, often referred to as a slicing criterion), and a forward slice identifies those statements that are affected by the point of interest. Specifically, the backward dynamic slice at instruction \( s \) concerning slicing criterion (t.s. \( value \)) (where \( t \) is a timestamp) consists of executed instructions with a direct or indirect effect on \( value \). More precisely, the transitive closure over dynamic data and control-dependences in the PDG starts from the slicing criterion. The primary goal of dynamic slicing is to produce a precise PDG that excludes as many spurious nodes and edges as possible while soundly preserving the true buggy statements relevant to the bug-triggering point under a specific program input.

However, these traditional dynamic slicing approaches are inadequate for Android apps, yielding unsound outcomes (unaware of Android's ICs) or imprecise results (many redundant Android events taken as inputs). Specifically, the input event sequence impacts the slicing size for Android apps. In this paper, we focus on addressing this challenge, contributing an effective solution for slicing Android mobile apps by isolating the failure-inducing event sequence. Android slicing was already attempted in the tools called AndroidSlicer (Alavi et al., 2019) and Mandoline (Ahmed et al., 2021). AndroidSlicer presents asynchronous callback constructions for control- and data-dependences by defining callbacks as nodes containing other nodes (i.e., instructions) or a supernode. Mandoline enables tracking data propagation via object fields with low-overhead instrumentation and claims slicing accuracy for Android applications. Since Mandoline focuses on data-dependences by proposing an inter-callback dependency graph, there is no clear explanation for ICC, lifecycle stages, or control-dependences among callbacks. Moreover, both AndroidSlicer and Mandoline do not consider the input (i.e., a sequence of user events) for debugging and still suffer from many redundant or bug-irrelevant nodes on its slice when analyzing real-world apps. The inputs of an Android app are inherently complex (in the form of a wide variety of user events), and the slicing results are sensitive to Android events and their execution order. Hence, the inputs are crucial for precise slicing in Android. This paper aims to investigate, for the first time, an event-aware slicing approach by simplifying Android's input events to produce more precise slicing results.

Consider, for example, the SiliCompressor app in Fig. 1. SiliCompressor, is a Video and Image compression library for Android with 1200 stars in GitHub. It provides a demo app for illustrating its functionality. The code of the app was simplified for illustration purposes. We also discuss the example and the slicing algorithm at the source-code level for simplicity. At the same time, our solution can process apps at the bytecode level, even when no source code is available. Fig. la is the simplified app code of SiliCompressor, and Fig. lb is the slice produced by AndroidSlicer. Fig. 1c shows the activity state changes when the user clicks the event sequences shown in Fig. 1d. Fig. 1d is the randomly generated event sequence that makes the app fail with ArithmeticException: divide by zero. Fig. 1e is the stack trace. In our example app, the method widthDecrementClick of SiliCompressor class (Lines 4–8) is called when the user clicks “-” for width. This method decreases the value in width. Similarly, the method heightDecrementClick (Lines 9–13) is called when the user clicks “-” for height. This method decreases the value in height. If the user clicks "COMPRESS", the method compressImageClick (Lines 14–16) is called. This method calculates \( \text{maxRatio} \) by dividing \( \text{width} \) by \( \text{height} \).

The app fails when the user decreases the value of \( \text{height} \) to zero and calculates for \( \text{maxRatio} \), making “divide by zero”, which leads to the ArithmeticException (Line 15). Regardless of the integer value in the object of \( \text{width} \), if the value in the object of \( \text{height} \) is zero, the ArithmeticException: divide by zero will be thrown. Consequently, in the randomly generated event sequence, only two click events (i.e., E2--E3) are failure-inducing events. Only the statements of the callbacks (i.e., heightDecrementClick, and compressImageClick), enabled by the failure-inducing events, affecting the point of interest should be in the resulting slice. Specifically, a slice from the faulty line can help narrow down the program execution only to code relevant to the failure, e.g., omitting the code dealing with \( \text{width} \) (Lines 4–8). However, the state-of-the-art tools (i.e., AndroidSlicer Alavi et al., 2019 and Mandoline Ahmed et al., 2021) do not consider the input events and include spurious slices, resulting in a larger slice and search space. Thus, it leads to time-consuming for the developer. In our approach, to address this problem, we isolate the failure-inducing events by using delta-debugging before backward dynamic slicing.

**Insights and Challenges.** A typical technique to simplify a test input is delta-debugging, which systematically breaks down the original test input into smaller sequences until a minimal failure-inducing sequence is found (Zeller and Hildebrandt, 2002). The delta-debugging has been used in dynamic program slicing to narrow down the search space for faulty code in non-event-based programs (Gupta et al., 2005). The delta-debugging also has been used to simplify the trace for Android events (Clapp et al., 2016; Jiang et al., 2017). These techniques work purely on test inputs, treat an app as a black box and do not perform code analysis on Android bytecode or source code. Thus, their end goal is not dynamic slicing whose objective is to extract precisely the control- and data-dependence at bytecode level. How to incorporate and simplify the input events to obtain sound and precise dynamic slices for Android apps using a slicing criterion remains an open research question.

**Our Solution.** This paper presents ESDroid, an Event-Aware precise dynamic slicing approach for Android by introducing segment-based delta-debugging into backward dynamic slicing. ESroid first simplifies program inputs (i.e., the third phase in Fig. 2) when exercising Android apps before backward dynamic slicing (i.e., the fourth phase in Fig. 2). Thus, ESDroid significantly reduces spurious nodes and edges on the dynamic PDG. Specifically, ESDroid reduces the event sequence (i.e., program inputs) by using segment-based delta-debugging and then applies the backward dynamic slicing. For dynamic slicing, ESDroid builds control and data dependence at both the instruction and event levels (i.e., the fourth phase in Fig. 2). ESDroid aims to find a sub-set of slices produced by the state-of-the-art dynamic slicing technique AndroidSlicer. Our approach yields a more compact and precise slice than AndroidSlicer through input events reduction to isolate bug-relevant events further while soundly capturing the same bug reported by the original event sequence. Fig. 2 gives an overview of our approach consisting of four major phases. In the first phase, ESDroid conducts instrumentation on the target app to log the execution history so that ESDroid can track UI events plus the underlying methods and instructions in each activity. To record the number of events triggered and construct the dependences among events, ESDroid appends event levels (i.e., the fourth phase in Fig. 2). To log the timestamp and the information of executed instructions (Alavi et al., 2019). Note that we use the timestamp only for the node (instruction) creation, which is
important for detecting dynamic data dependences (Wang and Roychoudhury, 2008) and distinguishing between objects created at the same allocation site. Section 3.1 describes this in detail. In the second phase, ESDroid applies Monkey-style stress testing to generate random event sequences to exercise an app to trigger a crash/exception. To avoid the modification of Monkey files (Jiang et al., 2017) in the device, we implement a Python program that supports different device versions using MonkeyRunner\(^2\) to generate random events.

The third and fourth phases together form our main contribution (as highlighted in Fig. 2). The third phase accepts a failure-inducing sequence of events (FSOE) and removes the redundant and/or irrelevant events to produce a minimum failure-inducing sequence of events (\(\Delta\mathrm{FSOE}\)). To get the shortest event sequence, ESDroid adopts two strategies; (1) Divide and Conquer and (2) Complement. Section 3.3 describes this in detail. The final phase conducts dynamic slicing using \(\Delta\mathrm{FSOE}\) as the input and produces a precise dynamic slice based on the slicing criteria against the static PDG. We have evaluated ESDroid using 38 real-world apps. Our results show that ESDroid outperforms AndroidSlicer in terms of precision by reporting up to 72% (27% on average) less execution of false instructions (i.e., Jimple instructions) on the slices (i.e., dynamic PDG).

In summary, this paper makes the following contributions:

- We present ESDroid, a new event-aware dynamic slicing technique for simplifying inputs for Android apps.

---

We present how to apply delta-debugging in dynamic slicing to yield a more precise and compact PDG while capturing the same bugs as the state-of-the-art tools AndroidSlicer, and Mandoline.

We have implemented ESDroid and evaluated it using 38 real-world apps against AndroidSlicer, and 10 apps against Mandoline. The results show that ESDroid outperforms AndroidSlicer and Mandoline by reducing redundant nodes (i.e., up to 72% fewer than AndroidSlicer, and 50% fewer than Mandoline) on the dynamic PDG while maintaining all relevant nodes on the PDG. The evaluation data and the source code for ESDroid are publicly available (GitHub3, Zenodo4).

The rest of the paper is organized as follows. Section 2 presents a motivating example to illustrate our key ideas. Section 3 states our approach. Section 4 describes our implementation. Section 5 evaluates ESDroid by reporting its effectiveness and comparing it with AndroidSlicer, and Mandoline. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2. A motivating example

This section uses an example bug found in a SiliCompressor from GitHub shown in Fig. 3, as our motivating example. We aim to highlight the important insights and motivate our design decisions. We explain the typical challenge (i.e., if more events are triggered, the larger searching space occurs.) faced by the traditional debugging techniques. Fig. 3(a) gives the code fragment of the demo app. Fig. 3(b) shows a randomly generated sequence of user click events (i.e., FSoE) which triggers an ArithmeticException at Line 15 and the slice produced by AndroidSlicer. Specifically, widthDecrementClick is invoked upon clicking the "-” sign for width on app screen. The callback heightDecrementClick is invoked when clicking the "-” sign for height. compressImageClick is invoked upon clicking “COMPRESS”. Fig. 3(c) shows the simplified failure-inducing event sequence (i.e., ∆FSoE) and the slice produced by ESDroid. While AndroidSlicer has three click events and 9 nodes (i.e., statements), ESDroid has two click events and 6 nodes. The original click event sequence and the simplified one both trigger the same ArithmeticException. This is because the app will always crash if height at Line 15 represents a zero value.

Although there are three click events in total for the original event sequence, only the last two click events (i.e., heightDecrementClick and compressImageClick) are the failure-inducing events. Thus, the dynamic slice should only include program statements of these two events affecting the point of interest. Specifically, the resulting dynamic slice should contain only Lines 2, 3, 10, 11, 12, and 15 shown in Fig. 3(c). With a thinner slice, the developer will have fewer buggy lines to inspect, which helps reduce the time and effort in debugging process. Moreover, the shorter event sequence saves developers time in validating the app’s behavior.

Table 1 demonstrates that ESDroid can successfully identify this failure-inducing event and remove other unrelated occurrences. Compared with the state-of-the-art dynamic slicing approach AndroidSlicer, ESDroid can produce a much smaller but more precise backward slice (with only six rather than nine statements) starting from the exception point. Specifically, our reduction process performs by producing FSoE, simplifying FSoE, and conducting backward dynamic slicing after instrumenting the SiliCompressor app.

2.1. Producing FSoE

To produce the event sequence that makes the app crash, ESDroid exercises the instrumented app by applying Monkey-style stress testing on it with randomly generated events until a crash is triggered. We select a scenario where after exercising three click events, the application failed with an ArithmeticException. This error occurs because the program attempted to divide by zero value. ESDroid records the executed instructions together with this failure triggering point into the trace file.

2.2. Simplifying FSoE

Our goal is to reduce the size of the event sequence, which triggers an exception, and to produce a more precise and compact program slice. ESDroid gradually removes some redundant events from the event sequence using segment-based delta-debugging. This is done iteratively by exercising a sub-sequence of events on the instrumented app to check which runs can produce the same exception. Table 1 illustrates the iteration process for the motivating example. The first column describes the number of iterations, and the second column records the corresponding click event sequence triggered. The third column presents the value stored in three integer objects (i.e., width, height, and maxRatio) at the timestamp once the last click event is triggered. “Test result” holds the outcome of each test.

Iteration 0 is the original FSoE. In Iteration 1, we divide the FSoE into two sub-sequences. The first sub-sequence contains E1 while the second sub-sequence includes E2 and E3. The testing is first conducted for the last sub-sequence (E2→E3) because the sub-sequence which includes the last event of FSoE has a higher chance of triggering the bug (Jiang et al., 2017). Since the second sub-sequence makes the app crash (i.e., the app crashes with the same stack trace of the original FSoE), we start the next iteration with the second sub-sequence. We take the result as “fail” if the event sequence triggers the same bug with the same stack trace. We describe details in Section 3.3. Note that, in our approach, once we find the event sequence, which causes the app to fail, we start the next iteration with the last failed event sequence.

---

3 https://github.com/hsumyatwin/ESDroid-artifact
In Iteration 2, we divide the failed event sequence of Iteration 1 into two sub-sequences, and each sub-sequence includes one event (i.e., the first sub-sequence contains E2, and the second sub-sequence contains E3). Both sub-sequences make the test pass (i.e., no bug is triggered), and the reduction process also reaches 1-minimal. The simplified event sequence (i.e., $\Delta FSoE$) is
generated with two events at Iteration 1 (i.e., E2 → E3). ESDroid can safely exclude the redundant click event (i.e., widthDecrementClick). Formally, we define the property as \( n \)-minimality: removing up to \( n \) events causes the failure to disappear. Suppose \( s \) is \(|s|\)-minimal, then \( s \) is the minimal number of removed event/s. A failure-inducing event sequence \( s \) composed of \(|s|\) events would be 1-minimal if removing any single event would cause the failure to disappear.

2.3. Backward dynamic slicing

To obtain the executed instructions that affect the value of \( \text{maxRatio} \), we perform backward dynamic slicing on both the original test case (i.e., FSoE) and the simplified test case (i.e., \( \Delta \text{FSoE} \)). The criteria we used are (1) the timestamp when the exception is thrown, (2) the object holding error (\( \text{maxRatio} \) at Line 15), and (3) the instruction at Line 15 accessing this object. As shown in Fig. 3(b), for the original event sequence with the three click events produced by AndroidSlicer, the slice has 9 lines (Lines 2, 3, 5, 6, 7, 10, 11, 12, and 15) from the program’s entry to the program failure point (the point of interest). Fig. 3(c) shows that the slice has 6 lines (Lines 2, 3, 10, 11, 12, and 15) with a simplified sequence of events (i.e., \( \Delta \text{FSoE} \)). ESDroid forms the smaller slice with six statements by capturing the bug triggering point at Line 15 and the root cause of the error. We observed that nodes on the original PDG (Lines 5, 6, and 7) are not required to be examined while determining the source of error; thus, they are irrelevant to the slicing criteria and irrelevant to include them in the slice. AndroidSlicer includes these counterfeit nodes because it slices all the executed instructions affecting the failure point based on the original sequence of events, provided there are control dependences and data dependences between these nodes based on the static PDG. Therefore, by considering input events, ESDroid successfully reduces redundant statements and yields a more compact and precise program slice than AndroidSlicer.

3. Approach

Fig. 2 shows the overall workflow of ESDroid. Given an app and a slicing criterion, ESDroid generates a reduced dynamic slice to identify the faulty code block. ESDroid consists of four phases. First, we instrument an app with each of its bytecode instructions shadowed with another instruction for runtime bookkeeping. In the second phase, ESDroid runs the instrumented app and extracts the event sequence that triggers a crash (we call this sequence \( \text{Failure-inducing Sequence of Events (FSoE)} \)). After producing the FSoE, we perform delta debugging to obtain a minimized FSoE. Finally, ESDroid conducts the dynamical slicing to capture control- and data dependence at both instruction and event levels by incorporating the reduced FSoE to produce a more precise dynamic slicing than the state-of-the-art.

3.1. Instrumentation

Before running an Android app, ESDroid performs lightweight instrumentation on the app to collect information on which events are triggered and which statements are executed during runtime. Specifically, ESDroid instruments the app to produce the trace, which includes the executed instructions, the information of intent creation, and callbacks. We use Soot (Vallée-Rai et al., 2010) to perform instrumentation, and a new Jimple instruction is injected for every application instruction to record the execution trace. The inserted instruction is responsible for bookkeeping the executed application instruction information, including its line number, corresponding class name, and method name. To construct the call graph of an Android app, we use FlowDroid (Arzt et al., 2014) by considering the Android’s event-based life cycle. For each node (i.e., program method) on the call graph, we use EventID to differentiate Android events. Note that though all the dynamically executed instructions, including those in the framework, are recorded in our execution log, these framework instructions do not manifest in the application’s dex code when performing our control- and data dependence analysis. Our dynamic slicing is performed at the application level.

ESDroid instruments and numbers an Android event with its corresponding eventID. ESDroid records the execution information based on the following format.

- **Timestamp** — time when the particular instruction runs.
- **Data** — eventID, program line number, class name, event name, and the instruction including objects if available.

Note that, we use eventID to record the number of events triggered for Section 5.2 and construct the dependences among events. The program line number is to map back the Jimple instruction to the program statement to check the quality of the slice in Section 5.5.

**Example 1.** The following shows a part of the execution trace after running the instrumented SiliCompressor app (i.e., the motivating example). In this recorded trace, for Line 15 in Fig. 3, we use a separator _ to denote different types of data. Specifically, 09-21 00:23:51.027 represents the timestamp, ID4 is an auto-incremental unique number for a callback, and 15 is the program line number. We also record the class name com.i.sc.SiliCompressor, the callback name compressImageClick, and the executed instruction (i.e., Jimple instruction) for Line 15 including the objects $r4 (i.e., maxRatio) and $r2 (i.e., width), $r3 (i.e., height).

```
09-21 00:23:51.027 System.out:ESDroid_ID4_15_com.i.sc.SiliCompressor_compressImageClick_Sr4=Sr2/Sr3;
```

3.2. Producing FSoE

ESDroid generates random events to exercise the instrumented apps until the program fails. For example, the event sequence E1 → E2 → E3 shown in Table 1 triggers an exception. While SimplyDroid (Jiang et al., 2017) relies on a modified Monkey for each Android version, we implement a Python program to be compatible with different device versions using MonkeyRunner. Specifically, we randomly set the \((x, y)\) coordinate, ranging from zero to the resolution of the emulator (the maximum height and the maximum width), to avoid generating out-of-bound values for the coordinate \((x, y)\). Note that this way of generating event sequences simulates clicks, rotations, and drags, and we currently do not support other complex events like changing the configuration of the phone. The maximum number of random events for each run is 5000. We rerun the app ten times using a newly generated event sequence (with different seed values) if the previous run is unable to trigger a bug.

3.3. Simplifying FSoE

The goal is to eliminate redundant events irrelevant to a program failure and retain as few relevant events that trigger the same exception as possible. An event on an event trace \( t \) can be safely removed by our delta debugging to produce a simplified trace \( t' \) only if \( t \) and \( t' \) trigger exactly the same bug, i.e., the same exception error and the same stack trace. For example, in Fig. 3, although we removed the event which triggers widthDecrementClick, the remaining two click events still trigger the same bug because both the original and reduced event
sequences feed the invalid values (i.e., zero) in height. The reduction process (i.e., segment-based delta-debugging) is repeated until ESDroid produces a minimum failure-inducing sequence of events (i.e., ΔFSOE), which is used for the later dynamic slicing because, with a shorter sequence, it is easier to find the error in terms of debugging process.

To determine whether the current event sequence is failure-inducing, we use the outcomes of app testing as the selection criteria. Following are four possible outcomes of app testing.

- The app exited normally without any crash.
- The app crashed with a different error or exception type.
- The app crashed with the same exception type and stack trace.
- The app exited normally without any crash for the test.

Among the above four possible outcomes, we define the first three outcomes as "pass" and the last as "fail". We take the event sequence with a "fail" outcome as a failure-inducing event sequence, and bring it to the next iteration. To mitigate the problem of flaky tests, (1) we re-run the event sequence under the same system environment, and (2) instead of only comparing the test outcome, we compare the test result (i.e., exception/error type) and stack trace for each iteration with the stack trace of the original FSOE. We discarded the cases where the test re-run did not crash with the same result test and stack trace. Note that our current debugging process requires a crash/exception for delta debugging. In the future, we will enhance ESDroid to handle non-crashing bugs.

**Definition 1 (n-Minimal Sequence).** An event sequence \( S \) is n-minimal if \( |S| \leq n \Rightarrow \forall x \in S: \text{test}(s - \{x\}) \neq \top \) holds, where \( \top \) is the fail outcome. Consequently, \( S \) is 1-minimal if \( \forall x \in S: \text{test}(s - \{x\}) \neq \top \) holds.

**Definition 2 (Granularity).** Granularity means the number of sub-sequences that ESDroid divides the sequence of events into.

**Definition 3 (Complement Logic).** The relative complement or sequence difference of sequences \( A \) and \( B \), denoted \( A - B \), is the sub-sequences \( x \) in \( A \) that are not in \( B \). In notation, \( A - B = \{x \in A \mid x \notin B\} \).

ESDroid first divides an FSOE into sub-sequences or so-called segments (sub-sequences of events) based on granularity (i.e., 2 at the beginning of the reduction process). We choose 2 as the granularity for the first iteration because there is no fixed value or obvious formula that could give the best split factor (size or performance-wise), and it could provide the worst and best-case behavior of the delta debugging process (Kiss, 2020). Moreover, we intend to reduce the slice, and the fundamental strategy of delta debugging is already robust and effective enough to obtain a significant reduction rate. In each iteration, ESDroid follows either of the two strategies for partitioning FSOE (i.e., the input for testing) to conduct the testing. One is Divide and Conquer, and the other is Complement (Zeller and Hildebrandt, 2002) based on the results after each iteration. ESDroid applies the Complement strategy once all sub-sequences do not trigger the same bug and the same stack trace with the original event sequence. Otherwise, ESDroid uses the Divide and Conquer strategy to narrow down the failure-inducing events.

For every iteration, ESDroid triggers the last sub-sequence (i.e., the event sequence, which includes the last event) first because the last sub-sequence has a higher chance of triggering the bug (Jiang et al., 2017). In addition, to reduce the iteration process, once ESDroid finds the failed event sequence, it terminates the current iteration and starts the next iteration with granularity 2 for the Divide and Conquer and maximum value between (current granularity-1) and 2 for the Complement.

**Algorithm 1: Simplifying failure-inducing sequence of events (FSOE).**

**Input:** a list of events FSOE, the stack trace \( e \) of FSOE. **Output:** a list of statements \( T_f \).

```
1 \( T_f \leftarrow \{\}\);
2 \( n \leftarrow 2\);
3 isFailed \leftarrow \text{false};
4 \text{if} \ FSOE.size() == 1 \text{then}
5 \( (\text{isFailed, } T_f) \leftarrow \text{test(} FSOE, e)\);
6 \text{end}
7 \text{while} \ FSOE.size() >= 2 \text{do}
8 \( S \leftarrow \text{divide } FSOE \text{ into } n \text{ sub-sequences } S_1, S_2, S_3, \ldots \)
9 \( S_n\); // Divide the event sequence into \( n \) (i.e., granularity) sub-sequences equally. \( \text{If the number of events in the sequence could not make sub-sequences equally, we favor the last sub-sequence to have one more event.}\)
10 \text{for each sub-sequence } S_i \text{ in } S \text{ do}
11 \text{if test(} FSOE|S_i\), e) != null \text{ then}
12 \( (\text{isFailed, } T_f) \leftarrow \text{test(} FSOE|S_i\), e)\);
13 \text{end}
14 \text{if } isFailed \text{ then}
15 \( FSOE \leftarrow FSOE|S_i\);
16 \( n \leftarrow \max(n - 1, 2)\);
17 break;
18 \text{end}
19 \text{end}
20 \text{if } isFailed \text{ then}
21 \text{if } n == FSOE.size() \text{ then}
22 break;
23 \text{end}
24 \( n \leftarrow \min(2n, FSOE.size())\); // Increase granularity and start Complement strategy
25 \text{end}
26 \text{end}
27 \text{return } T_f;
```

**Procedure:** test(list of events \( S_n \), stack trace \( e \))

```
28 \text{if } S_i \text{ triggers the app crash then}
29 \( x \leftarrow \text{dumpStack()}; // print stack trace of crash}
30 \text{if } e == x \text{ then}
31 \( T_f \leftarrow \text{logcat(); // get all executed program statements}\)
32 \text{return } \text{true}(T_f);\)
33 \text{end}
34 \text{end}
35 \text{return } \text{null};;
```

**Example 2.** Table 2 shows the process of the Divide and Conquer. For the first iteration, ESDroid divides an FSOE into two sub-sequences (i.e., one with E3 --- E4 and the other with E1 --- E2). We first test for the last sub-sequence (i.e., E3 --- E4). Since E3 --- E4 triggers the bug, ESDroid uses it as input for the next iteration. At Iteration 2, ESDroid divides the latest sub-sequence, which makes the app fail, into 2 sub-sequences (i.e., one with E4 and the other with E3). ESDroid conducts the testing for the last sub-sequence first (E4) and the program fails. Since ESDroid iterates the reduction process until 1-minimal sub-sequence, it terminates the process and E4 is the event that is responsible for program failure (i.e., ΔFSOE). Note that the granularity for the Divide and Conquer is 2 for every iteration.

ESDroid adopts the Complement strategy for the next iteration if neither sub-sequence produces the bug in the current iteration.
because the smaller sub-sequences and testing the complement of the smaller sub-sequence gives a higher chance of resulting in program failure (Zeller and Hildebrandt, 2002).

**Example 3.** Table 3 shows the process of the Complement. There are 8 events in FSoE and the current granularity is 2 (i.e., 2 sub-sequences with 4 events in each sub-sequence). At Iteration 1, since both sub-sequences are unable to trigger the same bug with the same stack trace as that of the FSoE, ESDroid increases the granularity from 2 to 4 (i.e., a minimum value between 8 events of FSoE and 2 times of current granularity). Therefore, we have 4 sub-sequences with 2 events in each for Iteration 2. We generate the granularity with two formulas. We use min(2n, FSoE.size()) to increase the granularity if none of sub-sequences in the same iteration triggers the app to fail. If one or more sub-sequences trigger the app to fail, we use max(n - 1, 2). Note that we start the next iteration once one of the sub-sequences in the same iteration makes the app fail. n is the current granularity. FSoE.size() is the number of events in the current working event sequence (i.e., the latest event sequence which makes the app fail). For example, 8 events in Iteration 1. We describe this in detail in Algorithm 1. At Iteration 2, ESDroid tests for the last complement (i.e., E3 → E4 → E5 → E6 → E7 → E8). Since the current complement triggers the bug, ESDroid brings the current complement to the next iteration which operates with granularity 3 (i.e., the maximum value between (current granularity-1) and 2). At Iteration 3, ESDroid skips the last complement because it is the same as the second sub-sequence of Iteration 1 and is tested for the next complement (i.e., E3 → E4 → E7 → E8). ESDroid terminates the reduction process if the smallest sub-sequence cannot be further reduced (i.e., 1-minimal sub-sequence is tested at Iteration 5) and the failure-inducing complement (i.e., ΔFSoE) is the last event sequence (i.e., E3 → E4 → E7 → E8) which makes the program fail.

Algorithm 1 describes the process of simplifying FSoE to the following:

- **FSoE:** A sequence of events which makes the app fail. Initially, it holds the failure-inducing sequence of events (FSoE) produced by the second phase.
- **Tf:** A list of executed program statements when the current FSoE triggers an instrumented APK (i.e., output). Initially empty.
- **S:** A list of sub-sequences after dividing current FSoE into n (i.e., granularity) sub-sequences (Line 8). Each sub-sequence includes the same number of events. If the sequence’s number of events could not equal the sub-sequences, we favor the last sub-sequence to have one more event.

Given two inputs: (1) an event sequence which makes the app fail (denoted as FSoE) and (2) the stack trace of FSoE (denoted as e), ESDroid iterates the reduction process until the count of events in the sequence is greater than or equal to 2 (Line 7) or the granularity n reaches 1-minimal sub-sequences (Lines 20, 21 and 22). For the case where no simplification is needed (FSoE has only one event), our approach collects and returns the log (Lines 4–6). If the number of events in FSoE is greater than one, we divide the event sequence into n (i.e., granularity) sub-sequences equally at Line 8. If the number of events in the sequence could not make sub-sequences equally, we favor the last sub-sequence to have one more event because the last sub-sequence which includes the last event of FSoE has a higher chance of triggering the bug (jiang et al., 2017). For example, if the event sequence has three events and the granularity is 2, we split one event for the first sub-sequence and two events for the second sub-sequence.

Note that, for the first iteration, the granularity is 2 (Line 2). We select 2 as the granularity for the first iteration because there is no fixed value or obvious formula that could give the best split factor in terms of size or performance-wise, and it could provide the worst and best-case behavior of the delta debugging process (Kiss, 2020). Moreover, we intend to reduce the slice, and the basic strategy of delta debugging is already reasonable and practical enough to obtain a considerable reduction rate. ESDroid then extracts the complement of the current sub-sequence (Note that, since there are only two sub-sequences for the Divide
and Conquer approach, the complement of one sub-sequence is the other sub-sequence) and conducts the testing (Lines 10–12) (Lines 28–35). If the current complement makes the program fail with the same stack trace \( e \), we keep the trace log including the executed statements as the latest (i.e., the output \( T \)) (Lines 31, 11) and update \( FSoE \) with the current complement (Line 14).

The algorithm stops using the Divide and Conquer strategy and starts using the Complement strategy once none of the sub-sequences in the same iteration triggers the bug with the same stack trace (Line 23). We describe details in Example 4.

### Example 4

Adjusting granularity \( n \) is done at Line 15 for the test failed. For example, (1) For the Divide and Conquer, the granularity is 2 (i.e., \( 2 = \max(2–1,2) \)). (2) For the Complement, if the current granularity is 4 (i.e., 4 sub-sequences in \( FSoE \) for current iteration), granularity for next iteration is 3 (i.e., \( 3 = \max(4–1,2) \)) because \( FSoE \) is updated with the current complement (i.e., 3 sub-sequences). Suppose all complements are unable to make the program fail. In that case, increasing granularity \( n \) is done at Line 23 (i.e., a minimum between 2 times of current granularity and count of events in current \( FSoE \)). Note that if the null value returned for \( T \) at the end of the algorithm, ESDroid stops at the current phase (i.e., Phase 2) because there is no input for the next phase (i.e., Phase 3). However, according to our experiment, none of the traces for all experiment apps is empty.

### Data-dependence

There are two data-dependence levels, i.e., the data-dependence between the program statements and the data-dependence between events. As shown in Fig. 4, at Line 5, a statement \( S_{t} \) utilizes the same object \( o_1 \) which is defined at Line 3 in \( S_{t} \) and \( S_{t+1} \) is data-dependent on \( S_{t} \) at time \( t \).

### Example 5

To illustrate the data dependencies in our approach, let us revisit the example in Fig. 3(c). The slice of \( maxRatio \) at Line 15 includes nodes 2, 3, 10, 11, 12, and 15 because \( maxRatio \) is defined with the value of \( width \) and \( height \) at Line 15, and where \( width \) is defined with the \( int \) value 1 at Line 2. Similarly, \( height \) is defined with the \( int \) value \( height−1 \) at Line 10 in \( heightDecrementClick \). Therefore, node 15 is data dependent on node 2, and node 10. The same approach applies to nodes 3, 11, and 12.

For data dependency among the events, as shown in Fig. 4, event \( E2 \) (onClick2 at Line 4) is data-dependent on event \( E1 \) (onClick1 at Line 2) because instruction \( S_{t+2} \) in \( E2 \) is data-dependent on \( S_{t+1} \) in \( E1 \). But, only because object \( o_2 \) used in \( S_{t+2} \) (Line 5) depends on object \( o_1 \) defined in \( S_{t+1} \) (Line 3) at that time \( t \).

### Example 6

In our motivating example in Fig. 3, if \( widthDecrementClick \) and \( heightDecrementClick \) are triggered before triggering \( compressImageClick \), \( compressImageClick \) is data-dependent on both \( widthDecrementClick \) and \( heightDecrementClick \) via \( width \) and \( height \) respectively. The slice in Fig. 3(c) contains node 12 because \( compressImageClick \) is data-dependent on \( heightDecrementClick \) via \( height \).

### Control-dependence

As with data-dependence, there are two levels of control dependence at the levels of instruction and event. For the former, as in Fig. 5, if an instruction \( S_{t} \) at Line 2 is executed only upon the execution result of \( S_{t+1} \) at Line 1, \( S_{t+1} \) is control-dependent on \( S_{t} \). To clarify, the value of condition (i.e., the predicate) at Line 4 determines the execution of \( S_{t} \). In other words, if \( S_{t} \) can alter the program’s control and it determines whether \( S_{t+1} \) executes (Ferrante et al., 1987). Examples of statements that can alter the control are if and while.

Because of Android’s life cycle nature, unlike traditional Java, for control dependence among events, there are two ways to determine the execution of another callback by a callback.

1. **Direct-control dependence**: A component’s event directly determines the execution of another event via an initialized object. For example, as shown in Fig. 5 (Lines 3–8), \( onCreate \) of \( Act3 \) has triggered the activity (i.e., initialized object \( Act4 \) context transitions via \( startActivity \) at Line 6 and the execution of \( onCreate \) of \( Act4 \) (i.e., \( E4 \)) is directly controlled by \( onCreate \) of \( Act3 \) (i.e., \( E3 \)). Therefore, \( E4 \) is direct-control-dependent on \( E3 \) (i.e., \( E4 \rightarrow \ E3 \)).

2. **Lifecycle-control dependence**: An event of a component initiates the execution of another component’s event because of Android’s component lifecycle. For example, as shown in Fig. 5 (Lines 9–15), \( onPause \) of \( Act7 \) (i.e., \( E7 \)) determines the execution of \( onCreate \) of \( Act8 \) (i.e., \( E8 \)) by completing itself because \( E8 \) will not be invoked until \( E7 \) returns. Therefore, \( E8 \) is control-dependent on \( E7 \) because of the lifecycle (i.e., \( E8 \rightarrow \ E7 \)).
Based on the control- and data dependence, ESDroid builds PDG. ESDroid then maps the executed statements in the simplified trace $\Delta FS\text{SoE}$ to the static PDG by conducting a backward dynamic slicing. ESDroid finds all the associated control- and data-dependence statements on the PDG based on a slicing criterion $\langle t, s, o \rangle$, where $t$ is a specified timestamp, $s$ is an error node (an executed instruction) occurring at $t$, and $o$ is a sequence of objects holding an error at the node $s$. Same as AndroidSlicer, the extracted control- and data-dependence slices are at the application level (manifest in the application’s dex code generated by Soot Vallée-Rai et al., 2010) when reporting to users.

Algorithm 2 illustrates our backward dynamic slicing with the data structure:

- $T_f$: A list of executed statements when $\Delta FS\text{SoE}$ is triggered on an instrumented APK.
- $idx$: An integer that is the location of the error instruction in $T_f$ (i.e., the last index of $T_f$ for the app crash because the last index holds the failure point, which is the point of interest).
- $S_l$: A list of executed statements affecting the point of interest (i.e., the output). Initially empty.
- $PDG_{CD}$: A list of nodes, which is a dynamic control dependence graph. Initially empty.
- $PDG_{DD}$: A list of objects which is a dynamic data dependence graph. Initially empty.

Given three inputs; (1) instrumented apk (denoted as apk), (2) a trace file (denoted as $T_f$) including the list of statements executed while $\Delta FS\text{SoE}$ is triggered, and (3) the index of $T_f$ (denoted as $idx$) in which an executed statement with the object holding error occurs at the particular timestamp, ESDroid slices the executed statements (i.e., the output of slicing process), denoted as $S_l$, affecting the point of interest until the app entry point. Note that the pre-conditions of the algorithm are (1) $T_f$ cannot be the empty set, and (2) idx must be a valid index. We sorted the executed statements according to the executed order in the execution trace because we use the trace log as input (i.e., $T_f$) that includes the execution trace. Constructing PDG (denoted as $PDG_{DD}$ for data dependence and $PDG_{CD}$ for control dependence) is done dynamically at Lines 18, 22 and 25 with the help of static PDG. Specifically, ESDroid collects all the used objects in the working node (i.e., checking data dependence) at Lines 16–20. To list the nodes for control dependence, $\Delta S\text{CD}$ checks whether the execution of the current working node (i.e., the node located at the current index $idx$ of $T_f$) (denoted as $T_f[\text{idx}]$) is determined by the previous node (denoted as $T_f[\text{idx-1}]$) for instruction-level control dependence (Lines 21–23). Particularly, $\Delta S\text{CD}$ examines if the node located at $T_f[\text{idx-1}]$ contains a predicate whose outcome controls the execution of the node located at $T_f[\text{idx}]$. ESDroid further checks for event-level control-dependences and, it appends $PDG_{CD}$ with the last node of the method which initiates the method of the current working node (Lines 24–26) with the help of static PDG if the method of the current working node is the callback. Specifically, $\Delta S\text{in}$ in the algorithm helps to get the last node of the method that initiates the method of the current working node. If ESDroid finds the current working node in dynamic PDG (i.e., $PDG_{DD}$ and $PDG_{CD}$) (Lines 4–12), and ESDroid adds the current working node to the output after checking for duplicated instructions (Lines 13–15). For example, when the same instruction occurs in the source code but is executed multiple times, ESDroid also checks whether the current instruction is dependent on previous occurrences in the output slice.

### 4. Implementation

We describe the implementation details of the four phases in ESDroid as follows:

**Instrumentation and Producing $FS\text{SoE}$**: ESDroid uses Soot (Vallée-Rai et al., 2010) to conduct the instrumentation to produce our customized logging information. Regarding producing $FS\text{SoE}$, there are several techniques to generate event sequences...
Algorithm 2: Backward Dynamic Slicing. Input: an Apk apk, a list of statements $T_f$, the position $idx$ which is the point of interest in $T_f$. Output: a list of statements $S_l$.

1. $S_l \leftarrow \emptyset$;
2. $isSlice \leftarrow true$;
3. while $idx>0$ do
4.  for each Object o defined at $T_f[idx]$ do
5.    if $PDG_o.contains(o)$ then
6.      $isSlice \leftarrow true$;
7.      break;
8. end
9. end
10. if $PDG_o.contains(T_f[idx])$ then
11.   $isSlice \leftarrow true$;
12. end
13. if $isSlice$ and $S_l.contains(T_f[idx])$ then
14.   $S_l.add(T_f[idx])$;
15. end
16. if $isSlice$ then
17.  for each Object o used in $T_f[idx]$ do
18.    $PDG_o.add(o)$;
19. end
20. end
21. // check $T_f[idx-1]$ contains a predicate whose
22. // outcome controls the execution of $T_f[idx]$
23. if $isSlice$ and isCD($T_f[idx].T_f[idx-1]$) then
24.   $PDG_o.add(T_f[idx-1])$;
25. end
26. if $isSlice$ and $T_f[idx]$'s method $m$ is callback then
27.   // add the last statement of callback which
28.   // initiates $m$
29.   $PDG_o.add(getS(apk,m))$;
30. end
31. $idx \leftarrow idx-1$;
32. $isSlice \leftarrow false$;
33. end
34. return $S_l$;

Backward Dynamic Slicing. In this phase, we first built the static PDG. There are two levels of dependency on the static PDG as described in Section 3.4, i.e., the event level that acquires the control- and data-dependence between Android events, and the method level that captures the dependence between two instructions. For the instruction level, we used the static PDG generated by Soot. For the event-level, we leveraged AndroidSlicer’s event-level PDG to produce the final static PDG. Next, our dynamic PDG was produced by our dynamic slicing algorithm. This includes only the executed statements of the static PDG based on the slicing criteria when running the instrumented app under the test input $\Delta FSoE$.

5. Evaluation

Existing automated debugging techniques for Android Apps include (1) MZoltar (Machado et al., 2013) that uses spectrum-based fault localization, (2) AndroidSlicer that performs dynamic slicing, (3) Mandoline that evaluates dynamic slicing with alias analysis. We choose to evaluate our approach on AndroidSlicer and Mandoline because (1) they are publicly available (we did not evaluate against MZoltar as it is not publicly available), and (2) they are state-of-the-art slicing techniques for Android Apps. Our experiments aim to evaluate the effectiveness of ESDroid by (1) comparing the size of the slices it produces with those produced by AndroidSlicer and Mandoline, and (2) analyzing the quality of those slices for debugging.

5.1. Experiment setup and methodology

5.1.1. Evaluation datasets

We evaluated ESDroid on 41 defects from 38 open-source Android apps for 17 exception types. These apps cover a wide range of domains as per listed in Table 4. Ten of these apps, used in previous literature (Alavi et al., 2019; Jiang et al., 2017), are available at Google Play (i.e., NPR News, Olam, Addi, CowSay, PasswordMaker, Tickmate, TripSit, Transistor, AnyMemo and GnuCash). We evaluated on benchmark apps because we need to manually verify whether the resulting slice includes the bug location. Table 4 lists the information about the evaluated apps. The “Exception Type” column contains information about the specific type of exception that causes the crash, whereas the “Dataset” column represents the dataset or Google Play. Overall, the evaluated datasets contain a wide variety of apps of various sizes (27-17654 KB of Dex code) with 1 to 27 activities. These datasets have different types of exceptions that lead to crashes. We selected these defects based on the following criteria:

C1: Apps from different categories

C2: Crashes with different types of exceptions to check whether ESDroid can capture the bug for different exception types.

C3: Crashes that our random event sequence generation can reproduce in at least one of the ten runs.

In addition, we ensured that these defects were obtained from the prior evaluation of analysis techniques of Android apps. Specifically, we evaluated:

- seven apps (i.e., WeightChart, DalvikExplorer, Ring-
droid, SyncMyPix, Tippy, WhooHaaMyStuff and Yahtzee) from the previous evaluation of SimplyDroid (Jiang et al., 2017).
- two apps (i.e., APV PDF Viewer, NPR News) from the previ-
ous evaluation of AndroidSlicer (Alavi et al., 2019).
- four apps (i.e., Fdroid, AnyMemo, GnuCash and Transis-
tor) from Droibench (Tan et al., 2018).
To evaluate the applicability of our approach beyond these benchmark apps, we further evaluated on eleven closed-source apps from Google play (i.e., Calculator, Carnet - Notes app, Fish-Bun Demo, Geometric Weather, Linux Deploy, Man Man, Mitzuli, PasswordMaker, Scribbler, Tickmate and TripSit) from DroidDefects (Su et al., 2020).

To evaluate the applicability of our approach beyond these benchmark apps, we further evaluated on eleven closed-source apps from Google play (i.e., Calculator, Carnet - Notes app, Fish-Bun Demo, Geometric Weather, Linux Deploy, Man Man, OB-SSD - OBS Stream Deck, Official Cambridge Guide to IELTS, Scale Image View Demo, Tailscale and Vanilla Music). We selected these closed-source apps because (1) they are diverse in terms of size and functionalities, and (2) they contain crashes that can be triggered without requiring any additional login information.

Specifically, we excluded 10 defects from the previous evaluation of the fault localization application in AndroidSlicer and 11 apps from RelFix because (1) the dataset was not publicly available, and (2) we failed to find the corresponding apps in GitHub. Moreover, we excluded 10 apps from Droixbench and 9 apps from DroidDefects in our experiments because (1) these crashes require complex inputs and specific sequences of events that cannot be generated automatically by our event sequence generation (does not satisfy $C_3$), and (2) instrumentation failed because Soot fails to parse the apk (i.e., Dex file overflow error for Android API 22). We also excluded 4 apps from DroidDefects because the test re-run did not crash with the same test result and stack trace. Although ESDroid operates on the apk file and supports both open-source apps and closed-source apps, we manually analyzed 27 out of the 38 apps (i.e., apps from the available datasets) to evaluate ESDroid's correctness because (1) we checked the fault location in the source code for verification, and (2) the available datasets have open-source apps.

### Table 4

Information of buggy apps and exceptions for RQ1, RQ2, RQ3, and RQ4.

<table>
<thead>
<tr>
<th>App</th>
<th>Dex code size (KB)</th>
<th># of activities</th>
<th>Program version</th>
<th>Exception type</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addi</td>
<td>656.9</td>
<td>4</td>
<td>1.98</td>
<td>ActivityNotFoundException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>Anymemo</td>
<td>8887.5</td>
<td>27</td>
<td>10.9</td>
<td>NullPointerException</td>
<td>Su et al. (2020),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>and Tan et al. (2018)</td>
</tr>
<tr>
<td>APV PDF Viewer</td>
<td>63.1</td>
<td>3</td>
<td>0.26</td>
<td>NullPointerException</td>
<td>Alavi et al. (2019)</td>
</tr>
<tr>
<td>Bankdroid</td>
<td>5199.7</td>
<td>12</td>
<td>1.9</td>
<td>IllegalArgumentException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>Birthdroid</td>
<td>451.1</td>
<td>3</td>
<td>0.63</td>
<td>NumberFormatException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>Bites</td>
<td>489.9</td>
<td>5</td>
<td>1.3</td>
<td>NumberFormatException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>Calculator</td>
<td>2149.3</td>
<td>1</td>
<td>1</td>
<td>IllegalArgumentException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>CampFahrplan</td>
<td>3223.7</td>
<td>7</td>
<td>1.32</td>
<td>IllegalArgumentException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>Carnet - Notes app</td>
<td>5053</td>
<td>22</td>
<td>0.24</td>
<td>NullPointerException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>Coway</td>
<td>18.7</td>
<td>1</td>
<td>1.3</td>
<td>CalledFromWrongThreadException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>DalvikExplorer</td>
<td>5216</td>
<td>16</td>
<td>3.4</td>
<td>NullPointerException</td>
<td>Jiang et al. (2017)</td>
</tr>
<tr>
<td>Fdroid</td>
<td>5860.0</td>
<td>10</td>
<td>0.98</td>
<td>SQLiteException</td>
<td>Tan et al. (2018)</td>
</tr>
<tr>
<td>FishBun Demo</td>
<td>3293</td>
<td>7</td>
<td>0.62</td>
<td>NullPointerException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>fooCam</td>
<td>514.9</td>
<td>1</td>
<td>2.0</td>
<td>NullPointerException, SecurityException</td>
<td>Liu et al. (2016)</td>
</tr>
<tr>
<td>Geometric Weather</td>
<td>4393</td>
<td>14</td>
<td>2.113</td>
<td>ActivityNotFoundException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>GnuCash</td>
<td>7948.0</td>
<td>20</td>
<td>2.14</td>
<td>IllegalArgumentException</td>
<td>Tan et al. (2018)</td>
</tr>
<tr>
<td>LibreNews</td>
<td>3637.7</td>
<td>2</td>
<td>1.4</td>
<td>ArrayIndexOutOfBoundsException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>Linux Deploy</td>
<td>2156</td>
<td>8</td>
<td>2.60</td>
<td>IllegalArgumentException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>Man Man</td>
<td>3562</td>
<td>2</td>
<td>2.10</td>
<td>ActivityNotFoundException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>Mitzuli</td>
<td>3329.7</td>
<td>2</td>
<td>1.07</td>
<td>BadTokenException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>NPR News</td>
<td>17654.3</td>
<td>14</td>
<td>2.4</td>
<td>NullPointerException</td>
<td>Alavi et al. (2019)</td>
</tr>
<tr>
<td>OBSSD - OBS Stream Deck</td>
<td>4085</td>
<td>4</td>
<td>1.22</td>
<td>IllegalArgumentException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>Olam</td>
<td>715.4</td>
<td>1</td>
<td>1.0</td>
<td>SQLIteException, StringIndexOutOfBoundsException</td>
<td>Alavi et al. (2019), and Su et al. (2020)</td>
</tr>
<tr>
<td>PasswordMaker</td>
<td>33168</td>
<td>3</td>
<td>1.11</td>
<td>NumberFormatException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>Ringdroid</td>
<td>607.0</td>
<td>4</td>
<td>2.6</td>
<td>IllegalStateException</td>
<td>Jiang et al. (2017)</td>
</tr>
<tr>
<td>Scale Image View Demo</td>
<td>4277</td>
<td>1</td>
<td>4.0</td>
<td>ActivityNotFoundException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>Scribbler</td>
<td>20.4</td>
<td>3</td>
<td>0.18</td>
<td>IllegalFormatException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>SynMyPic</td>
<td>231.0</td>
<td>8</td>
<td>0.15</td>
<td>NoClassDeFoundError</td>
<td>Jiang et al. (2017)</td>
</tr>
<tr>
<td>Tailscale</td>
<td>2008</td>
<td>1</td>
<td>1.83</td>
<td>ActivityNotFoundException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>Tickmate</td>
<td>5919.6</td>
<td>6</td>
<td>1.20</td>
<td>CursorIndexOutOfBoundsException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>TippY</td>
<td>88.0</td>
<td>6</td>
<td>1.13</td>
<td>ArithmeticException</td>
<td>Jiang et al. (2017)</td>
</tr>
<tr>
<td>Transistor</td>
<td>2993.2</td>
<td>3</td>
<td>1.23</td>
<td>RuntimeException</td>
<td>Su et al. (2020),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>and Tan et al. (2018)</td>
</tr>
<tr>
<td>TripSit</td>
<td>2311.7</td>
<td>8</td>
<td>1.0</td>
<td>RuntimeException</td>
<td>Su et al. (2020)</td>
</tr>
<tr>
<td>Vanilla Music</td>
<td>1408</td>
<td>13</td>
<td>1.10</td>
<td>CursorIndexOutOfBoundsException, ResourcesNotFoundException</td>
<td>GooglePlay (2021)</td>
</tr>
<tr>
<td>WeightManager</td>
<td>5416</td>
<td>6</td>
<td>1.04</td>
<td>ActivityNotFoundException</td>
<td>Jiang et al. (2017)</td>
</tr>
<tr>
<td>WhoHasMyStuff</td>
<td>47.3</td>
<td>4</td>
<td>1.07</td>
<td>NullPointerException</td>
<td>Jiang et al. (2017)</td>
</tr>
<tr>
<td>Zahirize</td>
<td>27.4</td>
<td>2</td>
<td>1.1</td>
<td>NumberFormatException</td>
<td>Jiang et al. (2017)</td>
</tr>
</tbody>
</table>

- one app (i.e., fooCam) from RelFix (Liu et al., 2016).
- 13 apps (i.e., Addi, Bankdroid, Birthdroid, Bites, CampFahrplan, Coway, LibreNews, Mitzuli, PasswordMaker, Olam, Scribbler, Tickmate and TripSit) from DroidDefects (Su et al., 2020).
RQ1: What is the effectiveness of ESDroid in reducing the size of the input event sequence?

RQ2: Which of our two key phases (i.e., Phase 3 = Simplifying FSofE (Segment-based Delta Debugging), Phase 4 = backward dynamic slicing) contributes more to improve the debugging process?

RQ3: What is the difference in the size of dynamic slices computed by ESDroid and AndroidSlicer?

RQ4: Are slices computed by ESDroid and AndroidSlicer correct?

RQ5: What is the difference in the size of dynamic slices computed by ESDroid and Mandoline?

Specifically, the objective of RQ1 is to find the effectiveness of reducing the search space with delta debugging for Android apps. RQ2 highlights the phase which contributes the most to the whole process and the phase which contributes the least, aiming for future enhancement. RQ3 and RQ5 show the point of narrowing the search space compared to the state-of-art tools. The purpose of RQ4 is to check our contribution is usable in terms of quality.

5.2. RQ1: Size of input event sequence

RQ1 aims to evaluate our tool's effectiveness in reducing the input event sequence (failure-inducing event sequence) by comparing the size of event sequence between the original event sequence and the simplified event sequence. We use segment-based delta-debugging to minimize the randomly generated event sequence. Given the input event sequence Seq, we measure its length using the following metrics:

- **# of events**: Number of events triggered in Seq
- **# of callbacks**: Number of callback methods invoked in Seq
- **# of instructions**: Number of method calls invoked in Seq
- **# of method calls**: Number of simple instructions executed in Seq

Table 5 shows the comparison in size between the originally generated event sequence Seqorig and the minimized event sequence SeqESDroid. Meanwhile, the second and the third column under the title "# of events" denote the number of events triggered in Seqorig and SeqESDroid, respectively. The two columns under the title "# of callbacks" represent the number of callback methods invoked in Seqorig and SeqESDroid, respectively. The two "# of method calls" columns denote the number of methods invoked in Seqorig and SeqESDroid. (note that "# method calls" counts all method calls, including all callback methods). The two "# of Instructions" columns denote the number of instructions executed in Seqorig and SeqESDroid. The "Duration (seconds)" column in Table 5 presents the time taken in seconds to perform the minimization using segment-based delta-debugging. This table shows our segment-based delta-debugging can effectively minimize the number of events for all evaluated apps (the minimized # of events ranges from 1–26 compared to the original # of events that ranges from 3–1097). On average, ESDroid can reduce 87% for # of events, 42% for # of callbacks, 42% for # of method calls and 45% for # of instructions with the average execution time in 3354 s.

We observed that two factors affect the reduction rate: (1) the GUI states, (2) the redundant events. Firstly, the simplicity of the GUI states is inversely proportional to the reduction rate for an app. If the app has many buttons on a single GUI screen, the probability of triggering the crash that requires specific ordering of event sequences is low, and the reduction rate for an app is high. In contrast, if an app has fewer buttons on a single GUI screen, it is easy to trigger the crash and has a lower reduction rate. In other words, if an app's GUI is designed in a simple way (with fewer GUI components), the reduction benefit can be less than that of a complex GUI design. Secondly, the redundant events with executed statements that do not affect the point of interest can also introduce many spurious nodes and edges on a dynamic PDG. As shown in Table 5, we found that Transistor has the highest reduction rate because the failed test case for Transistor selects an item from the long options menu that generates redundant events. Specifically, the original event sequence for Transistor has 15 callback events, including the callback event (i.e., onOptionsItemSelected) that is repeated six times, and five of them are redundant. Moreover, the Calculator app has the second-highest reduction rate because it has only one GUI screen and 18 buttons are occupying almost one-fourth of the whole screen. Therefore, it is difficult to get the event sequence to cause the app to crash and generates redundant events. Specifically, the original event sequence for Calculator has 43 callback events consisting of the callback event (i.e., onClickNumber) that is repeated 23 times, and all of them are redundant. Similarly, the test case for Cowsay has 64 callback events, including the callback event onTextChanged that repeated 18 times, and all of them are redundant. Moreover, none of them has the statements that affect the point of interest.

In contrast, the reduction rate for APV PDF Viewer is the lowest among all evaluated apps. During our manual analysis, we found that it has one GUI screen with only seven items in ListView, and it is easy to generate the failing test case with nine input events, and four of them are failure-inducing events. Specifically, the original event sequence for APV PDF Viewer has only five callback events, and two of them are required to generate the failing test case. Moreover, Cowsay has the second lowest reduction rate. Cowsay is an English–Malayalam dictionary and it searches for the definitions of English/Malayalam words. We found out that it sets focus on EditText and IME keyboard is up when the app is launched. Therefore, although the IME keyboard occupies half of the GUI screen, it is easy to generate the failed test case because the cursor position is already defined and the crash can be triggered easily. Specifically, the original event sequence for Cowsay has four callback events, and all of them are required to cause the app to fail.

In terms of processing time, we observe that it takes a longer time to minimize (1) if there are many input events in the event sequence in different Activities and (2) if the failure-inducing events with corresponding GUI states in the event sequence are in different sub-sequences while the original event sequence is divided. As shown in Table 5, WhoHasMyStuff and GnuCash have the longest processing time for the reduction in the experiment. Specifically, for WhoHasMyStuff, the original sequence that makes the app fail has the 26 failure-inducing events, and its corresponding GUI states are in different sub-sequences. For GnuCash, the originally generated event sequence that makes the app crash contains six different Activities. However, the basic strategy of delta debugging is already robust and effective enough to obtain a large reduction rate. Exercising more strategies (e.g., hierarchical delta debugging) could be an interesting future topic.

5.3. RQ2: Effectiveness of different phases in ESDroid

To evaluate which phases contributed to the overall reduction of our approach (reducing the search space), we computed the number of executed instructions for each phase in Fig. 2. Fig. 6 shows the reduction results for 41 defects of 38 apps.
The rowsof Table 6 and Fig. 6 show that the reduction rate in phase 4 is higher than in phase 3. At phase 4, the maximum reduction rate is 99% (i.e., Fdroid, Calculator, FishBun Demo, Geometric Weather, OBSSD - OBS Stream Deck, Official Cambridge Guide to IELS, Scale Image View Demo) and the minimum is 36% (i.e., Yahtzee). At phase 3, 0% reduction rate for two apps (i.e., Olam and Yahtzee) because the original event sequences and the simplified event sequences in phase 2 and phase 3 are identical, and the number of methods and callbacks invoked are identical. However, in phase 4, when ES Droid slices all the executed instructions affecting the point of interest, the reduction rate becomes more than 0% (i.e., 99% for Olam and 36% for Yahtzee). Therefore, phase 4 contributes more to the overall optimization than phase 3.
5.4. RQ3: Difference in the size of dynamic slices computed by ESDroid and AndroidSlicer

We compare the effectiveness of ESDroid against AndroidSlicer by measuring the sizes of the dynamic slices produced by the two approaches. Employing the following metrics, we evaluated the effectiveness of the two approaches:

**S1: # of executed Jimple lines**: The number of Jimple instructions in the dynamic slice

**S2: Time**: Time taken to perform dynamic slicing

Fig. 7(a) shows the number of Jimple instructions in the generated slice of both approaches (i.e., AndroidSlicer, and ESDroid).
Table 6
Output comparison (#) between three phases in ESDroid (i.e., Phase 2 = Producing Failure-inducing Sequence of Events (FSoE), Phase 3 = Simplifying FSoE (Segment-based Delta Debugging), Phase 4 = Backward dynamic slicing). The values in the fourth column with the title (i.e., (%)(1)) and the sixth column (i.e., (%)(2)) are calculated by using the matrix (1) and matrix (2), respectively.

<table>
<thead>
<tr>
<th>Apps</th>
<th># of instructions</th>
<th>Phase 2</th>
<th>Phase 3</th>
<th>(%)(1)</th>
<th>Phase 4</th>
<th>(%)(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addi</td>
<td>136 938</td>
<td>79 722</td>
<td>42</td>
<td>721</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Anymemo</td>
<td>464 816</td>
<td>428 269</td>
<td>8</td>
<td>2222</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>APV PDF Viewer</td>
<td>948</td>
<td>347</td>
<td>63</td>
<td>49</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>Bankdroid</td>
<td>155 151</td>
<td>102 113</td>
<td>34</td>
<td>522</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Birthdroid</td>
<td>905</td>
<td>893</td>
<td>1</td>
<td>100</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>Bites</td>
<td>2 109</td>
<td>389</td>
<td>82</td>
<td>91</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>Calculator</td>
<td>34 974</td>
<td>8 983</td>
<td>74</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CampFahrplan</td>
<td>1 250 075</td>
<td>268 769</td>
<td>78</td>
<td>1010</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Carnet - Notes app</td>
<td>398 939</td>
<td>223 098</td>
<td>44</td>
<td>6697</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>Cowsay</td>
<td>4 345</td>
<td>1 610</td>
<td>63</td>
<td>86</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>DalvikExplorer</td>
<td>8 502</td>
<td>1 014</td>
<td>88</td>
<td>184</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>Fdroid</td>
<td>2 678 033</td>
<td>1 878 576</td>
<td>80</td>
<td>1731</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>FishBun Dem</td>
<td>86 049</td>
<td>19 959</td>
<td>77</td>
<td>262</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>fooCama</td>
<td>811</td>
<td>405</td>
<td>50</td>
<td>31</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>fooCamb</td>
<td>909</td>
<td>500</td>
<td>45</td>
<td>132</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>GeometricWeather</td>
<td>585 048</td>
<td>332 065</td>
<td>43</td>
<td>2036</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>GnuCash</td>
<td>936</td>
<td>922</td>
<td>1</td>
<td>60</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>LibreNews</td>
<td>636 956</td>
<td>232 652</td>
<td>63</td>
<td>163</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Linux Deploy</td>
<td>1 294 897</td>
<td>541 846</td>
<td>58</td>
<td>8591</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>ManMan</td>
<td>1 176 582</td>
<td>121 901</td>
<td>90</td>
<td>5980</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Mirzuli</td>
<td>1 230 777</td>
<td>905 576</td>
<td>26</td>
<td>1203</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>NPR News</td>
<td>25 597</td>
<td>10 107</td>
<td>51</td>
<td>412</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>OBSSD - OBS Stream Deck</td>
<td>4 754 980</td>
<td>1 200 819</td>
<td>75</td>
<td>9821</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Official Cambridge Guide to IELS</td>
<td>537 874</td>
<td>355 432</td>
<td>34</td>
<td>1275</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Olam1</td>
<td>26 597</td>
<td>26 597</td>
<td>0</td>
<td>295</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Olam2</td>
<td>42 324</td>
<td>42 324</td>
<td>0</td>
<td>148</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>PasswordMaker</td>
<td>30 969</td>
<td>18 225</td>
<td>41</td>
<td>839</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Ringdroid</td>
<td>26 263</td>
<td>2 500</td>
<td>90</td>
<td>390</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Scale Image View Demo</td>
<td>295 088</td>
<td>52 468</td>
<td>82</td>
<td>182</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Scrubber</td>
<td>132</td>
<td>83</td>
<td>37</td>
<td>38</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>SyncMyPic</td>
<td>356</td>
<td>275</td>
<td>33</td>
<td>23</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Tailscale</td>
<td>174 283</td>
<td>120 950</td>
<td>31</td>
<td>1842</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Tickmate</td>
<td>3 491</td>
<td>3 093</td>
<td>11</td>
<td>152</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>Tippy</td>
<td>3 775</td>
<td>2 406</td>
<td>36</td>
<td>317</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>Transistor</td>
<td>1 625</td>
<td>1 242</td>
<td>24</td>
<td>104</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>TripSit</td>
<td>1 488</td>
<td>1 313</td>
<td>9</td>
<td>51</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>Vanilla Music1</td>
<td>92 527</td>
<td>83 118</td>
<td>10</td>
<td>3171</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>WeightChart</td>
<td>17 432</td>
<td>1 711</td>
<td>90</td>
<td>292</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>WhoHasMyStuff</td>
<td>10 820</td>
<td>604</td>
<td>94</td>
<td>159</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Yahzee</td>
<td>33</td>
<td>33</td>
<td>0</td>
<td>21</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Vanilla Music2</td>
<td>153 206</td>
<td>93 073</td>
<td>39</td>
<td>3300</td>
<td>98</td>
<td></td>
</tr>
</tbody>
</table>

Mean 392 780  
174 779  
45 1336  
94

*The app refers to the defect that throws NullPointerException.
*The app refers to the defect that throws SecurityException.
*The app refers to the defect that throws SQLiteException.
*The app refers to the defect that throws StringIndexOutOfBoundsException.
*The app refers to the defect that throws CursorsNotOpenedException.
*The app refers to the defect that throws ResourcesNotFoundException.

whereas Fig. 7(b) compares the time taken by each approach in generating the dynamic slice. The numbers given beside the bars in Figs. 7(a) and 7(b) show the reduction rate (in percentage) for the size of the slices and the time taken in generating the dynamic slice, respectively. Overall, our results in Fig. 7(a) show that ESDroid is able to produce a thinner slice compared to AndroidSlicer for all the evaluated apps, except for APV PDF Viewer, Olam and Yahzee. For these apps, ESDroid fails to reduce the slice because the event sequence leading to the exception has fewer than five extra events, and there is no data or control-dependence found among these extra event sequences. ESDroid and AndroidSlicer shared common instrumentation performance by employing the same instrumentation using Soot. We further analyzed the results reported in Fig. 7a using statistical and effect size tests. In particular, we used the Wilcoxon rank sum test (Conover, 1999) and the Vargha–Delaney’s A12 effect size (Vargha and Delaney, 2000). We used the Wilcoxon test to assess whether the differences in the number of Jimple instructions between AndroidSlicer and ESDroid are statistically significant. We considered the level of significance to be α = 0.05. According to the Wilcoxon tests, the slices generated by ESDroid are statistically significant smaller than the slices generated by AndroidSlicer (p-value < 0.00001). The Vargha–Delaney’s A12 measure reports a medium effect size A12 = 0.56.

Although ESDroid can produce a thinner slice than AndroidSlicer, the results in Fig. 7(b) show that the overall time taken by both approaches to perform the dynamic slicing is similar (i.e., from 0% to 29%). These results illustrate the efficiency of our algorithm in performing dynamic slicing without incurring too much additional overhead. In fact, for the Fdroid, ESDroid can generate the dynamic slice faster than AndroidSlicer because the size of the trace log (i.e., executed instructions) for Fdroid is the largest of all the apps used in our experiment and the analysis time (i.e., checking against static PDG) shows longer
duration. In general, the test case with more redundant events with statements that do not affect the failure point is more likely to include spurious slices (e.g., Calculator, DalvikExplorer and fooCam). Moreover, even with a smaller number of callbacks and events in our experiments, ESDroid still reduced a substantial portion of the redundant PDG nodes. We believe increasing the events will favor ESDroid even further.

5.5. RQ4: Correctness of slices computed by ESDroid and Android-Slicer

In this section, we aim to ensure the output of our approach is useful in locating the bug. Since our approach does not require the source code, we manually examined the apps to assess precision using bytecode. We decompiled each app to get the Java bytecode and mapped the Jimple instruction to the program statement via the program line number. We then manually checked the slices related to the slicing criterion with the following three steps:

1. Instruction — We checked which instructions were related to the failure point (the point of interest).

2. Method — We investigated which particular call paths qualified for the above instructions. Specifically, we examined what corresponding methods were required.

3. Segment — As we recorded the execution history using the segment, we also analyzed the program by checking which segments enabled the methods mentioned above to ensure each segment reflected the required state and events for the app’s crashes.

Then, we compared the extracted information with the slice generated by ESDroid. We checked all generated slices manually to ensure that our slice computation was correct. In addition, to make sure that the slices produced by ESDroid included the instructions related to the failure point, we manually analyzed the differences between the output of ESDroid and the output of AndroidSlicer. Our analysis confirmed that the slices generated by both ESDroid and AndroidSlicer included the statements affecting the failure point. Since both ESDroid and AndroidSlicer include the instructions related to the failure point, a thinner slice generated by ESDroid is a better outcome because it reduces the time
taken by the developers to inspect the slice during debugging to state one enhancement.

5.6. RQ5: Difference in the size of dynamic slices computed by ESDroid and Mandoline

Table 7: Information of buggy apps and exceptions for RQ5.

<table>
<thead>
<tr>
<th>App</th>
<th>Dex code size (KB)</th>
<th># of activities</th>
<th>Program version</th>
<th>Exception type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anki</td>
<td>4490</td>
<td>21</td>
<td>1</td>
<td>FileUriExposedException</td>
</tr>
<tr>
<td>Birthdroid</td>
<td>431.1</td>
<td>3</td>
<td>0.6.3</td>
<td>NullPointerException</td>
</tr>
<tr>
<td>Fastadapter</td>
<td>6376</td>
<td>23</td>
<td>2.5.1</td>
<td>SQLliteException</td>
</tr>
<tr>
<td>Fdroid</td>
<td>5860.0</td>
<td>10</td>
<td>0.98</td>
<td>IllegalArgumentException</td>
</tr>
<tr>
<td>GnuCash</td>
<td>7948.0</td>
<td>20</td>
<td>2.1.4</td>
<td>ActivityNotFoundException</td>
</tr>
<tr>
<td>K9</td>
<td>4684</td>
<td>29</td>
<td>1</td>
<td>NumberFormatException</td>
</tr>
<tr>
<td>Micromath</td>
<td>4927</td>
<td>2</td>
<td>1</td>
<td>NumberFormatException</td>
</tr>
<tr>
<td>Newsblur</td>
<td>3828</td>
<td>36</td>
<td>1</td>
<td>NullPointerException</td>
</tr>
<tr>
<td>SiliCompressor</td>
<td>2153</td>
<td>1</td>
<td>1.1.0</td>
<td>ArithmeticException</td>
</tr>
<tr>
<td>Specialdates</td>
<td>2149</td>
<td>11</td>
<td>1</td>
<td>IllegalFieldValueException</td>
</tr>
</tbody>
</table>

5.7. Threats to validity

We identify the following threats to the validity of our evaluation:

**Internal validity:** For random test case generation, ESDroid supports events that simulate clicks, rotations, and drags but does not support complex events like GUI text input and system events. This limitation may affect the internal validity of this work and impact the results. In future, we plan to improve our tool to support complex events. This is not a limitation of our slicer but rather on our random test generation. Despite the removal of the majority of spurious slices, the precision of ESDroid depends on the precision of its underlying static analysis. Specifically, we implemented our instrumentation on top of Soot and FlowDroid. Moreover, while there are several slicing approaches, we only compare our approach against AndroidSlicer, and Mandoline because, to the best of our knowledge, they are the only dynamic slicing techniques for Android and their tools are publicly available. Moreover, as the available implementation of Mandoline throws NoSuchElementException, and NullPointerException for some apps because Mandoline does not consider the control dependence among the lifecycle callbacks. This leads to the unsatisfactory paths in the dependence graph. We thus contribute an enhanced version of Mandoline, called Mandoline++, which addresses the Mandoline implementation issue. Table 7 shows the apps we evaluated. The “Exception Type” column contains information about the specific type of exception that causes the crash. We compare the effectiveness of ESDroid against Mandoline by measuring the sizes of the dynamic slices produced by the two approaches. The available implementation of Mandoline throws NoSuchElementException, and NullPointerException for some apps because Mandoline does not consider the control dependence among the lifecycle callbacks. It leads to the unsatisfactory paths in the dependence graph. We thus contribute an enhanced version of Mandoline, called Mandoline++, which addresses the Mandoline implementation issue. Table 8 shows the slice size (#JS) (number of Jimple instructions) for the slice produced by each of the tools (columns 2, 3, and 4). The column with (%) is the reduction rate from Mandoline++ to ESDroid.

ESDroid outperforms Mandoline++ in terms of reducing the slices in six apps and performs equivalently in the remaining four: Anki, Fastadapter, Micromath and Newsblur. ESDroid cannot achieve a higher reduction rate for four apps; we observed that the events in the randomly generated event sequence (i.e., failure-inducing sequence of events) are the same as the simplified event sequence. Overall, ESDroid can produce up to 50% thinner slices than Mandoline. We also observed a similar finding with RQ3 that the test case with more redundant events with statements that do not impact the failure point is more likely to include spurious slices. On average, ESDroid can reduce 18% for # of Jimple instructions in the slice. We further analyzed the results using statistical and effect size tests. We used the Wilcoxon test to assess whether the differences in the number of jimple instructions between Mandoline++ and ESDroid are statistically significant. Based on the Wilcoxon test, we found that the result is statistically significant (p-value < 0.05). The Vargha–Delaney’s A12 measure reports a medium effect size A12 = 0.55.
our approach is unable to handle non-crash bugs and also unable to conduct slicing for obfuscated apps (e.g., whose bytecode is transformed using reflection), which might lead to imprecise slicing results. In addition, our study is limited to the evaluated Android apps and our results may not be able to be generalized beyond them. We mitigate this threat by (1) including closed-source Android apps with bugs, and (2) obtaining Android apps from five different data sets (Tan et al., 2018; Jiang et al., 2017; Alavi et al., 2019; Liu et al., 2016; Su et al., 2020).

6. Related work

**Delta-Debugging:** Several approaches have applied delta-debugging to identify the failure-inducing deltas in traditional desktop applications (Yu et al., 2012; Gupta et al., 2005), compilers (Mishgerhi and So, 2006), browsers (Zeller and Hildebrandt, 2002), Web applications (Hammoudi et al., 2015), and microservice systems (Zhou et al., 2018). However, these approaches are not designed for handling the asynchronous event nature of Android apps, where they become ineffective in detecting event sequences. For Android apps, several algorithms based on delta-debugging have been proposed to minimize GUI event sequences for reaching a particular target activity (Clapp et al., 2016), and for reproducing a crash SimplyDroid (Jiang et al., 2017). The end goal of this work is completely different from SimplyDroid.

The objective of our approach is to conduct more precise dynamic slicing to produce a more compact and precise program dependence graph, while SimplyDroid aims to simplify crash traces. Second, SimplyDroid treats an app as a black box and does not perform code analysis on Android bytecode or source code, while our slicing approach does. Though both approaches use delta-debugging, we use delta-debugging as a means to an end, but not an end. This paper makes a step forward by introducing segment-based delta debugging in backward dynamic slicing to reduce search space, yielding a thinner slice that includes the effective statements on the failure point at the bytecode level.

**Slicing for Web applications:** Several techniques have been proposed for slicing in Web applications (Maras et al., 2011; Tonella and Ricca, 2005). Although Web applications share similar event-based execution paradigms with Android apps, the event’s nature in the Web application and the nature of the event of Android apps are different. Unlike Web applications, Android apps pose unique challenges to slicing with (1) life cycle management rules among components (for example, Fragment and Activity), and (2) intercomponent communication employed not only in the same application but also across different applications.

**Slicing for Java:** Slicing for traditional Java programs (Wang and Roychoudhury, 2008) has been investigated. Unlike traditional Java, Android has several entry points via various channels, and calls to other processes within applications or external applications. It can be undertaken in both an explicit and implicit way. Given an automatic test case (in the form of event sequences), ESDroid takes account of the characteristics of Android apps to produce a reduced program slice.

**Fault Localization for Android Apps:** Traditional spectrum-based fault localization techniques perform statistical analysis on program execution traces to produce a ranked list of suspicious statements (i.e., statements that are relevant to the root cause of a defect) (Parin and Orso, 2011; Jones and Harrold, 2005; Wong et al., 2016; Pearson et al., 2017; Li et al., 2019). To handle the unique characteristics of Android apps, MZoltar (Machado et al., 2013) performs spectrum-based fault localization on instrumented apps. Different from MZoltar and other spectrum-based fault localization approaches, ESDroid (1) does not rely on the existence of passing tests (which may not be available for Android apps) to pinpoint the faulty location, and (2) produces a program slice where each statement within the same slice shared the same rank rather than a ranked list of suspicious statements.

**Slicing for Android Apps:** Several slicing approaches have been designed for Android Apps (Hoffmann et al., 2013; Alavi et al., 2019; Ahmed et al., 2021). SAAF (Hoffmann et al., 2013) performs static slicing to detect suspicious behavior patterns for malicious Android apps. Meanwhile, AndroidSlicer performs dynamic slicing by modeling asynchronous data and the control dependences of Android apps. Mandoline presents dynamic slicing via alias analysis. In much the same way as AndroidSlicer, ESDroid uses dynamic slicing to produce the program slices that aid debugging for Android apps. ESDroid differs from AndroidSlicer, and Mandoline in that (1) it offers a fully automated approach for minimizing the event sequences to produce the final program slices, (2) it considers the control dependences among the lifecycle callbacks, (3) our experiments show that ESDroid can produce a thinner slice than AndroidSlicer, and Mandoline.

**Automated program repair for Android Apps:** Many automated techniques have been proposed to generate patches to fix bugs in Android apps (Dilhara et al., 2018; Kong et al., 2019; Liu et al., 2016; Marginean et al., 2019; Xu, 2019; Tan et al., 2018). Our dynamic slicing approach is orthogonal to these automated bug-fixing approaches and can be combined with them to improve debugging process and subsequently generate high-quality patches.

7. Conclusion and future work

We, for the first time, introduce delta-debugging into dynamic slicing for Android to significantly boost its precision, as confirmed in our experiments. Our dynamic slicing supports control- and data-dependence at both the instruction-level and event-level by leveraging the simplified input event sequence that triggers the same bug using segment-based delta-debugging. ESDroid is able to produce a more precise but smaller dynamic PDG with up to 72% (27% on average) fewer false executed instructions than the state-of-the-art AndroidSlicer, and up to 50% (18% on average) fewer than Mandoline, while maintaining only the relevant buggy statements to capture precisely the same bugs as AndroidSlicer and Mandoline. In the future, we plan to enhance ESDroid to handle non-crashing bugs with oracle by exercising more strategies (e.g., hierarchical delta debugging), and including test cases with complex interactions such as GUI text input and system events.

**Declaration of competing interest**

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: This work was supported by the Southern University of Science and Technology, China (SUSTech) - University of Technology, Sydney, Australia (UTS) Joint Ph.D. Program.

**Acknowledgments**

The authors would like to thank the reviewers for their insightful feedback. This work was supported by the National Natural Science Foundation of China, China (Grant No. 61902170) and Australian Research, Australia Grants (DP200101328 and DP210101348).

**References**

Hussein M.Y. Win is currently a Ph.D. student enrolled in the SUSTECH-UTS Joint Ph.D program. She is supervised by Dr. Yulei Su from UTS and Dr. Shin Hwei Tan from Southern University of Science and Technology, China (SUSTech). She received her M.Tech (Software Engineering) degree from the National University of Singapore (NUS) and her B.Tech. degree from the University of Computer Studies, Yangon, Myanmar (UCSY). Her main research interest is android testing and program analysis.

Shin Hwei Tan is a tenure-track Assistant Professor at Southern University of Science and Technology in Shenzhen, China. She obtained her Ph.D. degree from National University of Singapore and her B.S(Hons) and M.Sc. degree from UILC. Her main research interest includes automated program repair, software testing and search-based software engineering. She received several prestigious awards, including Distinguished Program Committee Members of ASE 2020, David J. Kuck Outstanding M.Sc. Thesis Award, Google Anita Borg Memorial Scholarship. She has served as PCs for several top-ranked conferences (ICSE, ASE, and VSTTE) and reviewers for several journals (TOSEM, TSE and EMSE). She also co-organized the 6th International Workshop on Genetic Improvement (co-located with ICSE 2019) and founded the first International Workshop on Automated Program Repair (APR 2020).
Yulei Sui is a Senior Lecturer a.k.a an Associate Professor at the School of Computer Science, Faculty of Engineering and Information Technology, University of Technology Sydney (UTS). He is broadly interested in Program Analysis, Secure Software Engineering, and Machine Learning. In particular, his research focuses on building fundamental static and dynamic analysis techniques and tools to improve the reliability and security of modern software systems. His recent interest lies at the intersection of programming languages, natural languages, and machine learning. Specifically, his current research projects include secure machine learning, program analysis for bug detection and repair through data mining and deep learning.

His papers have been published in the top-tier conferences and journals in the field of software engineering and program analysis such as TSE, TOSEM, ICSE, FSE, OOPSLA, ECOOP, ISSTA, ASE, SAS, CGO, and CC. He was a plenary talk speaker at EuroLLVM 2016 and has been awarded a 2021 ICSE Distinguished Reviewer, 2020 OOPSLA Distinguished Paper, a 2019 SAS Best Paper, a 2018 ICSE Distinguished Paper, a 2013 CGO Best Paper, and an ARC Discovery Early Career Researcher Award (2017–2019).