The Journal of Systems & Software 198 (2023) 111606

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

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

ARTICLE INFO

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

ABSTRACT

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).

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 $s_i \xrightarrow{d} s_j$ means any computation performed in s_i depends on the computed value at node s_j. A control dependence edge $s_i \xrightarrow{c} s_j$ 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

☆ Editor: Fabio Palomba.
* Corresponding author at: Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, Guangdong, 518055, PR China.
** Corresponding author.
E-mail addresses: hsumyat.win@student.uts.edu.au (H.M. Win), tansh3@sustech.edu.cn (S.H. Tan), yulei.sui@uts.edu.au (Y. Sui).

https://doi.org/10.1016/j.jss.2023.111606

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 $\langle t, s, value\rangle$ (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 ICCs) 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,[1] 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 bye-code level, even when no source code is available. Fig. 1a is the simplified app code of SiliCompressor, and Fig. 1b 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 *maxRatio* by dividing *width* by *height*.

The app fails when the user decreases the value of *height* to zero and calculates for *maxRatio*, making "divide by zero", which leads to the `ArithmeticException` (Line 15). Regardless of the integer value in the object of *width*, if the value in the object of *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 *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. ESDroid 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 `eventID` to 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

[1] https://github.com/Tourenathan-G5organisation/SiliCompressor, https://github.com/Tourenathan-G5organisation/SiliCompressor/issues/10.

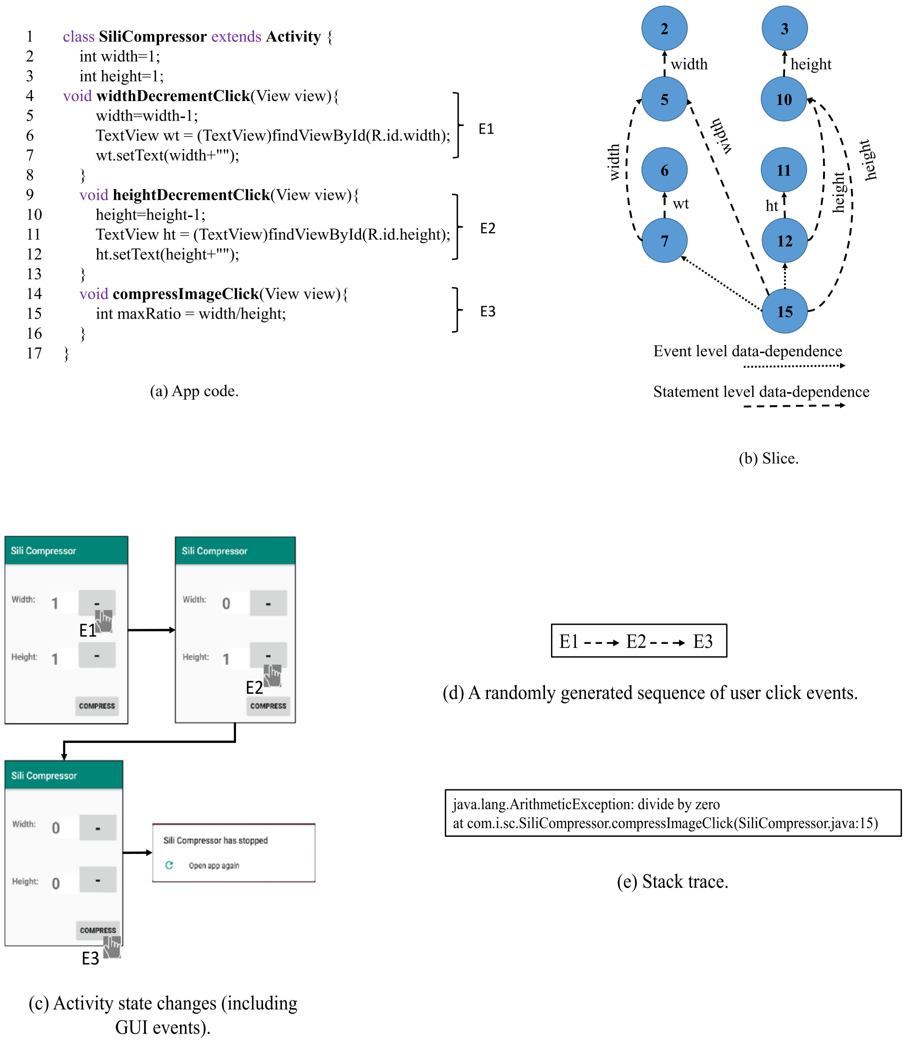


Fig. 1. Our Motivation. SiliCompressor app. ArithmeticException has thrown while the program attempted to divide by zero.

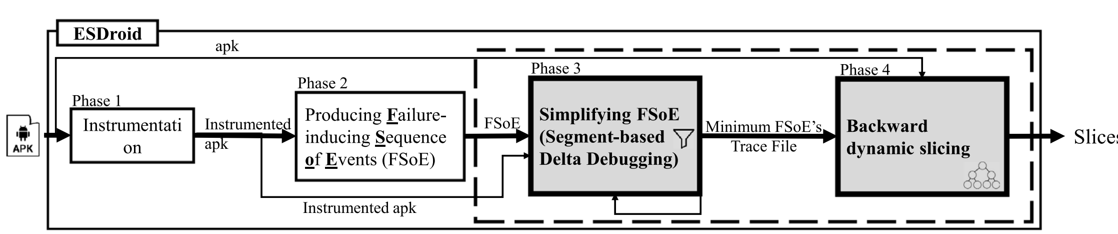


Fig. 2. Overview architecture of ESDroid.

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 (ΔFSoE). 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 ΔFSoE 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.

2 https://developer.android.com/studio/test/monkeyrunner.

Table 1
Iteration process of simplifying FSoE for Fig. 3 - Motivating example.

Iteration	Sequence of events	Value stored in the object at the timestamp once after the last event is triggered.			Test result	Remarks
		width	*height*	*maxRatio*		
0	E1⇢E2⇢E3	0	0	ArithmeticException	Fail	Original FSoE.
1	E2⇢E3 E1	1 –	0 –	ArithmeticException –	Fail –	Divide the original event sequence (i.e., FSoE) into two sub-sequences and test the last sub-sequence (E2⇢E3) and the test failed. Bring the failed sub-sequence.
2	E3 E2	1 1	1 0	1 –	Pass Pass	Divide the last failed event sequence into two sub-sequences and test both sub-sequences. The reduction finished with 1-minimal event. The latest test which made the app fail is ΔFSoE (E2⇢E3).

- 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 (GitHub,[3] Zenodo[4]).

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` callback is invoked upon clicking "-" for width on app screen. The callback `heightDecrementClick` is invoked when clicking "-" sign for height. `compressImageClick` is invoked upon clicking on "COMPRESS". Fig. 3(c) shows the simplified failure-inducing event sequence (i.e., ΔFSoE) 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

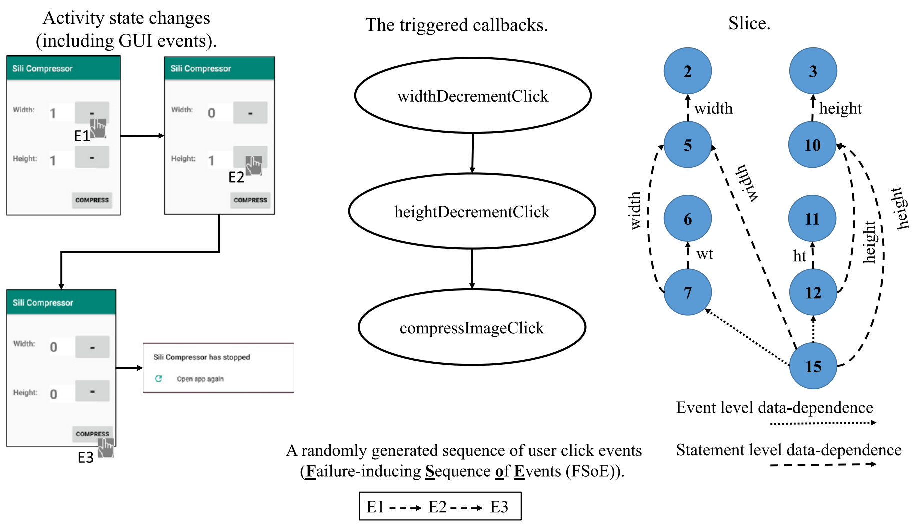
Our goal is to reduce the size of the event sequence, which triggers an exception, and to produce a more precise and compact program slice. ESDroid gradually removes some redundant events from the event sequence using segment-based delta-debugging. This is done iteratively by exercising a sub-sequence 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.

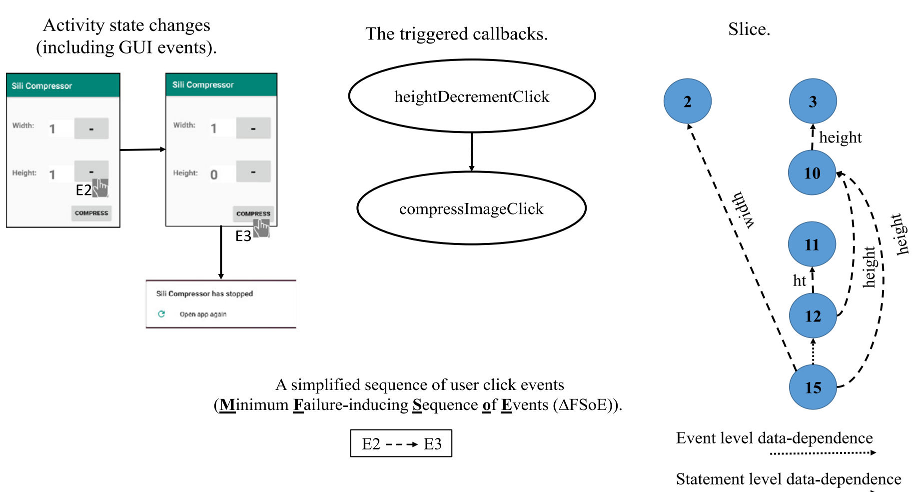
[3] https://github.com/hsumyatwin/ESDroid-artifact.
[4] https://doi.org/10.5281/zenodo.7074680.

```
class SiliCompressor extends Activity {
  int width=1;
  int height=1;
void widthDecrementClick(View view){
    width=width-1;
    TextView wt = (TextView)findViewById(R.id.width);      E1
    wt.setText(width+"");
  }
  void heightDecrementClick(View view){
    height=height-1;
    TextView ht = (TextView)findViewById(R.id.height);     E2
    ht.setText(height+"");
  }
  void compressImageClick(View view){                      E3
    int maxRatio = width/height;
  }
}
```

(a) App code abstracted from `SiliCompressor`.



(b) Slice produced by AndroidSlicer for the exception (i.e., ArithmeticException: divide by zero).



(c) Slice produced by ESDroid for the same exception (i.e., ArithmeticException: divide by zero).

Fig. 3. A motivating example (i.e., SiliCompressor app). ArithmeticException has thrown while the program attempted to divide by zero. The reduction process for event sequences in Table 1.

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., ΔFSoE) 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 *maxRatio*, we perform backward dynamic slicing on both the original test case (i.e., FSoE) and the simplified test case (i.e., ΔFSoE). The criteria we used are (1) the timestamp when the exception is thrown, (2) the object holding error (*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, ΔFSoE). 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 *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), $r3 (i.e., height).
09-21 00:23:51.027 System.out:ESDroid_ID4_15_com.i.sc.SiliCompressor_compressImageClick_$r4=$r2/$r3;

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., $\triangle$FSoE), 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 error/exception type but a different stack trace.
- The app crashed with the same error/exception type, and the same stack trace.

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 test result 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 $s \subseteq s_{\text{✗}}$ is n-*minimal* if $\forall s' \subset s \cdot |s| - |s'| \leq n \Rightarrow (\text{test}(s') \neq \text{✗})$ holds, where ✗ is the fail outcome. Consequently, s is 1-*minimal* if $\forall \delta_i \in s \cdot \text{test}(s - \{\delta_i\}) \neq \text{✗}$ 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 and $x \notin$ 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 *Tf*.

```
Tf ⟵ {};
n ⟵2;
isFailed ⟵ false;
if FSoE.size() ==1 then
    (isFailed, Tf) ⟵ test( FSoE, e);
end
while FSoE.size() >=2 do
    S ⟵ divide FSoE into n sub-sequences S1, S2, S3,...,
        Sn;// Divide the event sequence into n (i.e.,
        granularity) sub-sequences equally. 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.
    for each sub-sequence Si in S do
        if test( FSoE\Si, e) != null then
            (isFailed, Tf) ⟵ test( FSoE\Si, e);
        end
        if isFailed then
            FSoE ⟵ FSoE\Si;
            n ⟵ max(n - 1, 2);
            break;
        end
    end
    if !isFailed then
        if n == FSoE.size() then
            break;
        end
        n ⟵ min(2n, FSoE.size());// Increase granularity
            and start Complement strategy
    end
    isFailed ⟵ false;
end
return Tf;

Procedure: test(list of events St, stack trace e)
if St triggers the app crash then
    x ⟵ dumpStack();// print stack trace of crash
    if e == x then
        Tf ⟵ logcat();// get all executed program
            statements
        return (true,Tf);
    end
end
return 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., $\triangle$FSoE). 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

Table 2
Iteration process of simplifying FSoE for Example 2 (The `Divide and Conquer` strategy).

Iteration	Sequence of events	Test result	Remarks
0	E1⇢E2⇢E3⇢E4	Fail	Original FSoE.
1	E3⇢E4 E1⇢E2	Fail –	Divide the original event sequence (i.e., FSoE) into 2 sub-sequences, test the second sub-sequence (E3⇢E4). The test failed. Bring forward the failed sub-sequence to the next iteration.
3	E4 E3	Fail –	Divide the last failed sub-sequence into 2 sub-sequences and test the last sub-sequence (E4). The test failed. 1-minimal with failed sub-sequence is ΔFSoE.

Table 3
Iteration process of simplifying FSoE for Example 3 (The `Complement` strategy).

Iteration	Sequence of events	Test result	Remarks
0	E1⇢E2⇢E3⇢E4⇢ E5⇢E6⇢E7⇢E8	Fail	Original FSoE
1	E5⇢E6⇢E7⇢E8 E1⇢E2⇢E3⇢E4	Pass Pass	Divide the original event sequence (i.e., FSoE) into 2 sub-sequences and test both sub-sequences and both passed. Increase the granularity from 2 to 4.
2	E3⇢E4⇢E5⇢E6⇢ E7⇢E8 E1⇢E2⇢E5⇢E6⇢ E7⇢E8 E1⇢E2⇢E3⇢E4⇢ E7⇢E8 E1⇢E2⇢E3⇢E4⇢ E5⇢E6	Fail – – –	Divide the last failed event sequence into 4 sub-sequences and test the complement of last sub-sequence (E3⇢E4⇢E5⇢E6⇢ E7⇢E8). The test failed. Bring the failed complement to the next iteration with granularity 3 (i.e., max(4-1,2)).
3	E5⇢E6⇢E7⇢E8 E3⇢E4⇢E7⇢E8 E3⇢E4⇢E5⇢E6	– Fail –	Divide the last failed event sequence into 3 sub-sequences and skip the first complement (i.e., the second sub-sequence of iteration 1) and test the second complement (E3⇢E4⇢E7⇢E8). The test failed. Bring the failed complement to the next iteration with granularity 2 (i.e., max(3-1,2)).
4	E7⇢E8 E3⇢E4	Pass Pass	Divide the last failed event sequence into 2 sub-sequences and test both complements. Both passed. Increase the granularity from 2 to 4.
5	E4⇢E7⇢E8 E3⇢E7⇢E8 E3⇢E4⇢E8 E3⇢E4⇢E7	Pass Pass Pass Pass	Divide the last failed event sequence (i.e., second complement of iteration 3) into 4 sub-sequences and test all complements. All passed. Terminate the reduction process since 1-minimal sub-sequence is tested. The latest test which made the app fail is ΔFSoE (E3⇢E4⇢E7⇢E8).

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 *Tf*) (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. 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 *Tf* 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.

Example 4. In this example, we demonstrate how Algorithm 1 handles non-adjacent failure-inducing events. Assume that we have an original sequence of events E1 ⇢ E2 ⇢ E3 ⇢ E4 and the smallest sub-sequence that triggers a bug is E1 ⇢ E4 (e.g. E1 changes the state in a way where an exception is raised only when E4 executes). In the first iteration, the algorithm starts with granularity 2 and we have two sub-sequences (i.e., E3 ⇢ E4, and E1 ⇢ E2). None of them makes the app crash with the same stack trace. The algorithm then starts using the `Complement` and divides into smaller sub-sequences with the granularity 4 (i.e., one event in each sub-sequence) (i.e., Line 23 in Algorithm 1). In the second iteration, we test the complements of each sub-sequence (i.e., E2 ⇢ E3 ⇢ E4, E1 ⇢ E3 ⇢ E4, E1 ⇢ E2 ⇢ E4, and E1 ⇢ E2 ⇢ E3). Although all complements that include E1, E4 could make the app crash with the same stack trace, our algorithm takes the first failure (i.e., the second complement (E1 ⇢ E3 ⇢ E4)). It starts the next iteration with the granularity 3 and one event in each sub-sequence (Line 15 in Algorithm 1). In the third iteration, we operate the complement of each sub-sequence (i.e., E3 ⇢ E4, and E1 ⇢ E4) and the second complement (E1 ⇢ E4) makes the app crash. The algorithm starts the next iteration with the granularity 2 for the latest failed complement (E1 ⇢ E4) and one event in each sub-sequence (one sub-sequence includes E1, and another one includes E4.) (Line 15 in Algorithm 1). In the fourth iteration, none of them makes the app crash and the algorithm exits since it reaches 1-minimal (Line 21 in Algorithm 1). Therefore, the latest failure-inducing event sequence (i.e., E1 ⇢ E4 at the third iteration) is the simplified failure-inducing event sequence (i.e., Δ FSoE). In this way, the algorithm extracts the minimal failure-inducing events (i.e., E1, and E4) for non-adjacent failure-inducing events.

3.4. Backward dynamic slicing

This phase conducts the backward dynamic slicing for the reduced event sequence (ΔFSoE), which triggers a bug. Our dynamic slicing captures two levels of control- and data-dependence at both the program statement and event levels to leverage the event information from the inputs. Our dynamic slicing is done by producing a subgraph of the static PDG by considering only the control- and data-dependence of the executed statements and their related activities. The following describes the common notation:

$\xrightarrow{d}$ Data dependences.
$\xrightarrow{c}$ Control dependences.
S_{it} The instance of instruction S_i at time t.
`Ei` The event triggered while `i` represents the event's ID.

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_{2t} utilizes the same object o1 which is defined at Line 3 in S_{1t} and S_{2t} is data-dependent on S_{1t} at time t.

Example 5. To illustrate the data dependences 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 dependence 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_{2t} in E2 is data-dependent on S_{1t} in E1. But, only because object *o*2 used in S_{2t} (Line 5) depends on object *o*1 defined in S_{1t} (Line 3) at that *t* time.

Example 6. In our motivating example in Fig. 3, if `widthDecrementClick`, and `heightDecrementClick` are triggered before triggering `compressImageClick`, `compressImageClick` is data-dependent on both `widthDecrem- entClick`, 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_{4t} at Line 2 is executed upon only the evaluation result of S_{3t} at Line 1, S_{4t} is control-dependent on S_{3t}. To clarify, the value of `condition` (i.e., the predicate) at S_{3t} determines the execution of S_{4t}. In other words, if S_{3t} can alter the program's control and it determines whether S_{4t} 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 $\xrightarrow{c}$ 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 $\xrightarrow{c}$ E7);

```
class Act1 extends Activity{
onClick1 (...){ //E1
o1  =1;} //S_1t
onClick2 (...){ //E2
o2 = o1+2;} //S_2t}
```

$S_{2t} \xrightarrow{d} S_{1t}$
E2 $\xrightarrow{d}$ E1

Fig. 4. Data dependence.

```
if(condition){ //S_3t
System.out.println("True"); //S_4t}
```

$S_{4t} \xrightarrow{c} S_{3t}$

```
class Act3 extends Activity{
onCreate(...){ //E3
i = new Intent(this,Act4.class);
startActivity(i);}}
class Act4 extends Activity{
onCreate(...){}} //E4
```

E4 $\xrightarrow{c}$ E3

```
class Act7 extends Activity{
onCreate(...){
i = new Intent(this,Act8.class);
startActivity(i);}
onPause(...){}} //E7
class Act8 extends Activity{
onCreate(...){}} //E8
```

E8 $\xrightarrow{c}$ E7

Fig. 5. Control dependence.

Based on the control- and data dependence, ESDroid builds PDG. ESDroid then maps the executed statements in the simplified trace ΔFSoE 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;

- *Tf*: A list of executed statements when ΔFSoE is triggered on an instrumented APK.
- *idx*: An integer that is the location of the error instruction in *Tf* (i.e., the last index of *Tf* for the app crash because the last index holds the failure point, which is the point of interest).
- *Sl*: 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 *Tf*) including the list of statements executed while ΔFSoE is triggered, and (3) the index of *Tf* (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 *Sl*, affecting the point of interest until the app entry point. Note that the pre-conditions of the algorithm are (1) *Tf* 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., *Tf*) 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, `isCD` checks whether the execution of the current working node (i.e., the node located at the current index *idx* of *Tf*) (denoted as *Tf*[*idx*]) is determined by the previous node (denoted as *Tf*[*idx*-1]) for instruction-level control dependence (Lines 21–23). Particularly, `isCD` examines if the node located at *Tf*[*idx*-1] contains a predicate whose outcome controls the execution of the node located at *Tf*[*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, `getS` 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 FSoE. ESDroid uses Soot (Vallée-Rai et al., 2010) to conduct the instrumentation to produce our customized logging information. Regarding producing FSoE, there are several techniques to generate event sequences

Algorithm 2: Backward Dynamic Slicing. **Input:** an Apk *apk*, a list of statements *Tf*, the position *idx* which is the point of interest in *Tf*. **Output:** a list of statements *Sl*.

```
Sl ⟵ ø;
isSlice ⟵true;
while idx>=0 do
    for each Object o defined at Tf[idx] do
        if PDG_DD.contains(o) then
            isSlice ⟵true;
            break;
        end
    end
    if PDG_CD.contains(Tf[idx]) then
        isSlice ⟵ true ;
    end
    if isSlice and !Sl.contains(Tf[idx]) then
        Sl.add(Tf[idx]);
    end
    if isSlice then
        for each Object o used in Tf[idx] do
            PDG_DD.add(o);
        end
    end
    // check Tf[idx-1] contains a predicate whose
       outcome controls the execution of Tf[idx]
    if isSlice and isCD(Tf[idx],Tf[idx-1]) then
        PDG_CD.add(Tf[idx-1]);
    end
    if isSlice and Tf[idx]'s method m is callback then
        // add the last statement of callback which
           initiates m
        PDG_CD.add(getS(apk,m));
    end
    idx ⟵ idx-1;
    isSlice ⟵false;
end
return Sl;
```

to exercise Android apps. Monkey-style stress testing is considered the most robust and popular approach to exercise an app based on previous literature (Patel et al., 2018a; Choudhary et al., 2015; Zeng et al., 2016). For example, a prior study states that: "*researchers found that Monkey (the most widely used tool of this category in industrial settings) outperformed all of the research tools in the study*" (Choudhary et al., 2015). We choose to implement a Python program that generates random events using MonkeyRunner. Our program loads the main activity at the beginning. We did not adopt other similar techniques for generating events, including Android's built-in Monkey (Patel et al., 2018b) because log messages originally generated by Monkey were not easily translatable back to the corresponding *adb* command. We did not use RERAN (Gomez et al., 2013) because it generates events from hexadecimal to decimal based on the information obtained from *adb* `getevent`, and cannot reliably reproduce the same sequence of events, especially when the devices' resolutions are different.

Simplifying FSoE. To simplify FSoE, we have implemented a standalone tool written in Java. Our implementation uses the `Runtime.exec(String command)` method to execute the Python script with `MonkeyRunner` to conduct the testing on Android's emulator. To compare the testing result of ΔFSoE with that of the original FSoE, the testing result includes the exception type information, line number, method name, and class name produced by *adb* `Logcat`.[5]

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 ΔFSoE.

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 (27–17 654 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`, `Ringdroid`, `SyncMyPix`, `Tippy`, `WhoHasMyStuff` and `Yahtzee`) from the previous evaluation of SimplyDroid (Jiang et al., 2017).
- two apps (i.e., `APV PDF Viewer`, `NPR News`) from the previous evaluation of AndroidSlicer (Alavi et al., 2019).
- four apps (i.e., `Fdroid`, `AnyMemo`, `GnuCash` and `Transistor`) from Droixbench (Tan et al., 2018).

5 https://developer.android.com/studio/command-line/logcat.

Table 4
Information of buggy apps and exceptions for RQ1, RQ2, RQ3, and RQ4.

App	Dex code size (KB)	# of activities	Program version	Exception type	Dataset
Addi	656.9	4	1.98	ActivityNotFoundException	Su et al. (2020)
Anymemo	8887.5	27	10.9.922	NullPointerException	Su et al. (2020), and Tan et al. (2018)
APV PDF Viewer	63.1	3	0.2.6	NullPointerException	Alavi et al. (2019)
Bankdroid	5199.7	12	1.9.10.6	IllegalArgumentException	Su et al. (2020)
Birthdroid	431.1	3	0.6.3	NumberFormatException	Su et al. (2020)
Bites	49.9	5	1.3	NumberFormatException	Su et al. (2020)
Calculator	2149.3	1	1	NumberFormatException	GooglePlay (2021)
CampFahrplan	3223.7	7	1.32.2	IllegalArgumentException	Su et al. (2020)
Carnet - Notes app	5053	22	0.24.1	NullPointerException	GooglePlay (2021)
Cowsay	18.7	1	1.3	CalledFromWrongThreadException	Su et al. (2020)
DalvikExplorer	521.6	16	3.4	NullPointerException	Jiang et al. (2017)
Fdroid	5860.0	10	0.98	SQLiteException	Tan et al. (2018)
FishBun Demo	3293	7	0.6.2	NullPointerException	GooglePlay (2021)
fooCam	514.9	1	2.0	NullPointerException, SecurityException	Liu et al. (2016)
Geometric Weather	4393	14	2.113	ActivityNotFoundException	GooglePlay (2021)
GnuCash	7948.0	20	2.1.4	IllegalArgumentException	Tan et al. (2018)
LibreNews	3637.7	2	1.4	ArrayIndexOutOfBoundsException	Su et al. (2020)
Linux Deploy	2156	8	2.6.0	IllegalArgumentException	GooglePlay (2021)
Man Man	3562	2	2.1.0	ActivityNotFoundException	GooglePlay (2021)
Mitzuli	3329.7	2	1.0.7	BadTokenException	Su et al. (2020)
NPR News	17 654.3	14	2.4	NullPointerException	Alavi et al. (2019)
OBSSD - OBS Stream Deck	4085	4	1.2.2	IllegalArgumentException	GooglePlay (2021)
Official Cambridge Guide to IELTS	42 848	2	11.3.0.0	IllegalStateException	GooglePlay (2021)
Olam	715.4	1	1.0	SQLiteException, StringIndexOutOfBoundsException	Alavi et al. (2019), and Su et al. (2020)
PasswordMaker	331.68	3	1.1.11	NumberFormatException	Su et al. (2020)
Ringdroid	607.0	4	2.6	IllegalStateException	Jiang et al. (2017)
Scale Image View Demo	4277	1	4.0	ActivityNotFoundException	GooglePlay (2021)
Scribbler	20.4	3	0.1.8	IllegalFormatConversionException	Su et al. (2020)
SyncMyPic	231.0	8	0.15	NoClassDefFoundError	Jiang et al. (2017)
Tailscale	2008	1	1.8.3	ActivityNotFoundException	GooglePlay (2021)
Tickmate	591.9	6	1.2.0	CursorIndexOutOfBoundsException	Su et al. (2020)
Tippy	88.0	6	1.1.3	ArithmeticException	Jiang et al. (2017)
Transistor	2993.2	3	1.2.3	RuntimeException	Su et al. (2020), and Tan et al. (2018)
TripSit	2311.7	8	1.0	RuntimeException	Su et al. (2020)
Vanilla Music	1408	13	1.1.0	CursorIndexOutOfBoundsException, ResourcesNotFoundException	GooglePlay (2021)
WeightChart	541.6	6	1.0.4	ActivityNotFoundException	Jiang et al. (2017)
WhoHasMyStuff	47.3	4	1.0.7	NullPointerException	Jiang et al. (2017)
Yahtzee	27.4	2	1.1	NumberFormatException	Jiang et al. (2017)

- one app (i.e., `fooCam`) from RelFix (Liu et al., 2016).
- 13 apps (i.e., `Addi`, `Bankdroid`, `Birthdroid`, `Bites`, `CampFahrplan`, `Cowsay`, `LibreNews`, `Mitzuli`, `Passw- ordMaker`, `Olam`, `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`, `FishBun Demo`, `Geometric Weather`, `Linux Deploy`, `Man Man`, `OBSSD - 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 C3), and (2) instrumentation failed because Soot fails to parse the apk (i.e., Dex file overflow error for Android API 22).[6] 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.

5.1.2. Methodology

We ran the event sequence generation for 10 runs to produce FSoE. Each run was terminated after all random events (5000 events) had been triggered, or when a crash occurred. We conducted our evaluation to answer the following research questions.

6 https://github.com/secure-software-engineering/FlowDroid/issues/61.

RQ1: What is the effectiveness of ESDroid in reducing the size of the input event sequence?

RQ2: Which of our key two phases (i.e., Phase 3 = Simplifying FSoE (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 method calls: Number of method calls invoked in *Seq*

# of instructions: Number of Jimple instructions executed in *Seq*

Table 5 shows the comparison in size between the originally generated event sequence Seq_{orig} and the minimized event sequence $Seq_{ESDroid}$. Meanwhile, the second and the third column under the title "# of events" denote the number of events triggered in Seq_{orig} and $Seq_{ESDroid}$, respectively. The two columns under the title "# of callbacks" represent the number of callback methods invoked in Seq_{orig} and $Seq_{ESDroid}$, respectively. The two "# of method calls" columns denote the number of methods invoked in Seq_{orig} and $Seq_{ESDroid}$. (note that "# method calls" counts all method calls, including all callback methods). The two "# of Instructions" columns denote the number of instructions executed in Seq_{orig} and $Seq_{ESDroid}$. 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 proportioned 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 is 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, `Olam` has the second lowest reduction rate. `Olam` 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 `Olam` 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, `WhoHasMyStuf` and `GnuCash` have the longest processing time for the reduction in the experiment. Specifically, for `WhoHasMyStuf`, 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.

Table 5
Comparison of the number (#) and the reduction ratio (%) between the original event sequence and the event sequence minimized by ESDroid to trigger the same exception.

Apps	# of events			# of callbacks			# of method calls			# of instructions			Duration
	Original	ESDroid	(%)	Original	ESDroid	(%)	Original	ESDroid	(%)	Original	ESDroid	(%)	(s)
Addi	19	3	84	58	15	74	4 087	3 754	8	136 938	79 722	42	727.62
Anymemo	22	5	77	85	58	31	6 602	5 695	13	464 816	428 269	8	5 033.99
APV PDF Viewer	4	2	50	5	2	60	57	21	63	948	347	63	21.57
Bankdroid	236	23	90	23	18	21	45 348	30 926	31	155 151	102 113	34	9 276.45
Birthdroid	1097	14	98	21	21	0	222	220	1	905	893	1	11 462.30
Bites	471	8	98	55	14	74	231	58	74	2 109	389	82	1 439.24
Calculator	59	1	98	43	2	95	117	7	94	34 974	8 983	74	295.06
CampFahrplan	333	7	97	220	139	36	50 618	16 188	68	1 250 075	268 769	78	3 209.03
Carnet - Notes app	58	6	89	237	218	8	4 809	3 149	34	398 939	223 098	44	1 755.83
Cowsay	65	1	98	64	10	84	244	112	54	4 345	1 610	63	366.16
DalvikExplorer	47	6	87	8	3	62	468	159	66	8 502	1 011	88	148.53
Fdroid	91	3	96	1095	1065	3	192 583	125 202	34	2 678 033	1 878 576	30	664.77
FishBun Demo	93	3	96	18	7	61	83	20	75	86 049	19 959	77	1 025.91
fooCam[a]	3	1	66	199	114	42	106	50	52	811	405	50	77.23
fooCam[b]	148	3	95	153	68	55	131	76	41	909	500	45	563.39
Geometric Weather	66	5	92	197	191	3	48 943	22 144	54	585 048	332 065	43	2 910.27
GnuCash	35	10	71	15	13	13	287	252	12	936	922	1	13 439.05
LibreNews	66	7	89	30	10	66	101 831	37 414	63	636 956	232 652	63	1 644.62
Linux Deploy	105	6	94	128	101	21	3 328	2 657	20	1 294 897	541 846	58	2 743.71
Man Man	62	3	95	105	52	50	2 203	355	83	1 176 582	121 901	90	363.39
Mitzuli	146	5	96	490	167	65	248 861	167 884	32	1 230 777	905 576	26	1 251.34
NPR News	37	2	94	293	38	87	1 798	605	66	25 597	10 107	61	271.44
OBSSD - OBS Stream Deck	94	17	81	55	34	38	851	383	54	4 754 980	1 200 819	75	2 034.56
Official Cambridge Guide to IELTS	41	10	75	188	177	5	2 018	1 991	1	537 874	355 432	34	12 278.07
Olam[c]	9	4	55	4	4	0	185	185	0	26 597	26 597	0	317.31
Olam[d]	5	2	60	4	4	0	61	61	0	42 324	42 324	0	120.50
PasswordMaker	26	10	76	21	10	52	896	387	56	30 969	18 225	41	3 016.59
Ringdroid	135	4	97	45	8	82	4 647	188	95	26 263	2 500	90	1 963.55
Scale Image View Demo	238	5	97	203	61	69	2 902	352	87	295 088	52 468	82	464.63
Scribbler	22	2	90	7	4	42	27	19	29	132	83	37	422.98
SyncMyPic	14	1	92	13	6	53	71	53	25	356	275	23	419.29
Tailscale	24	2	91	25	24	4	126	104	17	174 283	120 950	31	231.67
Tickmate	52	3	94	31	12	61	935	857	8	3 491	3 093	11	737.23
Tippy	76	12	84	32	17	46	515	330	35	3 775	2 406	36	12 485.64
Transistor	638	2	99	15	10	33	154	118	23	1 625	1 242	24	1 263.84
TripSit	14	1	92	7	4	42	157	107	32	1 448	1 313	9	178.38
Vanilla Music[e]	11	2	80	459	453	1	6 258	6 249	1	92 527	83 118	10	789.18
Vanilla Music[f]	21	4	80	491	468	4	14 461	7 360	49	153 206	93 073	39	3 776.37
WeightChart	34	3	91	66	12	81	475	115	75	17 432	1 711	90	822.08
WhoHasMyStuf	1025	26	97	139	8	94	1 749	107	93	10 820	604	94	19 399.56
Yahtzee	131	6	83	2	2	0	5	5	0	33	33	0	18 093.75
Mean	143	6	87	130	89	42	18 279	10 632	42	398 720	174 780	45	3354

[a]The app refers to the defect that throws `NullPointerException`.
[b]The app refers to the defect that throws `SecurityException`.
[c]The app refers to the defect that throws `SQLiteException`.
[d]The app refers to the defect that throws `StringIndexOutOfBoundsException`.
[e]The app refers to the defect that throws `CursorIndexOutOfBoundsException`.
[f]The app refers to the defect that throws `ResourcesNotFoundException`.

In Table 6, the second, the third, and the fifth column under the title "# of instructions" denote the number of instructions executed in phase 2 (i.e., Producing Failure-inducing Sequence of Events (FSoE)), phase 3 (i.e., Simplifying FSoE (Segment-based Delta Debugging)), and phase 4 (i.e., Backward dynamic slicing) respectively. The fourth and sixth columns describe the reduction rate calculated on the count of instructions executed in phase 2. For instance, for `APV PDF Viewer`, 63% of trace was lessened in phase 3 compared to trace in Phase 2, while 95% was decreased in phase 4 as opposed to tracing in phase 2. The reduction for phase 2 is 0%–94% with an average reduction of 45%, whereas phase 3 is 36%–99% with an average reduction of 94%.

To evaluate our phases, we use the following two metrics.

$$\textit{Reduction rate in Phase 3} = \frac{\textit{\# of instructions executed in Phase 2} - \textit{\# of instructions executed in Phase 3}}{\textit{\# of instructions executed in Phase 2}} \tag{1}$$

$$\textit{Reduction rate in Phase 4} = \frac{\textit{\# of instructions executed in Phase 2} - \textit{\# of instructions executed in Phase 4}}{\textit{\# of instructions executed in Phase 2}} \tag{2}$$

The rows of 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 Dem`, `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 ESDroid 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.

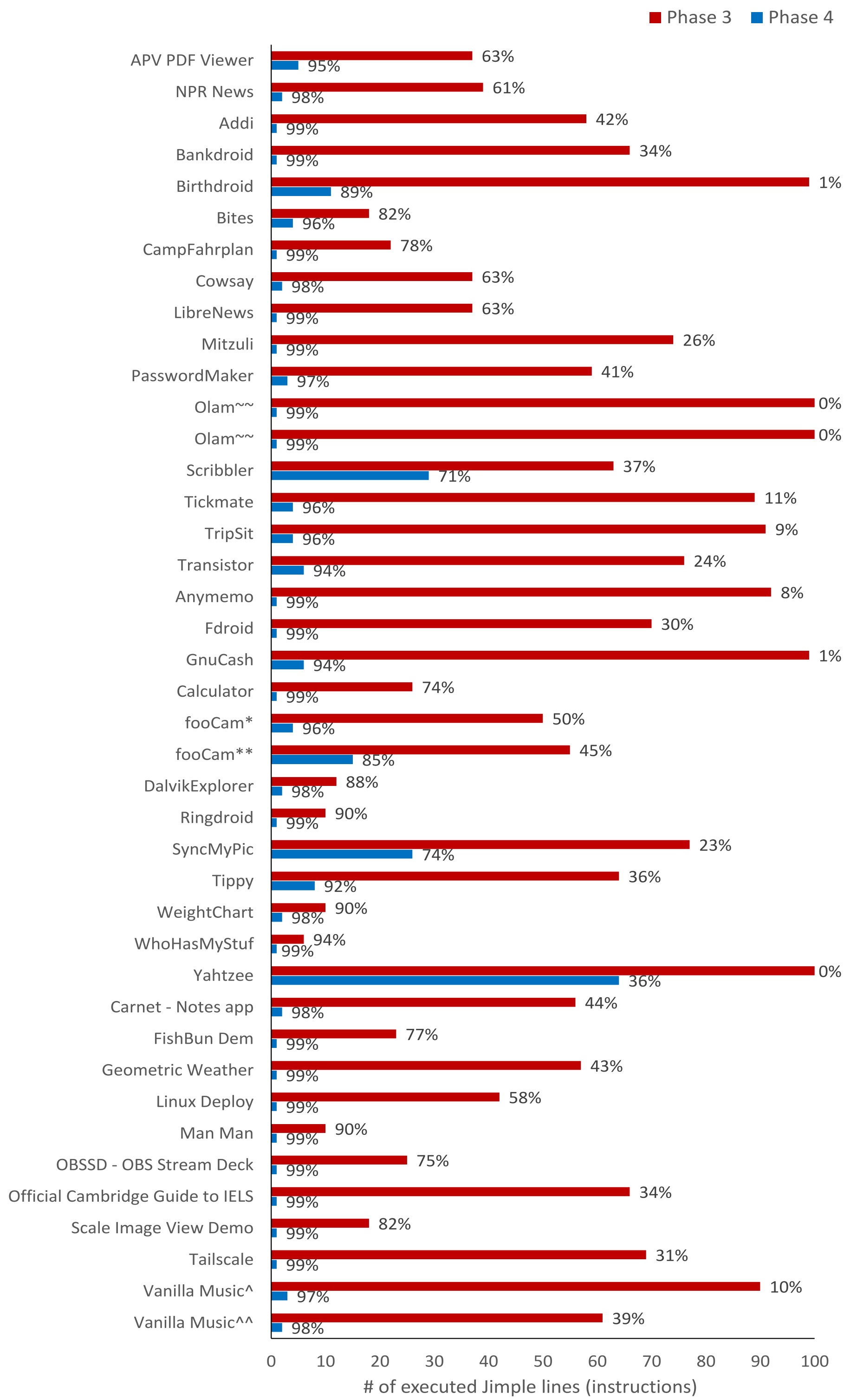


Fig. 6. The reduction rate (in percentage) for the number of instructions executed in Phase 3 and Phase 4. The app marked by * refers to the defect that throws `NullPointerException`, the app marked by ** refers to the defect that throws `SecurityException`, the app marked by ~ refers to the defect that throws `SQLiteException`, the app marked by ~~ refers to the defect that throws `StringIndexOutOfBoundsException`, the app marked by ^ refers to the defect that throws `CursorIndexOutOfBoundsException`, and the app marked by ^^ refers to the defect that throws `ResourcesNotFoundException`.

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),

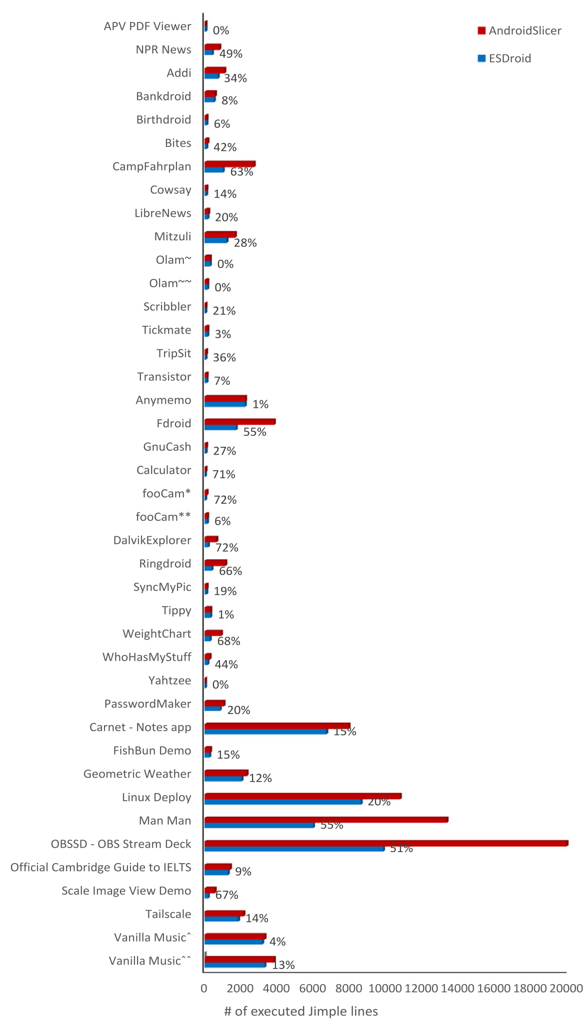
Table 6
Output comparison (#) between three phases in ESDroid (i.e., Phase 2 = Producing Failure-inducing Sequence of Events (FSoE), Phase 3 = Simplifying FSoE (Segment-based Delta Debugging), Phase 4 = Backward dynamic slicing). The values in the fourth column with the title (i.e., (%) (1)) and the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

Apps	# of instructions				
	Phase 2	Phase 3	(%) (1)	Phase 4	(%) (2)
Addi	136 938	79 722	42	721	99
Anymemo	464 816	428 269	8	2222	99
APV PDF Viewer	948	347	63	49	95
Bankdroid	155 151	102 113	34	522	99
Birthdroid	905	893	1	100	89
Bites	2 109	389	82	91	96
Calculator	34 974	8 983	74	15	99
CampFahrplan	1 250 075	268 769	78	1010	99
Carnet - Notes app	398 939	223 098	44	6697	98
Cowsay	4 345	1 610	63	86	98
DalvikExplorer	8 502	1 011	88	184	98
Fdroid	2 678 033	1 878 576	30	1731	99
FishBun Dem	86 049	19 959	77	262	99
fooCam[a]	811	405	50	31	96
fooCam[b]	909	500	45	132	85
Geometric Weather	585 048	332 065	43	2036	99
GnuCash	936	922	1	60	94
LibreNews	636 956	232 652	63	163	99
Linux Deploy	1 294 897	541 846	58	8591	99
Man Man	1 176 582	121 901	90	5980	99
Mitzuli	1 230 777	905 576	26	1203	99
NPR News	25 597	10 107	61	412	98
OBSSD - OBS Stream Deck	4 754 980	1 200 819	75	9821	99
Official Cambridge Guide to IELS	537 874	355 432	34	1275	99
Olam[c]	26 597	26 597	0	295	99
Olam[d]	42 324	42 324	0	148	99
PasswordMaker	30 969	18 225	41	839	97
Ringdroid	26 263	2 500	90	390	99
Scale Image View Demo	295 088	52 468	82	182	99
Scribbler	132	83	37	38	71
SyncMyPic	356	275	23	91	74
Tailscale	174 283	120 950	31	1842	99
Tickmate	3 491	3 093	11	152	96
Tippy	3 775	2 406	36	317	92
Transistor	1 625	1 242	24	104	94
TripSit	1 448	1 313	9	51	96
Vanilla Music[e]	92 527	83 118	10	3171	97
WeightChart	17 432	1 711	90	292	98
WhoHasMyStuff	10 820	604	94	159	99
Yahtzee	33	33	0	21	36
Vanilla Music[f]	153 206	93 073	39	3300	98
Mean	392 780	174 779	45	1336	94

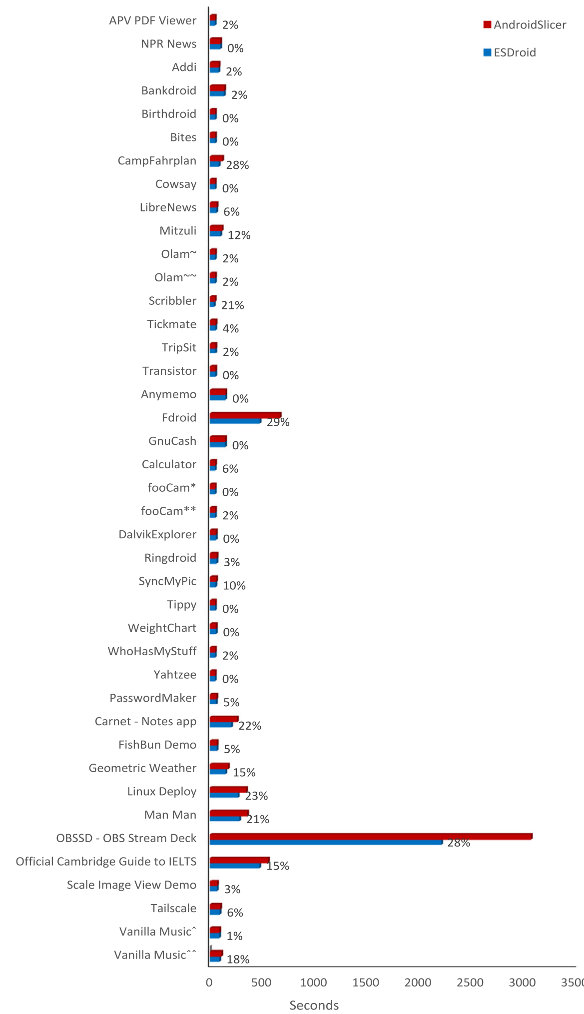
[a]The app refers to the defect that throws `NullPointerException`.
[b]The app refers to the defect that throws `SecurityException`.
[c]The app refers to the defect that throws `SQLiteException`.
[d]The app refers to the defect that throws `StringIndexOutOfBoundsException`.
[e]The app refers to the defect that throws `CursorIndexOutOfBoundsException`.
[f]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 `Yahtzee`. 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 $\hat{A}12$ 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 $\alpha = 0.05$. 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 $\hat{A}12$ measure reports a medium effect size $\hat{A}12 = 0.56$.

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



(a) # of executed Jimple lines (i.e., slice).



(b) The time taken (in seconds).

Fig. 7. The reduction rate (in percentage) for the size of the slices and the time taken in generating the dynamic slice by ESDroid compared with AndroidSlicer. The app marked by * refers to the defect that throws `NullPointerException`, the app marked by ** refers to the defect that throws `SecurityException`, the app marked by ~ refers to the defect that throws `SQLiteException`, the app marked by ~~ refers to the defect that throws `StringIndexOutOfBoundsException`, the app marked by ^ refers to the defect that throws `CursorIndexOutOfBoundsException`, and the app marked by ^^ refers to the defect that throws `ResourcesNotFoundException`.

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 AndroidSlicer

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

Table 7
Information of buggy apps and exceptions for RQ5.

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

Table 8
Comparison of the number (#) of Jimple instructions (JS) on the slice between Mandoline and ESDroid.

Apps	#JS			
	Mandoline	Mandoline++	ESDroid	(%)
Anki	NoSuchElementException	3	3	0
Birthdroid	NullPointerException	14	7	50
Fastadapter	NoSuchElementException	85	85	0
Fdroid	NoSuchElementException	447	280	37
Gnucash	NoSuchElementException	270	221	18
SiliCompressor	39	39	26	33
K9	NoSuchElementException	120	87	25
Micromath	NoSuchElementException	263	263	0
Newsblur	NoSuchElementException	138	138	0
Specialdates	NoSuchElementException	404	361	10
Mean	–	175	145	18

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

To compare the effectiveness of our approach versus Mandoline, we additionally evaluated our approach against Mandoline for 10 apps (9 apps used in the original experiments in Mandoline Ahmed et al., 2021, and the motivating example). We exclude one of Mandoline defects (i.e., `Habdroid`) because we cannot reliably reproduce the exception in the app after running the test generation 10 times with different seed values (does not satisfy C3). 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 unfeasible 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 $\hat{A}12$ measure reports a medium effect size $\hat{A}12 = 0.55$.

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 (Arzt et al., 2014) so it inherits the current limitations of these approaches. For example, ESDroid does not support debugging for multi-threading in Android apps due to the lack of sound support in 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, we modified Mandoline (in Mandoline++) to address the Mandoline implementation issue. The modification could have introduced defects. We mitigate this threat by making minimal modifications to Mandoline. Furthermore, we manually evaluate the quality of the generated program slices to ensure that our generated reduced slices include the program statement triggering the crash. As our delta-debugging step uses stack trace information to simplify the failure-inducing sequence of events, our reduced slice is guaranteed to include the statement triggering the crash by construction. Hence, the manual analysis is a relatively straightforward check. In addition, our focus is not to tune different delta-debugging strategies but to make dynamic slicing input-aware. The basic strategy of delta-debugging is already good enough and exercising more strategies (e.g., hierarchical delta-debugging) is an interesting future topic.

External validity: Since finding the exception is the prerequisite for dynamic slicing, we believe that one challenge lies in finding the exception in the first place for an evaluated app. Moreover, 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 (Misherghi and Su, 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 used 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) (Parnin 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

Weiser, M., 1984. Program slicing. IEEE Trans. Softw. Eng. (4), 352–357.
Agrawal, H., Horgan, J.R., 1990. Dynamic program slicing. ACM SIGPlan Not. 25 (6), 246–256.

Horwitz, S., Reps, T., Binkley, D., 1988. Interprocedural slicing using dependence graphs. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. pp. 35–46.
Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) 9 (3), 319–349.
Agrawal, H., DeMillo, R.A., Spafford, E.H., 1991. Dynamic slicing in the presence of unconstrained pointers. In: Proceedings of the Symposium on Testing, Analysis, and Verification. pp. 60–73.
Korel, B., Laski, J., 1988. Dynamic program slicing. Inform. Process. Lett. 29 (3), 155–163.
Wang, T., Roychoudhury, A., 2008. Dynamic slicing on Java bytecode traces. ACM Trans. Program. Lang. Syst. 30 (2), 1–49.
Alves, E., Gligoric, M., Jagannath, V., d'Amorim, M., 2011. Fault-localization using dynamic slicing and change impact analysis. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, pp. 520–523.
Gupta, R., Harrold, M.J., Soffa, M.L., 1992. An approach to regression testing using slicing. In: ICSM, Vol. 92. Citeseer, pp. 299–308.
Agrawal, H., Horgan, J.R., London, S., Wong, W.E., 1995. Fault localization using execution slices and dataflow tests. In: Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95. IEEE, pp. 143–151.
Alavi, A., Neamtiu, I., Gupta, R., 2019. Dynamic slicing for android. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 1154–1164.
Ahmed, K., Lis, M., Rubin, J., 2021. Mandoline: Dynamic slicing of android applications with trace-based alias analysis. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 105–115.
Zeller, A., Hildebrandt, R., 2002. Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. 28 (2), 183–200.
Gupta, N., He, H., Zhang, X., Gupta, R., 2005. Locating faulty code using failure-inducing chops. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 263–272.
Clapp, L., Bastani, O., Anand, S., Aiken, A., 2016. Minimizing GUI event traces. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 422–434.
Jiang, B., Wu, Y., Li, T., Chan, W.K., 2017. SimplyDroid: Efficient event sequence simplification for android application. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 297–307.
Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V., 2010. Soot: A Java bytecode optimization framework. In: CASCON First Decade High Impact Papers. pp. 214–224.
Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Not. 49 (6), 259–269.
Kiss, A., 2020. Generalizing the split factor of the minimizing delta debugging algorithm. IEEE Access 8, 219837–219846.
Patel, P., Srinivasan, G., Rahaman, S., Neamtiu, I., 2018a. On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey. In: Proceedings of the 13th International Workshop on Automation of Software Test. pp. 34–37.
Choudhary, S.R., Gorla, A., Orso, A., 2015. Automated test input generation for android: Are we there yet?(e). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 429–440.
Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W., Xie, T., 2016. Automated test input generation for android: Are we really there yet in an industrial case? In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 987–992.
Patel, P., Srinivasan, G., Rahaman, S., Neamtiu, I., 2018b. On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey. In: Proceedings of the 13th International Workshop on Automation of Software Test. ACM, pp. 34–37.
Gomez, L., Neamtiu, I., Azim, T., Millstein, T., 2013. Reran: Timing-and touch-sensitive record and replay for android. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, pp. 72–81.
Machado, P., Campos, J., Abreu, R., 2013. MZoltar: automatic debugging of Android applications. In: Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile. ACM, pp. 9–16.
Su, T., Fan, L., Chen, S., Liu, Y., Xu, L., Pu, G., Su, Z., 2020. Why my app crashes understanding and benchmarking framework-specific exceptions of android apps. IEEE Trans. Softw. Eng..
Tan, S.H., Dong, Z., Gao, X., Roychoudhury, A., 2018. Repairing crashes in android apps. In: Proceedings of the 40th International Conference on Software Engineering. pp. 187–198.
2021. [link]. URL https://play.google.com/store/apps.
Liu, J., Wu, T., Yan, J., Zhang, J., 2016. Fixing resource leaks in Android apps with light-weight static analysis and low-overhead instrumentation. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 342–352.
Conover, W.J., 1999. Practical Nonparametric Statistics, Vol. 350. john wiley & sons.
Vargha, A., Delaney, H.D., 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. 25 (2), 101–132.
Yu, K., Lin, M., Chen, J., Zhang, X., 2012. Practical isolation of failure-inducing changes for debugging regression faults. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 20–29.
Misherghi, G., Su, Z., 2006. HDD: hierarchical delta debugging. In: Proceedings of the 28th International Conference on Software Engineering. pp. 142–151.
Hammoudi, M., Burg, B., Bae, G., Rothermel, G., 2015. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 333–344.
Zhou, X., Peng, X., Xie, T., Sun, J., Li, W., Ji, C., Ding, D., 2018. Delta debugging microservice systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 802–807.
Maras, J., Carlson, J., Crnković, I., 2011. Client-side web application slicing. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, pp. 504–507.
Tonella, P., Ricca, F., 2005. Web application slicing in presence of dynamic code generation. Autom. Softw. Eng. 12 (2), 259–288.
Parnin, C., Orso, A., 2011. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 199–209.
Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 273–282.
Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.
Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 609–620.
Li, X., Li, W., Zhang, Y., Zhang, L., 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 169–180.
Hoffmann, J., Ussath, M., Holz, T., Spreitzenbarth, M., 2013. Slicing droids: program slicing for smali code. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. pp. 1844–1851.
Dilhara, M., Cai, H., Jenkins, J., 2018. Automated detection and repair of incompatible uses of runtime permissions in Android apps. In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems. ACM, pp. 67–71.
Kong, P., Li, L., Gao, J., Bissyandé, T.F., Klein, J., 2019. Mining Android crash fixes in the absence of issue-and change-tracking systems. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 78–89.
Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A., Scott, A., 2019. Sapfix: Automated end-to-end repair at scale. In: Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice. IEEE Press, pp. 269–278.
Xu, T., 2019. Improving automated program repair with retrospective fault localization. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, pp. 159–161.

Hsu Myat Win is currently a Ph.D. student enrolled in the SUSTech-UTS Joint-PhD program. She is supervised by Dr. Yulei Sui 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.C.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 UIUC. 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, FSE, ASE) and reviewers for several journals (TOSEM, TSE and EMSE). She also coorganized 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).