C02029: Doctor of Philosophy
CRICOS Code: 009469A
33874 PhD Thesis: Software Engineering
March 2023

Automated Analysis and Debugging of Android Applications

Hsu Myat Win

School of Computer Science
Faculty of Engineering and IT
University of Technology Sydney
NSW - 2007, Australia

Automated Analysis and Debugging of Android Applications

A thesis submitted in fulfilment of the requirements
for the degree of

Doctor of Philosophy
in
Software Engineering

by

Hsu Myat Win

to

School of Computer Science
Faculty of Engineering and Information Technology
University of Technology Sydney
NSW - 2007, Australia

March 2023

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, Hsu Myat Win, declare that this thesis is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the School of Computer Science, Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Production Note:
Signature removed prior to publication.

SIGNATURE: _____

[Hsu Myat Win]

DATE: 27th March, 2023

ABSTRACT

Among mobile platforms, Android stands out as the most commonly used platform. It is an open-source development platform, and many stakeholders, such as researchers and developers, can contribute their ideas to enhance work processes. Since anyone can access the code and fix a bug for continuous improvement, new versions or features are added to the software occasionally. As a result, the rapid change of SDK versions makes the developers update the apps frequently. Regardless of the code inspection process and extensive testing before releasing a new app version, the chance of introducing bugs is still high in Android apps. Moreover, unlike standard Java, Android has several entry points and an event-driven feature that allows foreground and background concurrently. Therefore, automated analysis, such as debugging, for Android apps is more difficult than standard Java. This dissertation tries to answer the question: How can we help developers to improve the debugging more effectively for Android apps?

We answer this question with two contributions geared towards minimizing the search space and expediting the debugging process. Firstly, to mitigate the expanded search space resulting from the lack of awareness regarding asynchronous events and lifecycle events in debugging Android apps, we introduce ESDroid, an Event-aware dynamic Slicing technique for AnDroid apps, an approach that combines segment-based delta debugging with dynamic backward slicing. This technique effectively narrows down the search space, providing a precise slice, and enhancing bug identification and resolution. Secondly, we propose a solution called SfR (Slicing for Resources) to address the limitations of existing static and dynamic slicing techniques for Android apps when

faults are located in application resources such as layout definitions and user interface strings. SfR identifies the dependencies between program statements and application resources, enabling a comprehensive slice that encompasses these crucial components of Android apps. By addressing the question of how to help developers improve debugging more effectively for Android apps, we contribute two complementary works to advancing the state-of-art in Android app development and debugging practices.

Under the umbrella of automated analysis of Android applications, we extend our work to encompass the automated analysis of unethical behavior in open-source software (OSS) projects, including Android projects. Inspired by various stakeholders in open-source software (OSS) projects, our third contribution focuses on proposing Ethic detector (Etor), an innovative approach that leverages automated analysis techniques to detect unethical behavior within the projects. By developing Etor, we aim to help developers enhance the development process, ensuring improved security, privacy, and code quality, ultimately establishing a foundation of user trust while making a meaningful contribution to the field of ethical software development and fostering responsible practices within the OSS community.

Dedication

In memory of my father

To my mother

With love and eternal appreciation

ACKNOWLEDGMENTS

I would like to express my deep gratitude to all those who have given me help and support during my four years at the University of Technology Sydney (UTS).

First, my heartiest thanks to my supervisors, Associate Professor Yulei Sui and Assistant Professor Shin Hwei Tan, for their helpful guidance, valuable suggestions and constant encouragement in my study. I learned from them everything I know of research in the field of program analysis. They have taught me how to think about research problems and helped me make significant progress in skills that are essential for a researcher. Their influence is present on every page of this thesis. I wish that I can use the skills I learned from them in my future research career.

I also thank my colleagues from "Fortnight-meeting program analysis group (UTS)" and "Southern University of Science and Technology (SUSTech)" for the inspiring discussions on various research topics.

Finally, my thanks would go to my dear siblings for their loving consideration and great confidence in me all through these years. I also sincerely thank my friends for their patience, which enlightens and removes my doubts when I feel sad and have doubts about life and the future.

List of Publications

Related to the Thesis :

1. Win, H. M. "Complement of dynamic slicing for Android applications with def-use analysis for application resources," *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. 2022. Status: Published

2. Win, H. M., Sui, Y., Tan, S. H. "Event-Aware Precise Dynamic Slicing for Automatic Debugging of Android Applications," *Journal of Systems and Software*. 2023. Status: Published

Others :

1. Win, H. M., Tan, S. H., Wang, H. "Towards Automated Detection of Unethical Behavior in Open-source Software Projects," *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023. Status: In progress

Table of Contents

LIST OF FIGURES

FIGURE **Page**

List of Tables

CHAPTER 1

INTRODUCTION

Generally, the software system's quality relies on the assurance that the program will perform satisfactorily in terms of its functional and nonfunctional specifications within the expected deployment environments. In a typical commercial software house, delivering this assurance via proper testing, debugging and verification activities can easily exceed 50 percent of the total development cost [76].

Among the activities, the global cost of debugging software (i.e., finding and fixing errors or bugs in the source code) has risen to $312 billion annually, according to a study from the Judge Business School of the University of Cambridge [1]. Moreover, on average, software developers spend up to 90 percent of their programming time finding and fixing bugs [27] because debugging is twice more challenging as writing the program in the first place [98], and it takes a long time, often more than creating it [185]. Therefore, finding and fixing bugs faster, especially more predictable and productive debugging, is consequently crucial for developers and organizations.

When debugging, developers have to find a way to relate an observable failure to the causing defect in the source code. The distance from defect to failure may be long in space and time. Therefore, many researchers have proposed automated analysis and debugging

techniques [16] [78][116] [151] [153] [172] [174] [44] [30] [173] [93] [115] [146] attempt to (1) find the causes of a program failure with or without human interactions (2) reduce the time and effort and (3) improve the software reliability and quality.

Despite significant advancements, the widespread adoption of automated debugging in practice still requires further improvement [139]. This situation can be attributed to two primary reasons. Firstly, developers must possess a deep understanding of the software system and its environment in order to trace the infection chain back to its root cause [21]. Secondly, the debugging techniques employed should be tailored to the nature of the software system and its environment to ensure effectiveness.

For example, in mobile applications, the unique software systems have distinct features that require developers to adapt their debugging approaches accordingly. Specifically, unlike traditional desktop applications, mobile apps interact with various hardware components and utilize specific operating systems, such as Android. Consequently, developers need to understand the nature of Android apps and the specific tools required to effectively tackle the debugging challenges posed by this environment. By addressing these distinctive aspects, developers can enhance the overall efficiency and accuracy of the debugging process in the context of Android applications.

In recent years, the mobile industry has witnessed explosive growth, and the complexity of mobile applications has increased rapidly. In Quarter 1 of 2023, the Android OS has taken 71.95% of the smartphone market share [2]. There are more than 2.6 million Android applications in Google Play. To increase end-users satisfaction, mobile application developers must improve the quality of their applications, and mobile debugging is one important measure to achieve this goal.

Moreover, Android is open-source, allowing stakeholders such as researchers and developers to participate in enhancing applications; unfortunately, those features trigger updating software more often and introduce new bugs while enhancing in a short time

frame. When focusing on updating software to be compatible with SDK, the debugging process becomes more costly than software which is rarely enhanced.

In addition, the Android framework employs an asynchronous event, a common way to collect and process data regarding user events. To schedule and execute a user event, the framework supports the event queue mechanism. Adding an event to and dispatching an event from the queue are non-deterministic because of arbitrary user interactions. Moreover, each Android component has its own lifecycle and must follow the prescribed lifecycle paradigm, which defines how the component is created, operated and destroyed. Such an event-driven system and lifecycle nature make debugging more difficult than those in traditional Java programs.

Recently, researchers have introduced the techniques to enhance the debugging process for Android apps. AndroidSlicer [23], utilizing the slicing technique, introduced asynchronous callback constructions to handle control- and data-dependences by defining callbacks as nodes containing other nodes or a supernode. On the other hand, Mandoline [22] focused on data-dependences, providing tracking of data propagation through object fields using low-overhead instrumentation and claiming high slicing accuracy for Android applications. However, Mandoline lacked a comprehensive understanding of lifecycle stages and control-dependences among callbacks. Additionally, both AndroidSlicer and Mandoline overlooked the significance of considering the input, which involves a sequence of user events, for effective debugging.

Therefore, our research question is, “How can we help developers to improve the debugging more effectively for Android apps?”. We address this question with two contributions aimed at reducing the search space and expediting the debugging process of Android apps by incorporating event and lifecycle awareness in the debugging of Android apps and analyzing the data-flow to the resources in the apps.

1. Event-aware precise dynamic slicing for Android apps

2. Complement of Dynamic Slicing for Android Applications with Def-Use Analysis for Application Resources

Specifically, the first work introduces an event-aware precise dynamic slicing technique tailored specifically for Android apps. This approach narrows down the search space, providing developers with a more focused and precise debugging process. The second work complements the dynamic slicing technique by incorporating Def-Use analysis for application resources. This enhancement addresses the limitations of existing techniques when debugging faults within resources such as layout definitions and user interface strings, ensuring a comprehensive and thorough debugging process. By combining these two works, we strive to provide developers with advanced tools and methodologies to improve the efficiency and effectiveness of debugging Android apps, ultimately advancing the state-of-the-art in Android app development practices.

Under the umbrella of automated analysis of Android applications, we broaden our focus to encompass the automated analysis of unethical behavior in Android projects, including open-source software (OSS) projects as well. By extending our work, our aim is to assist developers in making better software or apps by checking for potential issues that could harm user security, privacy, or the app's quality. Therefore, as our third work, we contribute;

3. Towards automated detection of unethical behavior in OSS projects

Specifically, Etor utilizes automated analysis techniques to identify unethical behavior within OSS projects, including those related to Android. Our goal in developing Etor is to contribute to the field of ethical software development and foster responsible practices within the OSS community. This approach combines insights gained from automated analysis techniques with the collaborative efforts observed in OSS projects, enabling the detection of unethical behaviors in Android applications and other OSS projects.

1.1 Research Aim and Objectives

Among different mobile operating systems, Android is the most popular one [37], and software evolution is faster than other operating systems because of its open-source nature. Within the short time frame, the quality of Android apps is questionable and automated analysis for Android applications becomes in high demand. Therefore, to help developers speed up the automated analysis (i.e., automated debugging) process for Android apps, there are two following aims to achieve in our work;

1. To understand the role of asynchronous events that is significant for Android apps and enables the search space's size, and
2. To highlight the importance of considering the data flowing through application resources.

To fulfill the first aim, we developed the tool called ESDroid to reduce the search space via event awareness. To fulfill the second aim, we implemented the tool called SfR to complement the slice by providing the connection between statements and resources. Additionally, both tools have significantly improved the efficiency of automated debugging for Android apps.

1.2 Contributions

This dissertation makes the following two main contributions.

- 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).

- We propose a novel approach called SfR (Slicing for Resources), which identifies the dependences between the program statements and the application resources to complete the slice for Android applications. We performed the static analysis to generate the resource dependence graph (RDG), which includes data dependences on application resources. We integrated RDG in AndroidSlicer and evaluated on 3 Android applications. The result shows that SfR is more efficient in accuracy than the existing state-of-the-art dynamic slicing technique, AndroidSlicer.

Inspired by automated analysis in software engineering, we also present the following additional contribution.

- We present a study of the types of unethical behavior in OSS projects in GitHub. Our study of 320 GitHub issues provides a taxonomy of 15 types of unethical behavior guided by six ethical principles (e.g., attribution, autonomy). Examples of unethical behavior include *soft forking* (copying a repository without forking) and missing the license file for a repository. We also identify 18 types of software artifacts affected by the unethical behavior. The diverse types of unethical behavior identified in our study (1) call for attentions of developers and researchers when making contributions in GitHub, and (2) point to future research on automated detection of unethical behavior in OSS projects. Based on our study, we propose Etor, an approach that can automatically detect six types of unethical behavior discovered in our study. Etor uses ontological engineering and Semantic Web Rule Language (SWRL) rules to model GitHub attributes and software artifacts. Our

evaluation on 195,621 GitHub issues (1,765 GitHub repositories) shows that Etor can automatically detect 552 violations of unethical behavior with an average of 80.5% true positive rate. This shows the feasibility of automated detection of unethical behavior in OSS projects.

1.3 Thesis Outline

In Chapter 2, we introduce terminologies and the background of our work. Specifically, Section 2.1 illustrate the Android framework, the execution of Android apps and their lifecycle. Section 2.2 represents the delta-debugging technique. Section 2.3 describes the def-use analysis, while Section 2.4 is about program slicing.

Chapter 3 illustrates the literature review, including the related work and its limitations. Particularly, Section 3.1 discuss the related work of automated debugging in software engineering, while Section 3.2, and Section 3.2.1 describe the literature review of delta debugging. We discuss the related work of program slicing in Section 3.3, Section 3.3.1, Section 3.3.2, and Section 3.3.3.

In Chapter 4, we present ESDroid, an Event-Aware precise dynamic Slicing approach for Android apps, by introducing segment-based delta-debugging into backward dynamic slicing. We aim to improve the automated debugging process for Android applications, which is vital for the software maintenance and a topic in scientific research

In Chapter 5, we introduce a new technique called SfR (Slicing for Resources) to complete the slicing for Android apps by providing resource dependences between the program statements and the application resources. Again, we aim to improve the automated debugging process for Android applications.

In Chapter 6, inspired by the automated analysis, we present a new approach called Etor, an ontology-based approach that can automatically detect unethical behavior in OSS projects. We aim to improve the automated analysis for OSS projects and raise

awareness of the importance of understanding ethical issues in OSS projects.

We conclude this dissertation in Chapter 7 with a summary of the contributions and possible future work.

CHAPTER 2

BACKGROUND

In this section, we provide the relevant background information for this thesis. First, we briefly overview the Android platform, its applications and nature, and stress the necessity of automated software analysis tools for Android apps. Second, we discuss Android application analysis challenges and highlight the need for more efficient debugging tools for Android apps. Third, we briefly introduce the traditional methodologies (i.e., (1) delta debugging, (2) def-use analysis, and (3) program slicing) used in our approaches and present the motivation for choosing these methodologies.

2.1 Android

Android is a popular Linux-based smartphone operating system designed by Google and released as the Android Open Source Project (AOSP) in 2007. The software can be freely obtained from a central repository and modified in terms of the license, mostly BSD and Apache [3]. The openness and extensibility allow developers and researchers to modify the system, and the development of Android takes place quickly; therefore, a new major release happens every few months.

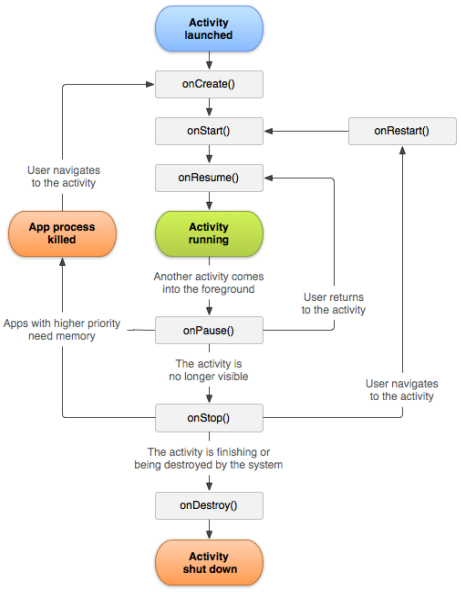


Figure 2.1: Activity Lifecycle.

Android applications (“apps”), written in Java and (optional) C/C++ native code. The Android software stack comprises a middleware component (i.e., Android Framework (AF)). It orchestrates control flow and mediates inter- and intra-app communication and communication between apps and the lower layers. Such oriented features make the mobile application more complex and the development of apps more difficult. In addition, applications are generally delivered within a very short time duration in comparison to desktop applications because the new major releases (major Android versions) come out every few months, and developers must continuously update their apps to be compatible with the versions. With a shorter development time for each update, the quality of apps becomes more difficult to guarantee.

The structure of an Android app is based on the following four basic components.

- **Activity** dictates the user interface, like a browser window or a settings page, and handles user interaction with the smartphone screen.

- **Service** handles background processing associated with an application. For example, the user can continue to interact with the activity while the service for

uploading data to a web resource runs because it executes in the background.

- **BroadcastReceiver** handles communication between Android OS and applications. For example, in the case of a low battery event, the app can be stopped from using any backend data polling mechanism because it listens to the low battery event registered.

- **ContentProvider** handles data and database management issues.

An application does not necessarily consist of all four components, but there must be at least one activity to present a graphical user interface.

2.1.1 Challenges of debugging for Android apps

In an Android app, each component follows its lifecycle callbacks (onXYZ() methods), which are called by the Android system to start/stop/resume the component following environment needs. For example, onCreate() is the initial method to set up an Activity, and onDestroy() is the counterpart to onCreate(). Figure 2.1 shows the lifecycle of an activity, and each method in the figure represents a lifecycle callback. Lifecycle transition also follows certain principles. For instance, an activity with the paused state (onPause()) could change to the resumed state (onResume()) or the stopped state (onStop()) or be killed by the Android system to free up RAM. Consequently, these features of Android apps hinder the soundness of some analysis scenarios because these lifecycle methods are not directly connected to the execution flow.

Unlike Java programs, Android apps do not have a single main method. Rather, the apps have multiple entry points (e.g., onCreate() and onResume() methods, which the Android framework calls at runtime). Therefore, an automated static analyzer must search for all entry points and build several call graphs without assurance on how these graphs may be connected.

Moreover, the Android app uses an event-driven model that dictates control flow via the event queue mechanism to schedule and execute a user event. An event can be a user action (e.g., touch), the arrival of sensor data (e.g., GPS), a lifecycle event (e.g., onPause() when the app is paused), and inter- or intra-app messages. All these traits, from externally orchestrated control flow to time-sensitive sensor input, render traditional automated analysis approaches, such as automated debugging, inapplicable to Android. In addition, 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 more complicated than traditional Java programs.

2.2 Delta Debugging

A typical problem in debugging is that, given an input, only a small part of that input may be responsible for the failure or the bug. To address this problem, the researchers introduced delta debugging to simplify failure-inducing circumstances and isolate failure causes automatically [184]. It is well-known for failure-inducing program input [186], failure-inducing changes [154], and failure-inducing user interaction [147] because no specific prerequisites are required, it is a robust algorithm, and easy to implement and use.

Delta debugging (DD) divides a test case that causes the program failure (We assume a test case is a set of changes.) into subsets. It then removes the subsets, which cause the failure disappears until it gets the smallest subset that causes failure. Initially, DD performs a binary search on the whole set of changes. Specifically, it first partitions the set of changes into two subsets and tests each of them individually: if any of the two subsets produces the failure, DD marks it as a minimal, failure-inducing test case. DD then recursively continues to search in it for a shorter one. The most straightforward reduction rule for DD is a classical `Divide and Conquer` approach.

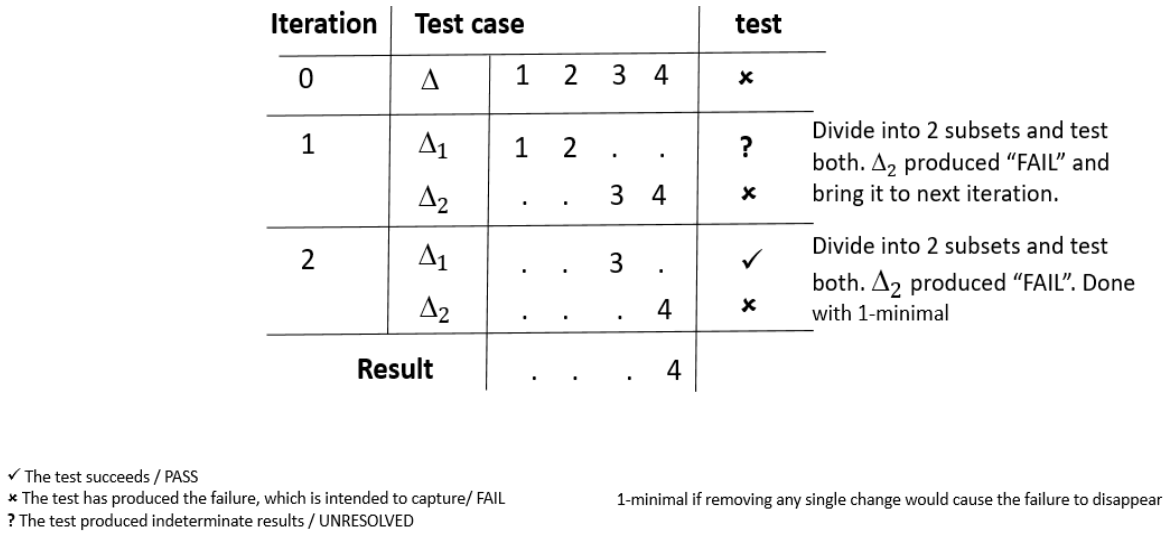


Figure 2.2: The `Divide and Conquer` strategy.

Figure 2.2 illustrates the process of the `Divide and Conquer` for a test case with four changes that can produce the bug. The first column describes the number of iterations. The second and third columns show each subset (Δ) with the changes included. The fourth column presents the outcome of each test. At Iteration 1, we divide the test case into two subsets (i.e., Δ_1 and Δ_2) and test each of them. We assume the second subset (i.e., Δ_2) makes the test fails. At Iteration 2, DD continues the search in the failing subset (i.e., Δ_2 at iteration 1) by partitioning it into two subsets and testing each. In this way, it iterates the same procedure to shrink the test case until 1-minimal. We call it 1-minimal if removing any single change would cause the failure to disappear. In our example, we got the smallest test case which causes failure after two iterations because each subset has one change. Note that the granularity for the `Divide and Conquer` is 2 for every iteration. **Granularity** means the number of subsets that the delta debugging divides the changes into.

In some cases, if none of the subsets produces the failure, DD increases the granularity by doubling it, and tests each complement (i.e., ∇). This reduction rule results in the `Complement` approach. The process is continued until each subset has a size of 1, and

no further reduction is possible. To calculate the granularity, we use two formulas; (I) $min(2n, size)$ if none of the subsets fails, and (II) $max(n-1, 2)$ if any subset fails. $size$ is the number of changes, and n is the number of subsets.

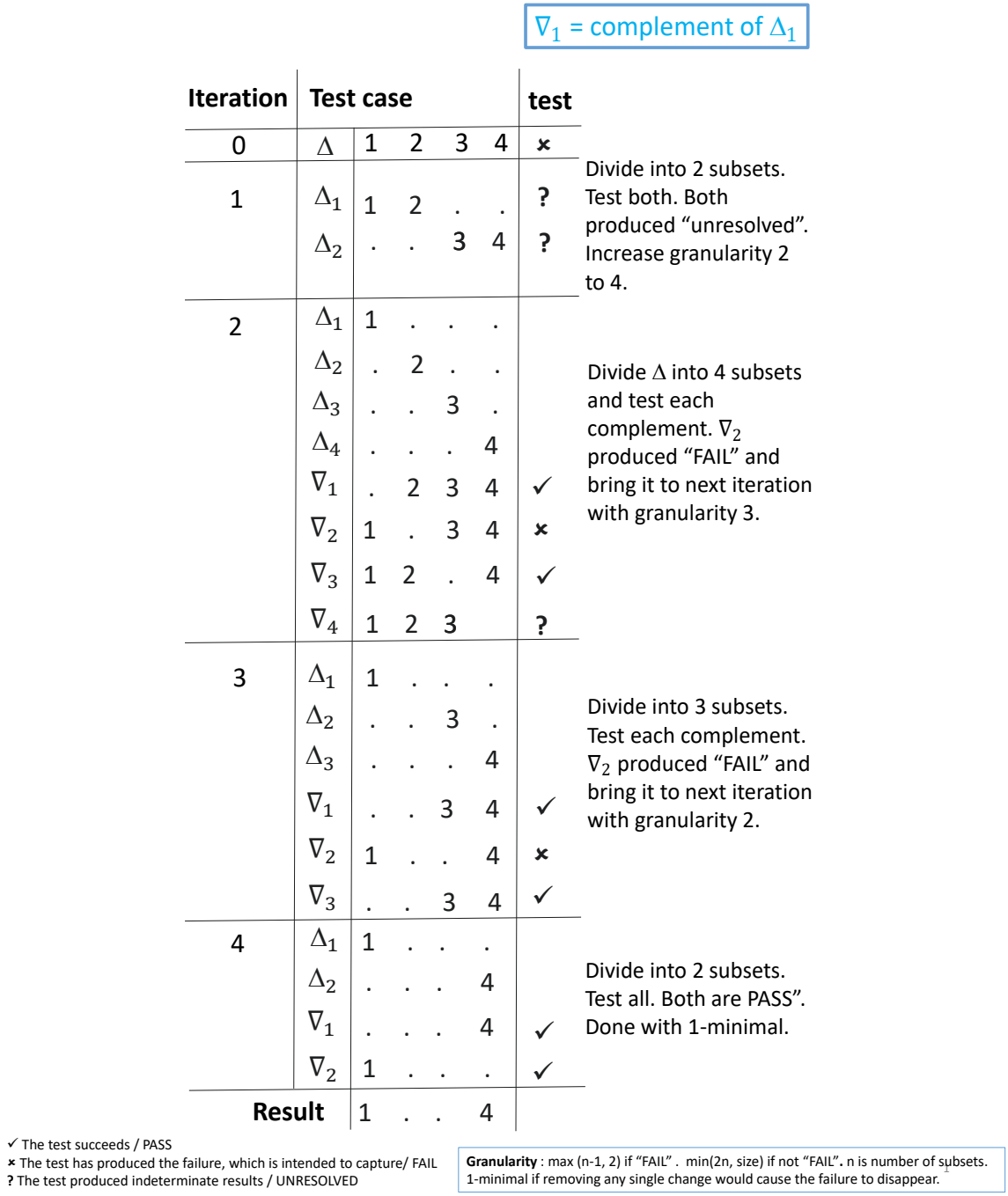


Figure 2.3: The `Complement` strategy.

We illustrate the process of the `Complement` approach in Figure 2.3. The first column describes the number of iterations. The second and third columns show each subset (Δ) or each complement (∇) with the changes included. The fourth column presents the outcome of each test. At Iteration 1, DD first partitions the whole set of changes into two subsets (i.e., Δ_1 and Δ_2), and tests each of them individually. We assume that every test outcome is unresolved. Therefore, we start applying the `Complement` strategy, and increase the granularity from 2 to 4. Note that, since none of the subsets fails, DD uses the first formula (I) (i.e., *min(2n, size)*) to calculate the granularity. At Iteration 2, DD partitions the whole set of changes into four subsets (i.e., Δ_1, Δ_2, Δ_3 and Δ_4). Here, we start testing the complement. For example, the complement of Δ_1 is Δ_2, Δ_3 and Δ_4.

We assume the complement of Δ_2 (i.e., ∇_2) produces the failure, and DD marks it as a minimal, failure-inducing test case. DD then continues to search in it for a shorter one. DD uses the second formula (II) (i.e., *max(n - 1, 2)*) to calculate the granularity (i.e., 3) for Iteration 3. At Iteration 3, DD divides the failing complement (i.e., ∇_2 at Iteration 2) into three subsets and tests each complement. Again, we assume the complement of Δ_2 (i.e., ∇_2) fails, and DD marks it as a minimal, failure-inducing test case. DD then continues to search in it for a shorter one. Again, DD uses the second formula (II) to calculate the granularity (i.e., 2) for Iteration 4. At Iteration 4, DD partitions the failing complement (i.e., ∇_2 at Iteration 3) into two subsets, and tests each complement. We assume both complements could not produce the failure and, DD stops with 1-minimal. We got the smallest test case (i.e., the complement of Δ_2 at Iteration 3) (i.e., ∇_2 at Iteration 3) which causes failure.

Generally, one objection against DD is that achieving the granularity to perform the actual reduction can take too long. To balance between theoretical guarantees and performance, some works introduce the split factor [99], and some introduce the hierarchical-based DD [131]. However, selecting the best value for the split factor is

non-trivial in practice, and there has yet to be a known formula. Despite its age, DD is still an inevitable cornerstone of this field of research and is still one of the most important works in automated test case reduction because of its robust algorithm.

2.3 Def-Use Analysis

A def-use analysis is a data-flow analysis collecting information about how the variables are defined and used in the program. It establishes a relationship between the definition statement where a variable is created and each statement where it is used. With that information, we can automatically detect or collect the statements affected by the variable. Def-use analysis is well known and has been shown to be useful not only for debugging [170], software testing [79], but also for program integration [87], and software maintenance [73].

S_1	int a = 1;	Def
S_2	int b = a + 2;	Use

Figure 2.4: Def-Use Analysis.

An example def-use analysis is shown in Figure 2.4. If the statement S_2 used the same object a which is defined as `int` with value `1` in S_1, S_1 affects S_2. It means there is the relationship between S_1 and S_2. In this way, we can track the data flow in the program and collect the statements affected by variable a.

2.4 Program Slicing

Program slicing is a technique to extract or slice a group of program statements. It is useful and well-known for program analysis, debugging and understanding. It collects program statements affecting the point of interest (slicing criteria). Given a program P, the programmer provides a slicing criterion (l, V), where l is a location in the program

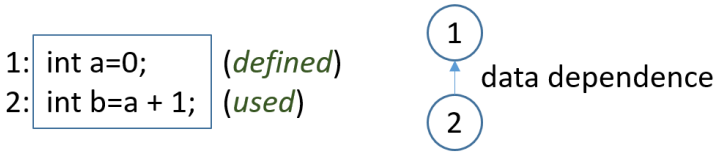


Figure 2.5: Data Dependence.

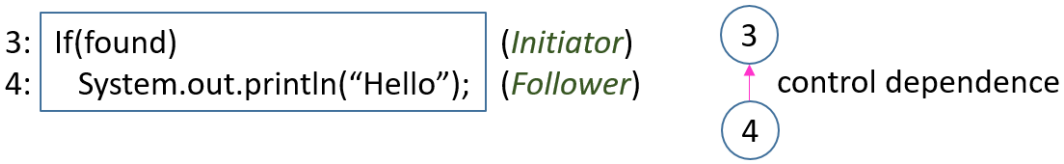
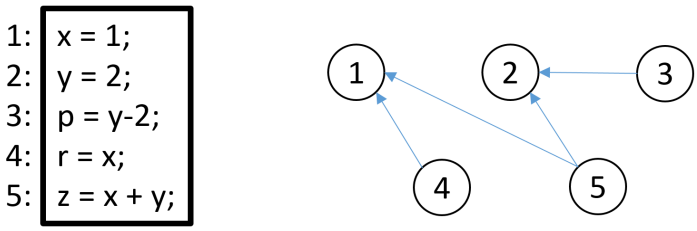


Figure 2.6: Control Dependence.

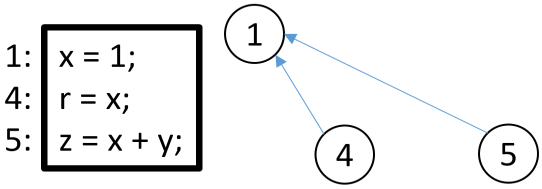
and V is a set of program variables referenced at l. Slicing operates via a program dependence graph (PDG), the nodes of the PDG represent statements or basic blocks, and the edges correspond to data or control dependences between nodes.

- **Data dependence** is similar to the Def-Use analysis. Recall that, for example, instruction A is data-dependent on instruction B if B uses the same object defined at A. Figure 2.5 illustrates the data dependence for a program fragment with two lines and the corresponding PDG with two nodes. Node 1 represents Line 1, and Node 2 represents Line 2. In Line 2, a statement utilizes the same object a, which is defined in Line 1. Therefore, Node 2 is data-dependent on Node 1 in the PDG.

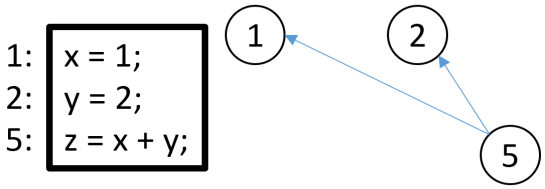
- **Control dependence** follows initiator-follower rules. For example, an instruction Y has a control dependency on a preceding instruction X if the outcome of X determines whether Y should be executed or not. Figure 2.6 shows the control dependence for a program fragment with two lines and the corresponding PDF with two nodes. Node 3 represents Line 3, and Node 4 represents Line 4. The execution of Line 4 depends on the evaluation result in Line 3 (i.e., the boolean outcome/ value of variable $found$). Therefore, Node 4 is control-dependent on Node 3 in the PDG.



(a) A program fragment to be sliced and its corresponding program dependence graph (PDG).



(b) Forward slice for the variable x.



(c) Backward slice for the variable z.

Figure 2.7: Forward Slicing vs Backward Slicing.

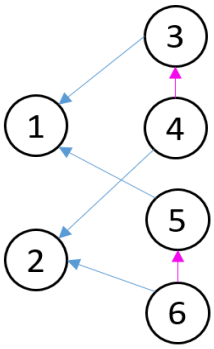
There are two ways of slicing (i.e., forward and backward slicing). While **forward slicing** starts from the criteria point onward, **backward slicing** starts from the criteria point backward. In other words, forward slicing slices the program's statements which are affected by the slicing criteria, while backward slicing collects the program's statements, which can have some effect on the slicing criteria.

We present forward and backward slicing examples in Figure 2.7. In Figure 2.7a, we have a program fragment with five lines and its corresponding PDG consisting of five nodes (each node represents each program line). In PDG, Node 5 is data-dependent on Nodes 1 and 2 because z is defined with the summation of x and y at Line 5, while x is defined at Line 1, and y is defined at Line 2. Similarly, Node 4 is data-dependent on Node 1 because Line 4 utilizes the variable x defined at Line 1. Node 3 is data-dependent on Node 2 because Line 3 utilizes the variable y defined at Line 2.

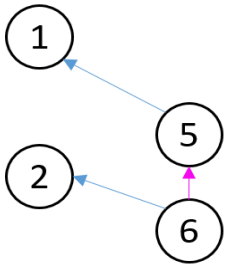
Figure 2.7b shows the forward slicing for variable x while Figure 2.7c shows the backward slicing for variable z. Since Line 4 and Line 5 utilizes the same object x defined at Line 1, the forward slice includes Nodes 1, 4 and 5. In other words, the value of x at

```
1:  int n = Integer.parseInt(System.console().readLine());
2:  int x = 0;
3:   if(n>0)
4:     {y = x - 1 ; }
5:  else if(n<0)
6:     {y = x + 1; }
```

(a) A program fragment to be sliced.



(b) Static slice for the variable y.



(c) Dynamic slice for the variable y when n = -5.

Figure 2.8: Static Slicing vs Dynamic Slicing.

Line 1 affects the value of r at Line 4 and z at Line 5. On the other hand, as shown in Figure 2.7c, the backward slice includes Nodes 1, 2 and 5 because Line 5 utilizes the same objects x defined in Line 1 and the same object y defined in Line 2. In other words, the value of z at Line 5 is affected by the value of x defined at Line 1 and the value of y defined at Line 2.

Slicing can be done dynamically or statically. **Static slicing** collects program lines for every possible execution pattern. Thus, static slicing contains all statements that may have affected the variable's value at a program point for any arbitrary execution of the program. On the other hand, **dynamic slicing** slices run-time effective program lines for specific input. Therefore, a dynamic slice includes all statements that affect the value of a variable at a program point for a particular execution of the program.

Figure 2.8 illustrates examples of static and dynamic backward slicing for a program

fragment with six lines. Specifically, Figure 2.8b shows the static slice for variable y while Figure 2.8c shows the dynamic slice for variable y if the input value for n is -5. The blue arrows represent data dependences, and the pink arrows represent control dependences in the figure. As shown in Figure 2.8b, the static slice includes all program lines (i.e., Line 1, 2, 3, 4, 5, and 6). In contrast, as shown in Figure 2.8c, the dynamic slice contains all executed program lines (i.e., Line 1, 2, 5 and 6) if the input value for n is "-5". While the static slice has six nodes (i.e., six lines), the dynamic slice has four nodes (i.e., four lines).

Therefore, static slicing is often conservative, leading to very large static slices, while dynamic slicing produces more precise slices, and the slice can be significantly reduced, leading to a finer localization of the criteria point. In addition, dynamic slicing, which aims to isolate the relevant code and data or control dependence for the execution of a program, has been shown very useful for debugging [19, 170] and fault localization [20, 54].

CHAPTER 3

LITERATURE REVIEW

This literature review explores the current state of research on automated analysis and debugging, focusing on identifying the context and contributions of prior research and literature relevant to our work. First, we discuss the overview of automated debugging in software engineering and its challenges. We then describe some studies related to delta debugging, program slicing and their limitations.

3.1 Automated Debugging

Debugging is a critical process in software engineering, aimed at identifying the causes of program failures and removing errors from a program. It is time-consuming and expensive, especially if the search space is big and the software application we debug is complicated. Therefore, over the decades, numerous works have proposed different automated debugging approaches [33] [70] [94] [144] [189] [48] [190] to solve this problem. The primary objective of automated debugging is to reduce the search space, with or without human interaction, before the actual fault localization is conducted.

In the context of software engineering, there are generally two approaches to debug-

ging processes. The first approach is **backward tracking**, which involves tracing the program execution backward, starting from the point of failure until the faulty statement is identified [19, 60, 104, 183]. The second approach is known as **forward tracing**, which involves tracing the program execution forward, starting from the program entry point until the faulty statement is identified [47, 92, 186]. Both approaches have their advantages and disadvantages. In general, backward tracking is considered more effective for identifying the root cause of the bug, while forward tracing is more useful for understanding the program flow and behavior.

The effectiveness of debugging approaches also depends on other factors such as the nature and complexity of the program being debugged, and the debugging expertise of the developers. For instance, in standard Java, which typically starts from a single entry point, the first approach of backward tracking can be used to achieve the debugging goal. However, in Android Java, which has several entry points, the first approach alone may not be sufficient to identify the root cause of the problem. Similarly, if only the second approach of forward tracing is used, it may be unable to reach faulty statements if the test input is invalid.

Automated debugging can be conducted statically [45] [100] [86] or dynamically [110] [51] [178]. **Static analysis** provides a comprehensive understanding of the system's architecture, which can help limit the search space and identify potential problem areas. However, static analysis can be time-consuming and may provide additional information that is not relevant to the debugging process. In contrast, **dynamic analysis** allows for a more specific analysis of the system's behavior during runtime, which can be more effective in identifying faults. However, dynamic analysis can miss certain problems if the specific test scenario fails to trigger the desired failure. For example, utilizing only static analysis would not completely understand the Android apps, and predefined test cases would not cover handling Android's events due to non-determinism in execution.

Particularly, to leverage the static analysis, some works use the **probability of being faulty** via testing [47, 154] [118] [176] [17]. Nevertheless, similar to other test-based methods, the accuracy of the results is directly related to the quality of the test cases used to generate the probabilities. For Android apps, MZoltar [121] offers a dynamic analysis of mobile apps that provides a diagnostic report to identify possible defects of an instrumented app via spectrum-based fault localization. However, it relies on the test suite, including both passed and failed tests for the same scenario, which is limited to real-world applications' failure. Moreover, the limited test suite that could cover a wide range of asynchronous event sequences is questionable.

On the other hand, not to miss the buggy code in dynamic analysis while debugging, researchers introduce the **template-based** approach [154] for a particular error. However, the tool's effectiveness depends on the availability of appropriate templates (dataset) that accurately capture the relevant characteristics of the targeted errors. Therefore, it is crucial to consider the nature of the program being analyzed and the types of bugs it may contain. Moreover, different programs may have different error patterns, which means that a tool or technique that works well for one program may not be as effective for another. Thus, researchers need to tailor their analysis methods to the particular program and the types of errors they are trying to detect and fix.

3.2 Delta Debugging

Delta Debugging (DD) approach introduced by Zeller et al. [186] [80], which can identify the smallest possible subset of code that can reproduce a given error. We can apply DD for arbitrary input without prior knowledge about the test case format. It has been used in various software systems, including traditional desktop applications [182], compilers [131], browsers [186], Web applications [77], and microservice systems [191]. Some work targeted textual failure-inducing inputs, while some targeted test case

reduction.

Hodován and Kiss suggested speeding improvements to the original DD algorithm with parallelization [83] while maintaining its guarantee of 1-minimal. By transforming the textual inputs to a tree representation and applying the DD algorithm to the levels of the tree, Misherghi and Su introduced hierarchical DD (HDD) to reduce the syntactically broken intermediate test cases [131]. However, it still created incorrect test cases because it removed nodes that caused syntax errors.

Therefore, Hodován and Kiss presented extended context-free grammar (eCFGs) for the tree-building step of HDD [82], and they achieved smaller outputs with fewer executed tests via more balanced tree representations. To reuse available non-eCFGs and balance recursive structures, Hodovàn et al. also proposed a variant of the original HDD called Coarse HDD [84]. Moreover, they discovered that analyzing the grammar to avoid excessive removals and using a new caching approach could improve reduction speed and increase efficiency [85]. All the approaches mentioned above aim at textual failure-inducing inputs and would not be suited for Android apps with multiple entry points with asynchronous events.

Some researchers also utilize DD for test case reduction with a broader application area. Colin et al. [150] minimized faulty event sequences of distributed systems. FuzzSMT worked on finding crashes of SMT solvers for bitvector and array instances [41]. Moreover, DD was used to reduce unit tests [108] [109] or even unit test suites [69]. For example, Hammoudi et al. proposed DD to minimize manually-written test suites (GUI test cases) for web applications [77]. In addition, some work employs DD by using historical data. For example, by leveraging the whole evolutionary history of the program, Artho presented iterative DD [29]. Similarly, to locate the sources of the regression faults introduced during some software evolution, Yu et al. presented a DD approach that analyzes traces and looks for the error sources using a defined test suite [182]. However,

these approaches are not designed for analyzing and handling the Android apps, where they become ineffective in detecting event sequences.

3.2.1 Delta Debugging for Android Apps

For Android apps, the algorithms based on delta-debugging have been proposed to minimize GUI event sequences for reaching a particular target activity [47] and reproducing a crash [92]. However, both techniques target to decrease the events in a failure-inducing input trace. It demands going beyond event sequence minimization to achieve the goal of aiding developers in locating the root causes of a crash.

3.3 Slicing

Dynamic program slicing was introduced by Korel and Laski [104], and it has been broadly studied in the literature, mainly for traditional server/ desktop applications [122] [24] [114]. Agrawal et al. introduced dynamic slicing using graphs of instances [19]. Agrawal et al. also presented dynamic slices to locate reaching definitions of pointers and to operate across methods for programs with pointers [18]. For multi-threaded programs, Duesterwald et al. expanded dynamic slicing via data dependencies between inter-thread [58]. To lessen the runtime overhead, Gupta et al. suggested a hybrid approach by instrumenting a limited number of statements [74]. Similarly, Tallam et al. presented a method to collect the traces for threads relevant to the bug [157]. Around the same time, Zhang et al. proposed an efficient dynamic slicing to find reaching definitions [190]. They first split the trace into chunks and then collected the variables defined in a chunk.

3.3.1 Slicing for Web Applications

The slicing technique has been presented in Web applications [123] [145] [161]. 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 cause unique challenges to slicing with (1) life cycle management rules among components (for example, the principles between fragment and activity), and (2) intercomponent communication employed not only in the same application but also across different applications.

3.3.2 Slicing for Java

Dynamic slicing for traditional Java programs [169] [156] [168] has been proposed. However, 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.

3.3.3 Slicing for Android Apps

Unlike Java programs, precise dynamic slicing for Android is inherently more difficult due to the non-deterministic event interleaving in Android. Dynamic slicing normally operates on an execution trace from an interesting program point (e.g., a program crash point) after running a program. For example, a coarse-grained trace would be a stack trace. However, different from many Java programs that have the unique call stack, Android's Inter-Component Communication (ICC) permits each event to have its separate call stack once it is launched (e.g., startActivity) via the corresponding intent. Then a call stack of the launched event starts from onCreate according to the life-cycle of Android's callback methods. As a result, the ICC feature, together with the event-driven mechanism, make dynamic slicing more challenging in Android.

Some prior works use static program slicing for Android apps. For example, Gibler et al. implemented taint-aware slicing for finding potential privacy leaks in Android apps [65], while Titze and Schütte provided a slicing tool called Apparecium to statically detect data flows in Android apps from arbitrary data sources to sinks [160]. Similarly, SAAF [86] performs static slicing to detect suspicious behavior patterns for malicious Android apps. Those works focus on finding specific issues, such as security problems [149] [188] and energy consumption [36], rather than detecting the different types of bugs. They also need to address the precise modeling of Android's callback sequences. Moreover, Android has non-deterministic event nature, and conducting static analysis alone can give extra or imprecise slices.

Some dynamic slicing approaches have been designed for Android Apps. AndroidSlicer performs dynamic slicing by modeling asynchronous data and the control dependences of Android apps. Mandoline [22] presents dynamic slicing via alias analysis. However, there needs to be more consideration of event-based lifecycle awareness for Android apps and the connection between the statements and application resources such as layout definitions and user interface strings.

CHAPTER 4

EVENT-AWARE PRECISE DYNAMIC SLICING FOR AUTOMATIC DEBUGGING OF ANDROID APPS

To reduce the search space and expedite the debugging process of Android apps, we present ESDroid: an Event-aware dynamic Slicing technique designed specifically for AnDroid apps. By incorporating awareness of asynchronous events and lifecycle events, this innovative technique significantly narrows down the search space, enabling the generation of precise code slices that greatly enhance the debugging process, leading to more efficient bug identification and resolution. 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).

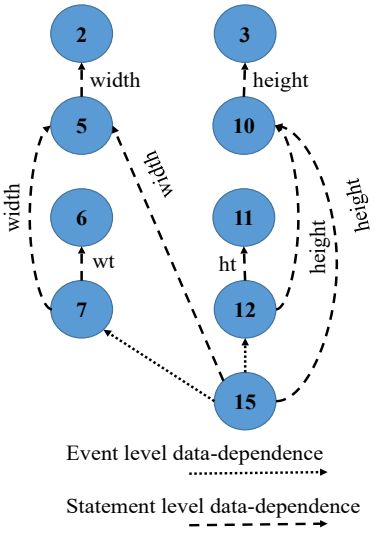
4.1 Introduction

Program slicing [170] 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 [19]. 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.

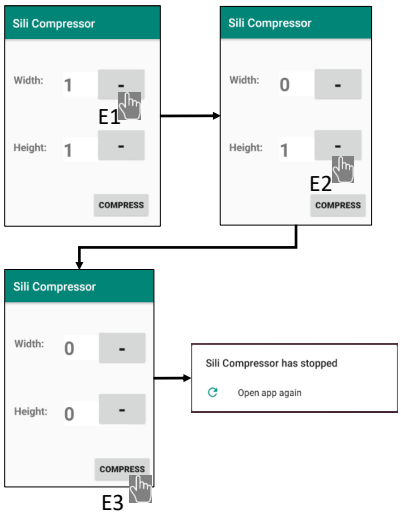
Static slicing techniques perform on a program dependence graph (PDG); the nodes of the PDG represent statements or a basic block, and the edges correspond to data or control-dependences between nodes [88]. Specifically, a directed data dependence edge $S_i \xrightarrow{d} S_j$ means any computation 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 [60], 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 [19] [18] [104], program understanding [169] [170], change impact analysis [24], regression test suite reduction [73], and fault localization [20]. However, dynamic slicing may include redun-

```
1   class SiliCompressor extends Activity {
2      int width=1;
3      int height=1;
4   void widthDecrementClick(View view){                    E1
5         width=width-1;
6         TextView wt = (TextView)findViewById(R.id.width);
7         wt.setText(width+"");
8      }
9      void heightDecrementClick(View view){                E2
10        height=height-1;
11        TextView ht = (TextView)findViewById(R.id.height);
12        ht.setText(height+"");
13     }
14     void compressImageClick(View view){                  E3
15        int maxRatio = width/height;
16     }
17  }
```

(a) App code.



(b) Slice.



(c) Activity state changes (including GUI events).

E1 - - → E2 - - → E3

(d) A randomly generated sequence of user click events.

```
java.lang.ArithmeticException: divide by zero
at com.i.sc.SiliCompressor.compressImageClick(SiliCompressor.java:15)
```

(e) Stack trace.

Figure 4.1: Our Motivation. SiliCompressor app. ArithmeticException has thrown while the program attempted to divide by zero.

dant 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 work, 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 [23] and Mandoline [22]. 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 work 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 Figure 4.1. 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. Figure 4.1a is the simplified app code of SiliCompressor, and Figure 4.1b is the slice produced by AndroidSlicer. Figure 4.1c shows the activity state changes when the user clicks the event sequences shown in Figure 4.1d. Figure 4.1d is the randomly generated event sequence that makes the app fail with `ArithmeticException: divide by zero`. Figure 4.1e 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$\dashrightarrow$E3) are failure-inducing events. Only the statements of the callbacks (i.e., `heightDecrementClick`,

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

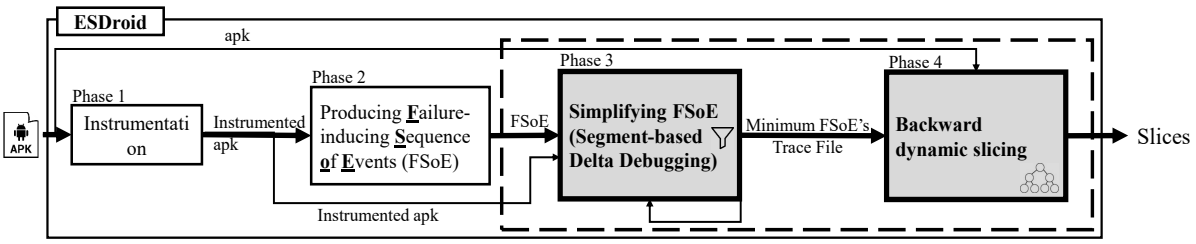


Figure 4.2: Overview architecture of ESDroid

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 [23] and Mandoline [22]) do not consider the input events and include spurious slices, resulting in a larger slice and search space. Thus, it leads to time-consuming tasks for developers. 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 [186]. The delta-debugging has been used in dynamic program slicing to narrow down the search space for faulty code in non-event-based programs [72]. The delta-debugging also has been used to simplify the trace for Android events [47] [92]. 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 work presents ESDroid, an Event-Aware precise dynamic Slicing approach for Android by introducing segment-based delta-debugging into backward dy-

namic slicing. ESDroid first simplifies program inputs (i.e., the third phase in Figure 4.2) when exercising Android apps before backward dynamic slicing (i.e., the fourth phase in Figure 4.2). 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 Figure 4.2). 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.

Figure 4.2 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 [23]. Note that we use the timestamp only for the node (instruction) creation, which is important for detecting dynamic data dependences [169] and distinguishing between objects created at the same allocation site. Section 4.3.1 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 [92] 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 Figure 4.2). 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

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

sequence of events (ΔFSoE). To get the shortest event sequence, ESDroid adopts two strategies; (1) `Divide and Conquer` and (2) `Complement`. Section 4.3.3 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 work makes the following contributions:

- We present ESDroid, a new event-aware dynamic slicing technique for simplifying inputs for Android apps.
- We present how to apply delta-debugging in dynamic slicing to yield a more precise and compact PDG while capturing the same bugs as the state-of-the-art tools AndroidSlicer, and Mandoline.
- We have implemented ESDroid and evaluated it using 38 real-world apps against AndroidSlicer, and 10 apps against Mandoline. The results show that ESDroid outperforms AndroidSlicer and Mandoline by reducing redundant nodes (i.e., up to 72% fewer than AndroidSlicer, and 50% fewer than Mandoline) on the dynamic PDG while maintaining all relevant nodes on the PDG. The evaluation data and the source code for ESDroid are publicly available (GitHub[3], Zenodo[4]).

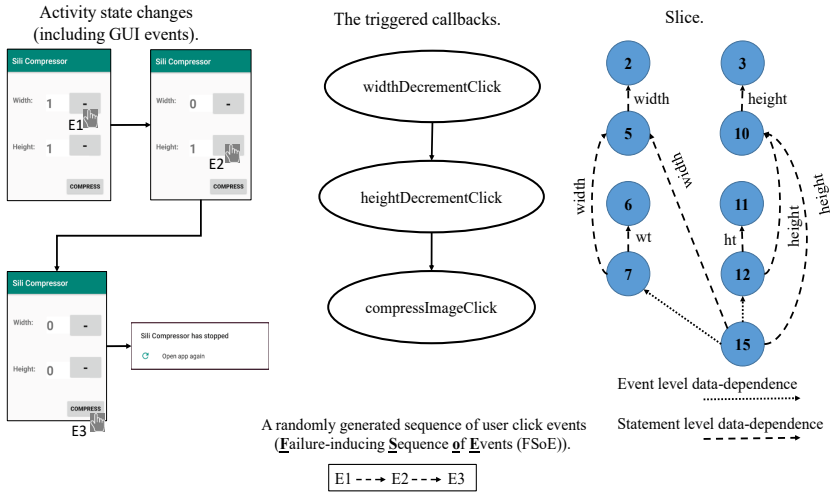
4.2 A Motivating Example

This section uses an example bug found in a SiliCompressor from GitHub shown in Figure 4.3, as our motivating example. We aim to highlight the important insights and

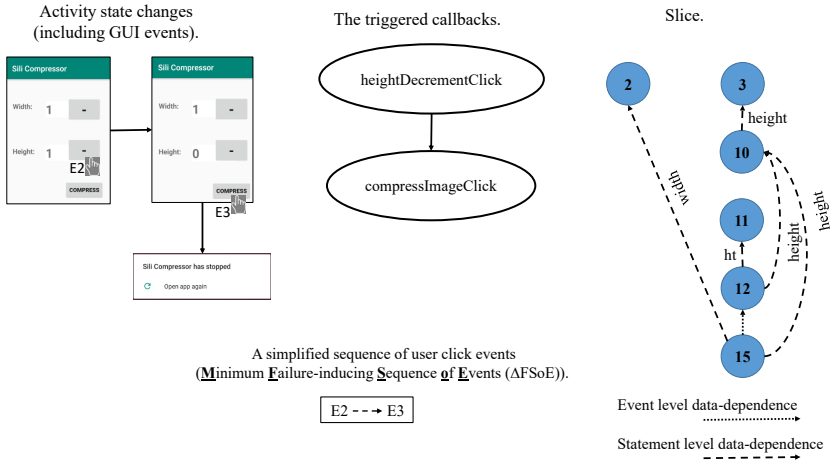
[3]https://github.com/hsumyatwin/ESDroid-artifact
[4]https://doi.org/10.5281/zenodo.7074680

```
1   class SiliCompressor extends Activity {
2     int width=1;
3     int height=1;
4   void widthDecrementClick(View view){                    ┐
5       width=width-1;                                      │
6       TextView wt = (TextView)findViewById(R.id.width);   ├ E1
7       wt.setText(width+"");                               │
8     }                                                     ┘
9     void heightDecrementClick(View view){                 ┐
10      height=height-1;                                    │
11      TextView ht = (TextView)findViewById(R.id.height);  ├ E2
12      ht.setText(height+"");                              │
13    }                                                     ┘
14    void compressImageClick(View view){                   ┐
15      int maxRatio = width/height;                        ├ E3
16    }                                                     ┘
17  }
```

(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).

Figure 4.3: 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 4.1.

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. Figure 4.3a gives the code fragment of the demo app. Figure 4.3b 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". Figure 4.3c 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 Figure 4.3c. 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 4.1 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.

4.2.1 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.

4.2.2 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 4.1 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`$\dashrightarrow$`E3`) because the sub-sequence which includes the last event of FSoE has a higher chance of triggering

Table 4.1: Iteration process of simplifying FSoE for Figure 4.3 - 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	Arithmetic-Exception	fail	Original FSoE.
1	E2--→E3	1	0	Arithmetic-Exception	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.
	E1	-	-	-	-	
2	E3	1	1	1	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).
	E2	1	0	-	pass	

the bug [92]. 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 4.3.3. 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.

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.

4.2.3 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 Figure 4.3b, 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). Figure 4.3c 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.

4.3 Approach

Figure 4.2 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.

4.3.1 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 [163] 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 [31] 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 4.5.2 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 4.5.5.

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 Figure 4.3, 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;

4.3.2 Producing FSoE

ESDroid generates random events to exercise the instrumented apps until the program fails. For example, the event sequence `E1` $\dashrightarrow$ `E2` $\dashrightarrow$ `E3` shown in Table 4.1 triggers an

exception. While SimplyDroid [92] 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 5,000. 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.

4.3.3 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 Figure 4.3, although we removed the event which triggers `widthDecrementClick`, the remaining two click events still trigger the same bug because both the original and reduced event sequences feed the invalid values (i.e., zero) in $height$. The reduction process (i.e., segment-based delta-debugging) is repeated until ESDroid produces a minimum failure-inducing sequence of events (i.e., ΔFSoE), which is used for the later dynamic slicing because, with a shorter sequence, it is easier to find the error in terms of debugging process.

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

- The app exited normally without any crash.
- The app crashed with a different error or exception type.
- The app crashed with the same 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. 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 $s \subseteq s_{✗}$ is $n\text{-}minimal$ if $\forall s' \subset s \cdot |s| - |s'| \leq n \Rightarrow$ (test(s') $\neq$ ✗) holds, where ✗ is the fail outcome. Consequently, s is $1\text{-}minimal$ if $\forall \delta_i \in s \cdot$ test(s − {δ_i}) $\neq$ ✗ 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 ∈ A and x ∉ 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

Table 4.2: 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	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.
	E1-->E2	-	
3	E4	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.
	E3	-	

provide the worst and best-case behavior of the delta debugging process [99]. 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` [186] 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 [92]. 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`.

Example 2. *Table 4.2 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*

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*.

```
1  Tf ⟵ {};
2  n ⟵2;
3  isFailed ⟵ false;
4  if FSoE.size() ==1 then
5  |  (isFailed, Tf) ⟵ test( FSoE, e);
6  end
7  while FSoE.size() >=2 do
8  |  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.
9  |  for each sub-sequence Si in S do
10 |  |  if test( FSoE\Si, e) != null then
11 |  |  |  (isFailed, Tf) ⟵ test( FSoE\Si, e);
12 |  |  end
13 |  |  if isFailed then
14 |  |  |  FSoE ⟵ FSoE\Si;
15 |  |  |  n ⟵ max(n - 1, 2);
16 |  |  |  break;
17 |  |  end
18 |  end
19 |  if !isFailed then
20 |  |  if n == FSoE.size() then
21 |  |  |  break;
22 |  |  end
23 |  |  n ⟵ min(2n, FSoE.size());// Increase granularity and start Complement strategy
24 |  end
25 |  isFailed ⟵ false;
26 end
27 return Tf;

   Procedure: test(list of events St, stack trace e)
28 if St triggers the app crash then
29 |  x ⟵ dumpStack();// print stack trace of crash
30 |  if e == x then
31 |  |  Tf ⟵ logcat();// get all executed program statements
32 |  |  return (true,Tf);
33 |  end
34 end
35 return null;
```

Table 4.3: 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	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.
	E1-->E2-->E3-->E4	pass	
2	E3-->E4-->E5-->E6--> E7-->E8	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)).
	E1-->E2-->E5-->E6--> E7-->E8	-	
	E1-->E2-->E3-->E4--> E7-->E8	-	
	E1-->E2-->E3-->E4--> E5-->E6	-	
3	E5-->E6-->E7-->E8	-	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)).
	E3-->E4-->E7-->E8	fail	
	E3-->E4-->E5-->E6	-	
4	E7-->E8	pass	Divide the last failed event sequence into 2 sub-sequences and test both complements. Both passed. Increase the granularity from 2 to 4.
	E3-->E4	pass	
5	E4-->E7-->E8	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).
	E3-->E7-->E8	pass	
	E3-->E4-->E8	pass	
	E3-->E4-->E7	pass	

other with `E1 --> E2`). We first test for the last sub-sequence (i.e.,`E3 --> E4`). Since `E3 --> E4` triggers the bug, ESDroid uses it as input for the next iteration. At Iteration 2, ESDroid divides the latest sub-sequence, which makes the app fail, into 2 sub-sequences (i.e., one with `E4` and the other with `E3`). ESDroid conducts the testing for the last sub-sequence first (`E4`) and the program fails. Since ESDroid iterates the reduction process until 1-minimal

sub-sequence, it terminates the process and E4 is the event that is responsible for program failure (i.e., ΔFSoE). Note that the granularity for the `Divide and Conquer` *is 2 for every iteration.*

ESDroid adopts the `Complement` strategy for the next iteration if neither sub-sequence produces the bug in the current iteration because the smaller sub-sequences and testing the complement of the smaller sub-sequence gives a higher chance of resulting in program failure [186].

Example 3. *Table 4.3 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.,* $E3 \dashrightarrow E4 \dashrightarrow E5 \dashrightarrow E6 \dashrightarrow E7 \dashrightarrow E8$*). 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.,* $E3 \dashrightarrow E4 \dashrightarrow E7 \dashrightarrow E8$*). 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., $\Delta FSoE$) 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 [92]. 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 [99]. 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.

4.3.4 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 Figure 4.4, 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.

```
1 class Act1 extends Activity{
2 onClick1(...){//E1
3 o1 =1;}//S1t
4 onClick2(...){//E2
5 o2 = o1+2;}//S2t}
```

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

Figure 4.4: Data dependence.

Example 5. *To illustrate the data dependences in our approach, let us revisit the example in Figure 4.3c. 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 Figure 4.4, 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 $o2$ used in S_{2t} (Line 5) depends on object $o1$ defined in S_{1t} (Line 3) at that t time.

Example 6. *In our motivating example in Figure 4.3, if* `widthDecrementClick`*, and* `heightDecrementClick` *are triggered before triggering* `compressImageClick`*,* `compressImageClick` *is data-dependent on both* `widthDecrementClick`*, and* `heightDecrementClick` *via width, and height respectively. The slice in Figure 4.3c contains node 12 because* `compressImageClick` *is data-dependent on* `heightDecrementClick` *via height.*

1 2	`if(condition){//`S_{3t} `System.out.println("True");//`S_{4t}`}`	$S_{4t} \xrightarrow{c} S_{3t}$

3
4
5
6
7
8
	`class Act3 extends Activity{`
	`onCreate(...){//E3`
	`i = new Intent(this,Act4.class);`
	`startActivity(i);}}`
	`class Act4 extends Activity{`
	`onCreate(...){}}//E4`
		E4 $\xrightarrow{c}$ E3

9
10
11
12
13
14
15
	`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

Figure 4.5: Control dependence.

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 Figure 4.5, 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 [60]. 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 Figure 4.5 (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., $\texttt{E4} \xrightarrow{c} \texttt{E3}$).

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 Figure 4.5 (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., $\texttt{E8} \xrightarrow{c} \texttt{E7}$);

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 [163]) when reporting to users.

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

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*.

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

- *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.

4.4 Implementation

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

Instrumentation and Producing FSoE. ESDroid uses Soot [163] to conduct the instrumentation to produce our customized logging information. Regarding producing FSoE, there are several techniques to generate event sequences to exercise Android apps. Monkey-style stress testing is considered the most robust and popular approach to exercise an app based on previous literature [46, 140, 187]. 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*" [46]. 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 [140] because log messages originally generated by Monkey were not easily translatable back to the corresponding *adb* command. We did not use RERAN [67] 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 4.3.4, 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.

4.5 Evaluation

Existing automated debugging techniques for Android Apps include (1) MZoltar [121] 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] https://developer.android.com/studio/command-line/logcat

Table 4.4: 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	[155]
Anymemo	8887.5	27	10.9.922	NullPointerException	[155], [158]
APV PDF Viewer	63.1	3	0.2.6	NullPointerException	[23]
Bankdroid	5199.7	12	1.9.10.6	IllegalArgumentException	[155]
Birthdroid	431.1	3	0.6.3	NumberFormatException	[155]
Bites	49.9	5	1.3	NumberFormatException	[155]
Calculator	2149.3	1	1	NumberFormatException	[4]
CampFahrplan	3223.7	7	1.32.2	IllegalArgumentException	[155]
Carnet - Notes app	5053	22	0.24.1	NullPointerException	[4]
Cowsay	18.7	1	1.3	CalledFromWrongThreadException	[155]
DalvikExplorer	521.6	16	3.4	NullPointerException	[92]
Fdroid	5860.0	10	0.98	SQLiteException	[158]
FishBun Demo	3293	7	0.6.2	NullPointerException	[4]
fooCam	514.9	1	2.0	NullPointerException, SecurityException	[119]
Geometric Weather	4393	14	2.113	ActivityNotFoundException	[4]
GnuCash	7948.0	20	2.1.4	IllegalArgumentException	[158]
LibreNews	3637.7	2	1.4	ArrayIndexOutOfBoundsException	[155]
Linux Deploy	2156	8	2.6.0	IllegalArgumentException	[4]
Man Man	3562	2	2.1.0	ActivityNotFoundException	[4]
Mitzuli	3329.7	2	1.0.7	BadTokenException	[155]
NPR News	17654.3	14	2.4	NullPointerException	[23]
OBSSD - OBS Stream Deck	4085	4	1.2.2	IllegalArgumentException	[4]
Official Cambridge Guide to IELTS	42848	2	11.3.0.0	IllegalStateException	[4]
Olam	715.4	1	1.0	SQLiteException, StringIndexOutOfBoundsException	[23], [155]
PasswordMaker	331.68	3	1.1.11	NumberFormatException	[155]
Ringdroid	607.0	4	2.6	IllegalStateException	[92]
Scale Image View Demo	4277	1	4.0	ActivityNotFoundException	[4]
Scribbler	20.4	3	0.1.8	IllegalFormatConversionException	[155]
SyncMyPic	231.0	8	0.15	NoClassDefFoundError	[92]
Tailscale	2008	1	1.8.3	ActivityNotFoundException	[4]
Tickmate	591.9	6	1.2.0	CursorIndexOutOfBoundsException	[155]
Tippy	88.0	6	1.1.3	ArithmeticException	[92]
Transistor	2993.2	3	1.2.3	RuntimeException	[155], [158]
TripSit	2311.7	8	1.0	RuntimeException	[155]
Vanilla Music	1408	13	1.1.0	CursorIndexOutOfBoundsException, ResourcesNotFoundException	[4]
WeightChart	541.6	6	1.0.4	ActivityNotFoundException	[92]
WhoHasMyStuff	47.3	4	1.0.7	NullPointerException	[92]
Yahtzee	27.4	2	1.1	NumberFormatException	[92]

4.5.1 Experiment Setup and Methodology

4.5.1.1 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 4.4. Ten of

these apps, used in previous literature [23, 92], 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 4.4 lists the information about the evaluated apps. The "Exception Type" column contains information about the specific type of exception that causes the crash, whereas the "Dataset" column represents the dataset or Google Play. Overall, the evaluated datasets contain a wide variety of apps of various sizes (27–17654 KB of Dex code) with 1 to 27 activities. These datasets have different types of exceptions that lead to crashes. We selected these defects based on the following criteria:

C1: Apps from different categories

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

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

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

- seven apps (i.e., `WeightChart`, `DalvikExplorer`, `Ringdroid`, `SyncMyPix`, `Tippy`, `WhoHasMyStuff` and `Yahtzee`) from the previous evaluation of SimplyDroid [92].
- two apps (i.e., `APV PDF Viewer`, `NPR News`) from the previous evaluation of AndroidSlicer [23].
- four apps (i.e., `Fdroid`, `AnyMemo`, `GnuCash` and `Transistor`) from Droixbench [158].
- one app (i.e., `fooCam`) from RelFix [119].

- 13 apps (i.e., Addi, Bankdroid, Birthdroid, Bites, CampFahrplan, Cowsay, LibreNews, Mitzuli, Passw- ordMaker, Olam, Scribbler, Tickmate and TripSit) from DroidDefects [155].

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 exclude 10 apps from Droixbench and 13 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]. 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.

4.5.1.2 Methodology

We ran the event sequence generation for 10 runs to produce FSoE. Each run was terminated after all random events (5,000 events) had been triggered, or when a crash

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

Table 4.5: 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. 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`.

Apps	# of events			# of callbacks			# of method calls			# of instructions			Duration
	Original	ESDroid	(%)	Original	ESDroid	(%)	Original	ESDroid	(%)	Original	ESDroid	(%)	(seconds)
Addi	19	3	84	58	15	74	4087	3754	8	136938	79722	42	727.62
Anymemo	22	5	77	85	58	31	6602	5695	13	464816	428269	8	5033.99
APV PDF Viewer	4	2	50	5	2	60	57	21	63	948	347	63	21.57
Bankdroid	236	23	90	23	18	21	45348	30926	31	155151	102113	34	9276.45
Birthdroid	1097	14	98	21	21	0	222	220	1	905	893	1	11462.30
Bites	471	8	98	55	14	74	231	58	74	2109	389	82	1439.24
Calculator	59	1	98	43	2	95	117	7	94	34974	8983	74	295.06
CampFahrplan	333	7	97	220	139	36	50618	16188	68	1250075	268769	78	3209.03
Carnet - Notes app	58	6	89	237	218	8	4809	3149	34	398939	223098	44	1755.83
Cowsay	65	1	98	64	10	84	244	112	54	4345	1610	63	366.16
DalvikExplorer	47	6	87	8	3	62	468	159	66	8502	1011	88	148.53
Fdroid	91	3	96	1095	1065	3	192583	125202	34	2678033	1878576	30	664.77
FishBun Demo	93	3	96	18	7	61	83	20	75	86049	19959	77	1025.91
fooCam*	3	1	66	199	114	42	106	50	52	811	405	50	77.23
fooCam**	148	3	95	153	68	55	131	76	41	909	500	45	563.39
Geometric Weather	66	5	92	197	191	3	48943	22144	54	585048	332065	43	2910.27
GnuCash	35	10	71	15	13	13	287	252	12	936	922	1	13439.05
LibreNews	66	7	89	30	10	66	101831	37414	63	636956	232652	63	1644.62
Linux Deploy	105	6	94	128	101	21	3328	2657	20	1294897	541846	58	2743.71
Man Man	62	3	95	105	52	50	2203	355	83	1176582	121901	90	363.39
Mitzuli	146	5	96	490	167	65	248861	167884	32	1230777	905576	26	1251.34
NPR News	37	2	94	293	38	87	1798	605	66	25597	10107	61	271.44
OBSSD - OBS Stream Deck	94	17	81	55	34	38	851	383	54	4754980	1200819	75	2034.56
Official Cambridge Guide to IELTS	41	10	75	188	177	5	2018	1991	1	537874	355432	34	12278.07
Olam~	9	4	55	4	4	0	185	185	0	26597	26597	0	317.31
Olam~~	5	2	60	4	4	0	61	61	0	42324	42324	0	120.50
PasswordMaker	26	10	76	21	10	52	896	387	56	30969	18225	41	3016.59
Ringdroid	135	4	97	45	8	82	4647	188	95	26263	2500	90	1963.55
Scale Image View Demo	238	5	97	203	61	69	2902	352	87	295088	52468	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	174283	120950	31	231.67
Tickmate	52	3	94	31	12	61	935	857	8	3491	3093	11	737.23
Tippy	76	12	84	32	17	46	515	330	35	3775	2406	36	12485.64
Transistor	638	2	99	15	10	33	154	118	23	1625	1242	24	1263.84
TripSit	14	1	92	7	4	42	157	107	32	1448	1313	9	178.38
Vanilla Music^	11	2	80	459	453	1	6258	6249	1	92527	83118	10	789.18
Vanilla Music^^	21	4	80	491	468	4	14461	7360	49	153206	93073	39	3776.37
WeightChart	34	3	91	66	12	81	475	115	75	17432	1711	90	822.08
WhoHasMyStuf	1025	26	97	139	8	94	1749	107	93	10820	604	94	19399.56
Yahtzee	131	6	83	2	2	0	5	5	0	33	33	0	18093.75
Mean	143	6	87	130	89	42	18279	10632	42	398720	174780	45	3354

occurred. We conducted our evaluation to answer the following research questions.

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.

4.5.2 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 4.5 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 4.5 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 seconds.

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

Table 4.6: 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 app marked by * refers to the defect that throws `NullPointerExcep- tion`, the app marked by ** refers to the defect that throws `SecurityExcept- ion`, the app marked by ~ refers to the defect that throws `SQLiteException`, the app marked by ~~ refers to the defect that throws `StringIndexOutOfBoun- dsException`, the app marked by ^ refers to the defect that throws `CursorInde- xOutOfBoundsException`, and the app marked by ^^refers to the defect that throws `ResourcesNotFoundException`. 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 (4.1) and matrix (4.2), respectively.

Apps	# of instructions				
	Phase 2	Phase 3	(%) (4.1)	Phase 4	(%) (4.2)
Addi	136938	79722	42	721	99
Anymemo	464816	428269	8	2222	99
APV PDF Viewer	948	347	63	49	95
Bankdroid	155151	102113	34	522	99
Birthdroid	905	893	1	100	89
Bites	2109	389	82	91	96
Calculator	34974	8983	74	15	99
CampFahrplan	1250075	268769	78	1010	99
Carnet - Notes app	398939	223098	44	6697	98
Cowsay	4345	1610	63	86	98
DalvikExplorer	8502	1011	88	184	98
Fdroid	2678033	1878576	30	1731	99
FishBun Dem	86049	19959	77	262	99
fooCam*	811	405	50	31	96
fooCam**	909	500	45	132	85
Geometric Weather	585048	332065	43	2036	99
GnuCash	936	922	1	60	94
LibreNews	636956	232652	63	163	99
Linux Deploy	1294897	541846	58	8591	99
Man Man	1176582	121901	90	5980	99
Mitzuli	1230777	905576	26	1203	99
NPR News	25597	10107	61	412	98
OBSSD - OBS Stream Deck	4754980	1200819	75	9821	99
Official Cambridge Guide to IELS	537874	355432	34	1275	99
Olam~	26597	26597	0	295	99
Olam~~	42324	42324	0	148	99
PasswordMaker	30969	18225	41	839	97
Ringdroid	26263	2500	90	390	99
Scale Image View Demo	295088	52468	82	182	99
Scribbler	132	83	37	38	71
SyncMyPic	356	275	23	91	74
Tailscale	174283	120950	31	1842	99
Tickmate	3491	3093	11	152	96
Tippy	3775	2406	36	317	92
Transistor	1625	1242	24	104	94
TripSit	1448	1313	9	51	96
Vanilla Music^	92527	83118	10	3171	97
WeightChart	17432	1711	90	292	98
WhoHasMyStuff	10820	604	94	159	99
Yahtzee	33	33	0	21	36
Vanilla Music^^	153206	93073	39	3300	98
Mean	392780	174779	45	1336	94

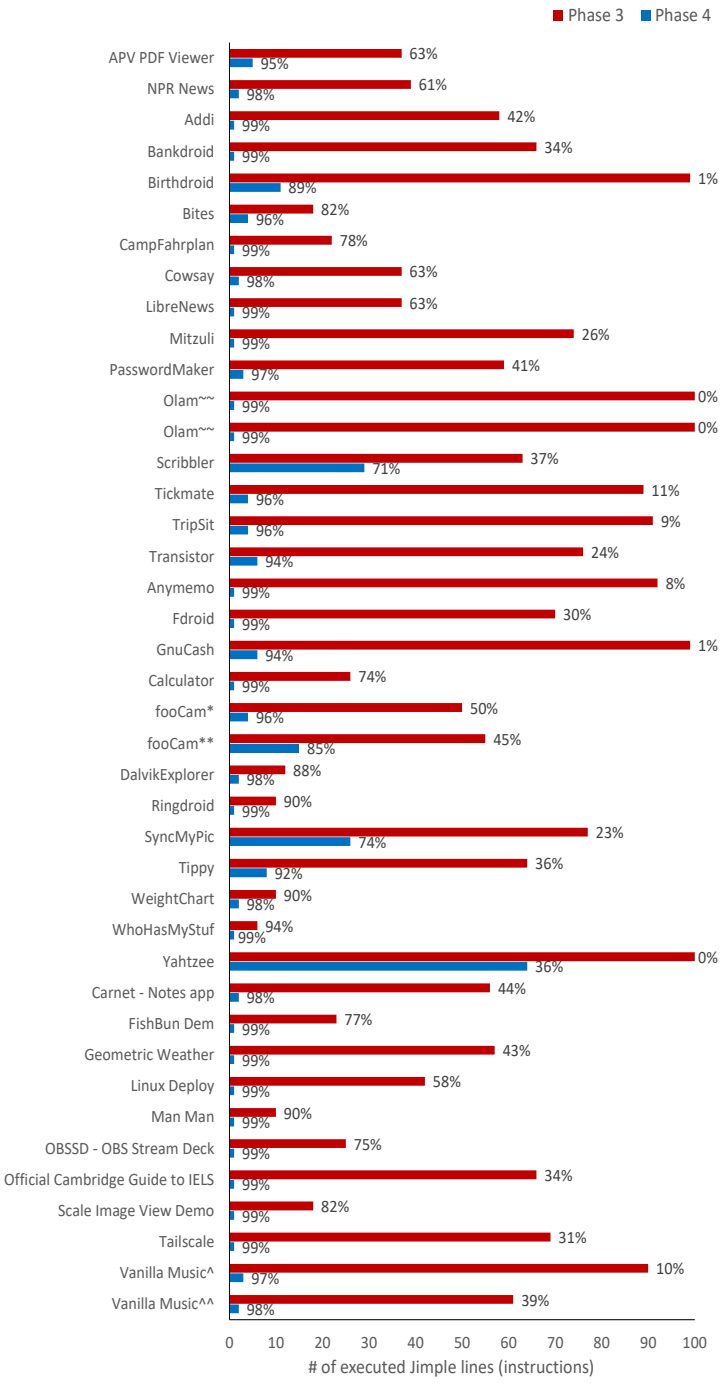


Figure 4.6: The reduction rate (in percentage) for the number of instructions executed in Phase 3 and Phase 4. The app marked by * refers to the defect that throws `NullPointerException`, the app marked by ** refers to the defect that throws `SecurityException`, the app marked by ~ refers to the defect 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`.

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 4.5, 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 4.5, `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.

4.5.3 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 Figure 4.2. Figure 4.6 shows the reduction results for 41 defects of 38 apps. In Table 4.6, 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.

(4.1)
$$Reduction\ rate\ in\ Phase\ 3 = \frac{\#\ of\ instructions\ executed\ in\ Phase\ 2 - \#\ of\ instructions\ executed\ in\ Phase\ 3}{\#\ of\ instructions\ executed\ in\ Phase\ 2}$$

(4.2)
$$Reduction\ rate\ in\ Phase\ 4 = \frac{\#\ of\ instructions\ executed\ in\ Phase\ 2 - \#\ of\ instructions\ executed\ in\ Phase\ 4}{\#\ of\ instructions\ executed\ in\ Phase\ 2}$$

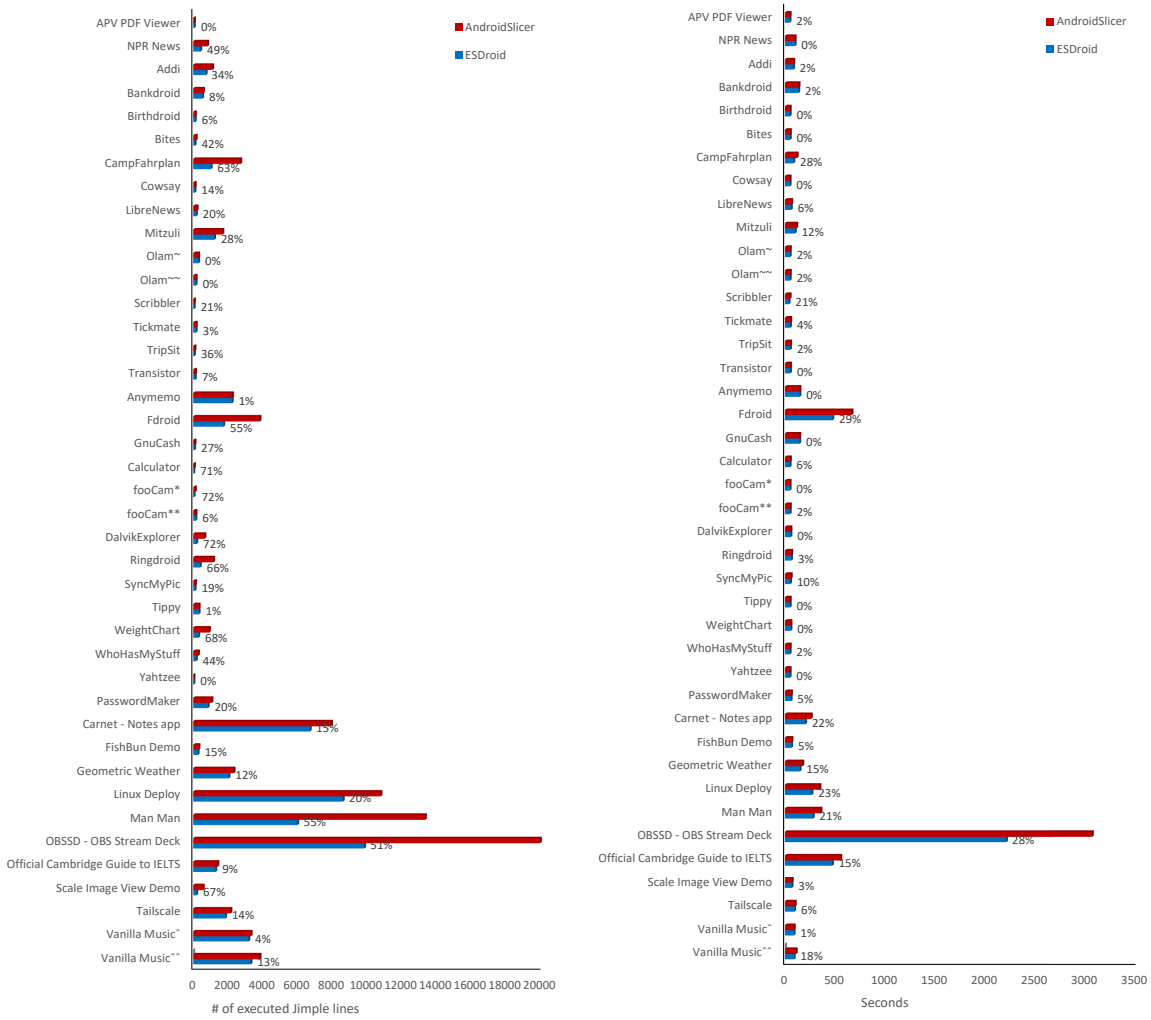
The rows of Table 4.6 and Figure 4.6 show that the reduction rate in phase 4 is higher than in phase 3. At phase 4, the maximum reduction rate is 99% (i.e., `Fdroid`, `Calculator`, `FishBun Dem`, `Geometric Weather`, `OBSSD - OBS Stream Deck`, `Official Cambridge Guide to IELS`, `Scale Image View Demo`) 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.

4.5.4 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



(a) # of executed Jimple lines (i.e., slice). (b) The time taken (in seconds).

Figure 4.7: 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 `StringIndexOutOfBoundsExce- ption`, the app marked by ^ refers to the defect that throws `CursorIndexOutOf- BoundsException`, and the app marked by ^^ refers to the defect that throws `ResourcesNotFoundException`.

Figure 4.7a shows the number of Jimple instructions in the generated slice of both approaches (i.e., AndroidSlicer, and ESDroid), whereas Figure 4.7b compares the time taken by each approach in generating the dynamic slice. The numbers given beside the bars in Figure 4.7a and Figure 4.7b 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 Figure 4.7a 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 Figure 7a using statistical and effect size tests. In particular, we used the Wilcoxon rank sum test [50] and the Vargha-Delaney's $\hat{A}12$ effect size [164]. 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 Figure 4.7b show that the overall time taken by both approaches to perform the dynamic slicing is similar (i.e., from 0% to 29%). These results illustrate the efficiency of our algorithm in performing dynamic slicing without incurring too much additional overhead. In fact, for the `Fdroid`, ESDroid can generate the dynamic slice faster than AndroidSlicer because the size of the trace log (i.e., executed instructions) for `Fdroid` is the largest of all the apps used in our experiment and the analysis time (i.e., checking against static

PDG) shows longer duration. In general, the test case with more redundant events with statements that do not affect the failure point is more likely to include spurious slices (e.g., `Calculator`, `DalvikExplorer` and `fooCam`). Moreover, even with a smaller number of callbacks and events in our experiments, ESDroid still reduced a substantial portion of the redundant PDG nodes. We believe increasing the events will favor ESDroid even further.

4.5.5 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

Table 4.7: 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

related to the failure point, we manually analyzed the differences between the output of ESDroid and the output of AndroidSlicer. Our analysis confirmed that the slices generated by both ESDroid and AndroidSlicer included the statements affecting the failure point. Since both ESDroid and AndroidSlicer include the instructions related to the failure point, a thinner slice generated by ESDroid is a better outcome because it reduces the time taken by the developers to inspect the slice during debugging to state one enhancement.

4.5.6 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 [22], and the motivating example). We exclude one of Mandoline defects because we cannot reliably reproduce the exception in the `Habdroid` app after running the test generation 10 times with different seed values (does not satisfy C3). Table 4.7 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 ap-

Table 4.8: 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

proaches. 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 4.8 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.

4.5.7 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. Note that 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 [31] 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. Further-

more, 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 [23, 92, 119, 155, 158].

4.6 Related Work

Delta-Debugging: Several approaches have applied delta-debugging to identify the failure-inducing deltas in traditional desktop applications [72, 182], compilers [131], browsers [186], Web applications [77], and microservice systems [191]. 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 [47], and for reproducing a crash

SimplyDroid [92]. 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 work 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 [123, 161]. 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 [169] 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) [93, 113, 139, 141, 175]. To handle the unique characteristics of Android apps,

MZoltar [121] 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 [22, 23, 86]. SAAF [86] 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 [55, 103, 119, 124, 158, 180]. 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.

4.7 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.

CHAPTER 5

COMPLEMENT OF DYNAMIC SLICING FOR ANDROID APPLICATIONS WITH DEF-USE ANALYSIS FOR APPLICATION RESOURCES

To address the limitations of existing static and dynamic debugging techniques for Android apps, as our second contribution, we propose a novel approach called SfR (Slicing for Resources), which identifies the dependences between the program statements and the application resources to complete the slice for Android applications. We performed the static analysis to generate the resource dependence graph (RDG), which includes data dependences on application resources. We integrated RDG in AndroidSlicer and evaluated on 3 Android applications. The result shows that SfR is more efficient in accuracy than the existing state-of-the-art dynamic slicing technique named AndroidSlicer.

5.1 Problem and Motivation

Program slicing [170] is to extract the program statements that affect the values computed at some point of interest (i.e., a particular statement or variable, often referred as a slicing criterion). While static slicing considers all possible program paths leading

to the slicing criterion, dynamic slicing focuses on one concrete execution for the given input [19]. Due to Android's event-driven nature, dynamic slicing for Android is more challenging than that for traditional Java programs. Furthermore, mobile apps rely on application resources, and thus a slicing solution has to consider data flowing through application resources.

5.2 Background and Related Work

Static slicing techniques typically operate 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 [88]. The dynamic PDG, which is a subgraph of the static PDG [60], consists of only those nodes and edges that are exercised during a particular run. Precisely, a dynamic slicing tool is first to collect an execution trace of a program by instrumenting the program. Then, the tool checks the control and data dependences of the trace statements, identifying 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 [19] [18]. While AndroidSlicer [23] performs dynamic slicing by modeling asynchronous data and control dependences of Android apps, [43] presents the dynamic slicing using alias analysis. However, the prior techniques limit locating the fault in application resources such as layout definitions and user interface strings. Likewise AndroidSlicer, SfR uses dynamic slicing to produce the program slices that aid debugging for Android apps. However, SfR differs from AndroidSlicer in that it can locate the fault if the bug is in application resources by offering the data dependences on the application resources.

5.3 Approach and Novelty

SfR consists of three major stages: (1) def-use analysis for application resources, (2) dynamic backward slicing, (3) slice complement.

Def-Use Analysis for Application Resources. Basically, we use def-use analysis for data dependences in PDG. If the statement S_2 used the same object a which is defined as `int` with value `1` in S_1, S_2 is data-dependent on S_1.

S_1	`int a=1;`	Def
S_2	`int b=a+2;`	Use

R_d	`<TextView android:id="@+id/tv" android:text="@string/m_t"/>`	Def
S_u	`TextView t = (TextView) findViewById(R.id.tv);`	Use

In our approach, for data dependences on resources, we use predefined keywords (i.e., `findViewById` for "use" and `android:id` for "def"), and we use a unique resource name for the element as a reference. If the statement S_u contains the predefined keyword indicating a "use" of the application resource (i.e., `findViewById`) with a unique resource name for the element (i.e., *tv*) which is defined as `TextView` with a string value of *m_t* in resource R_d, S_u is data-dependent on R_d. In this way, engineers can enhance the default set of keywords characterizing uses and definitions of application resources. Note that SfR cannot handle different resources with the same resource name (e.g., same resource name for widgets in different Activities). However, we aim to show the slicing quality improvement, and SfR is enough to prove it.

In static RDG, a node can be either a tag element or a statement, and an edge corresponds to data dependence on application resources. By using the "def" keyword, SfR constructs the mapping (we call $rMap$) with a reference (i.e., a unique resource name for the element) and the corresponding tag element, including value and attributes before generating RDG. SfR builds a static RDG by scanning "use" keywords. If SfR found

the “use” keyword in a statement, the statement is marked as “use” with a reference to link to the corresponding tag with the help of rMap.

Dynamic Backward Slicing. A backward dynamic slice is the set of instructions whose execution affects the slicing criterion (i.e., the instructions on which the slicing criterion is data or control dependent, either directly or transitively) [104]. Inspired by AndroidSlicer, SfR generates the backward dynamic slice from the point of interest by using dynamic PDG that includes asynchronous data and control dependences.

1	−<string name="m_t">Themen</string> +<string name="m_t">Thread</string>	strings.xml
2	<TextView android:id="@+id/tv" android:text="@string/m_t"/>	dx.xml
3	TextView t = (TextView) findViewById(R.id.tv);	
4	String wt = t.getText().toString();	Dj.java

(Input propagation) Line 1 ⟶ Line 2 ⟶ Line 3 ⟶ Line 4

(a) App code.

Instruction number: Instruction
36612: onCreate:...Dj: $r8 = virtualinvoke **$r11**.<...TextView: ... getText()>()
36611: onCreate:...Dj: $r11 = (...TextView) **$r2**
36610: findViewById:...: $r1 = virtualinvoke $r2.<...>(**$i0**)

(Value propagation) $r8 ⟶ $r11 ⟶ $r2 ⟶ $i0

(b) Slice generated by AndroidSlicer.

```
android:id="@+id/tv" android:text="@string/m_t" />
```

(c) Complement generated by SfR.

Figure 5.1: An example bug found in `NewPipe` app.

Slice Complement. In this stage, SfR completes the slice by using static RDG. Specifically, for each instruction in slice generated by the first stage (i.e., dynamic backward

slicing), SfR checks against the static RDG recursively and extracts the corresponding tag element. Since the slice generated by AndroidSlicer includes Jimple instructions, SfR provides the separate output for extracted tag elements. Particularly, we aim to help developers by providing all statements and application resources affecting the point of interest.

Figure 5.1 shows an example bug [1], the wrong text (`Themen`) for label on user interface (UI), found in the `NewPipe` app. Specifically, the expected value for wt is `Thread`, however, `TextView` object returns the wrong text (`Themen`). In Figure 5.1b, AndroidSlicer generated the slice from the point of interest (i.e., $r8$ holding wrong value `Themen`), and it included Line 4 (Instruction number 36612) and Line 3 (Instruction number 36611 and 36610) and missed the location of the fault in application resources (i.e., Line 1 and 2). In Figure 5.1c, SfR generated the complement by using RDG. Specifically, the variable (tv) used at Line 3 is defined at Line 2.

5.4 Results and Contributions

Experiment. We implemented SfR on AndroidSlicer to evaluate the effectiveness of our approach because (1) it is publicly available, and (2) it is one of the state-of-the-art slicing techniques for Android apps. Although AndroidSlicer does not target application resources, we chose AndroidSlicer to compare because (1) we aim to show that SfR can improve the slicing quality for Android apps, and (2) no slicing tool is available to compare if the bug is located in Android application resource. Our evaluation studies the research question **RQ**: Does SfR help to improve the quality of the slices generated by AndroidSlicer?

Results and Discussions. Table 5.1 shows the experiment results. We chose the apps whose traces were small and verifiable within a reasonable effort and manually computed

[1] https://github.com/TeamNewPipe/NewPipe/issues/5546

the slices w.r.t. the slicing criterion. We then compare the manual slices with the output produced by SfR and AndroidSlicer and calculate the recall (R), precision (P), and F-Measure (F) achieved by each tool to answer **RQ**. We denoted "instructions in computed slice" as I_c and "instructions in manual slice" as I_m. Our experiments show that using def-use analysis for application resources is effective and achieves 96% accuracy on average if the fault location is in application resources. Note that the complement includes the corresponding elements and slices generated by AndroidSlicer consists of the Jimple instructions. Hence, we counted an attribute of an element as one instruction to calculate F-measure.

$R = \frac{\lvert\{I_c\} \cap \{I_m\}\rvert}{\lvert\{I_c\}\rvert}$	$P = \frac{\lvert\{I_c\} \cap \{I_m\}\rvert}{\lvert\{I_m\}\rvert}$	$F\text{-}measure = 2\frac{R*P}{R+P}$

Table 5.1: Accuracy. Instructions are denoted as IS.

App	**Manual**	**AndroidSlicer**				**SfR**			
	#IS	**#IS**	**R%**	**P%**	**F%**	**#IS**	**R%**	**P%**	**F%**
NewPipe	11	8	99	73	84	10	99	91	95
FAST	31	28	99	90	94	30	99	97	98
Simplenote	19	16	99	84	91	18	99	95	97

Conclusion. This study proposes resource dependences to complete the slicing. We observed improvements in the quality of slices and have evaluated for 3 apps. On average, the accuracy is 90% for AndroidSlicer and 96% for SfR. This indicates that the data flow between statements and application resources impacts the accuracy of slicing tools for Android applications. We intend to build the tool supporting the parent-child tag element in the future.

CHAPTER 6

TOWARDS AUTOMATED DETECTION OF UNETHICAL BEHAVIOR IN OPEN-SOURCE SOFTWARE PROJECTS

We expand our work in the realm of automated analysis of Android applications to encompass the detection of unethical behavior in both Android and open-source software (OSS) projects. With inspiration from stakeholders in the OSS community, our third contribution introduces Etor, an innovative approach that utilizes automated analysis techniques to identify unethical behavior within OSS projects, including Android projects. Through the development of Etor, we strive to make a valuable contribution to the field of ethical software development and to foster responsible practices within the OSS community, aiming to assist developers in creating more reliable software or apps.

Past studies focused on specific ethical issues (e.g., gender bias and fairness in OSS). There is little to no study on the different types of unethical behavior in OSS projects. We present the first study of unethical behavior in OSS projects from the stakeholders' perspective. Our study of 320 GitHub issues provides a taxonomy of 15 types of unethical behavior guided by six ethical principles (e.g., autonomy). Examples of new unethical behavior include *soft forking* (copying a repository without forking) and *self-promotion*

(promoting a repository without self-identifying as contributor to the repository). We also identify 18 types of software artifacts affected by the unethical behavior. The diverse types of unethical behavior identified in our study (1) call for attentions of developers and researchers when making contributions in GitHub, and (2) point to future research on automated detection of unethical behavior in OSS projects.

Based on our study, we propose Etor, an approach that can automatically detect six types of unethical behavior by using ontological engineering and Semantic Web Rule Language (SWRL) rules to model GitHub attributes and software artifacts. Our evaluation on 195,621 GitHub issues (1,765 GitHub repositories) shows that Etor can automatically detect 552 unethical behavior with 80.5% average true positive rate. This shows the feasibility of automated detection of unethical behavior in OSS projects.

6.1 Introduction

The advent of Open-Source Software (OSS) has created a large ecosystem of applications, libraries, and components that are readily available for download. Ethical considerations have become an important topic with the massive growth of OSS development. For example, the recent incident where researchers from the University of Minnesota attempted to check the feasibility of stealthily introducing vulnerabilities in OSS by making *hypocrite commits* (commits that intentionally introduces critical bug into code), has provoked active discussion among the Linux kernel community, researchers, and other OSS developers [177]. The Linux developers argued that the “hypocrite commits” experiment is “not ethical” and is wasting developers’ time reviewing the invalid patches [5]. More importantly, this incident has revealed an attack on the basic premise of OSS itself (i.e., the fact that anyone can contribute to the code and any OSS project is susceptible to a similar incident). Indeed, unethical behavior committed by contributors of OSS project might lead to broken trust between the OSS community and the contributor,

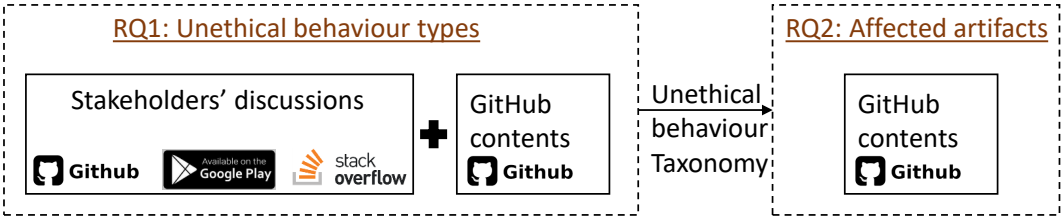


Figure 6.1: Overview of our empirical study on unethical behavior.

whereas unethical software development might lead to loss of funding, reputation, or other resources of the OSS organization involved.

Prior studies discussed ethical issues faced by *stakeholders* (individuals taking part or interested in the OSS project, and can either affect or be affected by the OSS project, software, or application) in OSS projects [138] [71]. Meanwhile, Gold and Krinke highlighted several ethical considerations when mining OSS repositories [66]. These studies stress the importance of considering ethical issues in OSS projects by using various examples and referring to several ethical principles. Unfortunately, prior study suggests that explicitly instructing students or professional developers to consider the ACM code of ethics has no significant impact on their ethical decision-making in software engineering tasks [128]. A similar argument has been made in AI ethics, which calls for *practical methods to translate principles into practice [132].*

To better understand the effects of unethical behavior in OSS projects, the major research direction mainly focuses on studying the effects of gender bias in OSS [91, 159], fairness of the code review process [64], and checking for software licensing [111] [166]. Although it is important to study these common types of unethical behavior, it is essential to thoroughly investigate the types of unethical behavior from stakeholders' perspectives to bridge the gaps between general ethical principles and OSS practices. However, to the best of our knowledge, there is no comprehensive empirical investigation into the *diverse types of unethical behavior*, their *characteristics*, and the corresponding *ethical principles* that drive these unethical behavior in OSS projects.

To the best of our knowledge, we present the first comprehensive taxonomy of the unethical behavior found in OSS projects. We crawled through GitHub issues/pull requests (PRs), and identified 320 issues that discuss about unethical behavior. Our study identifies 15 types of unethical behavior, including (1) S1: No attribution to the author in code, (2) S2: Soft forking, (3) S3: Plagiarism, (4) S4: License incompatibility, (5) S5: No license provided in public repository, (6) S6: Uninformed License change, (7) S7: Depending on proprietary software, (8) S8: Self-promotion, (9) S9: Unmaintained project with paid service, (10) S10: Vulnerable code/API, (11) S11: Naming confusion, (12) S12: Closing issue/PR without explanation, (13) S13: Offensive language, (14) S14: No opt-in or no option allowed, and (15) S15: Privacy Violation. The unethical behavior is affecting 18 types of software artifacts.

Inspired by our study, we developed an automatic detection tool, called Ethic detector (Etor) based on *ontological engineering* (a description of entities and their properties, relationships, and behaviors) and Semantic Web Rule Language rules (SWRL rules) to model GitHub attributes, and software artifacts. In summary, we made the following contributions:

- To the best of our knowledge, we conducted the first study of the types of unethical behavior in OSS projects. Our study of 320 GitHub issues/PRs from 305 repositories reveals that: (1) there are 15 types of unethical behavior in OSS projects, and (2) these unethical behavior affects 18 different types of software artifacts. The diverse types of unethical behavior in OSS calls for attention from the developers and project owners. Identifying the artifacts helps in modeling these behaviors, bringing us one step closer to designing a tool that can automatically detect the violations of these issues.
- We propose Etor, a novel ontology-based approach that automatically detects unethical behavior in OSS projects. Our approach models GitHub attributes using

ontologies, and we design SWRL rules that can check unethical behavior affecting diverse types of software artifacts.

- Our evaluation on 195,621 GitHub issues/ PRs from 1,765 repos shows that Etor can automatically detect 552 issues with 80.5% true positive rate on average. The dataset and the source code for Etor are publicly available at GitHub[1].

6.2 Background and Related work

Prior studies on ethical principles in OSS projects mainly focus on six aspects: (1) accountability, (2) attribution, (3) autonomy, (4) informed consent, (5) privacy, and (6) trust [162] [62] [42]. **Accountability** means that an individual is accountable for his/her actions. **Attribution** (i.e., intellectual property, copyright, etc.) means giving credit to authors when the credit is due. **Autonomy** allows an individual to decide, plan, and act to achieve their goals. In OSS projects, individuals inherently have autonomy because they can choose which tasks to perform but may gain or lose autonomy once they agree to participate. **Informed Consent** is an agreement between the individual and the institution maintaining ethical values such as autonomy, and trust. **Privacy** is a right or entitlement of a stakeholder on what information another stakeholder can obtain and communicate to others. **Trust** refers to expectations between people through goodwill.

Web Ontology Language (OWL) is a standard ontology language endorsed by the W3C to construct an OWL knowledge model [6] [126] [28]. It is a semantic web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. OWL is a computational logic-based language such that knowledge expressed in OWL can be exploited by computer programs, e.g., to verify the consistency of that knowledge or to make implicit knowledge explicit. Thus, we design our tool based on ontology engineering.

[1]https://github.com/EtorChecker/Etor

Semantic Web Rule Language (SWRL) is a language that combines OWL and Rule Markup Language (RuleML), which can be used to express Horn-like rules and logic [7]. SWRL rules are used to infer new knowledge regarding the individual (instance) by chains of properties. We choose to model the unethical behavior in OSS projects using SWRL because (1) its expressiveness [167] is well-suited for modeling unethical behavior that involves different GitHub attributes and diverse types of software artifacts, and (2) it has been widely used to model concepts such as privacy for medical data [40] and access control policy [38, 97]. SWRL rule has the form:

$$\texttt{antecedent} \rightarrow \texttt{consequent}$$

where both `antecedent` and `consequent` are conjunctions of atoms written as $a_1 \wedge \cdots \wedge a_n$. Variables in SWRL rules are denoted by prefixing them with a question mark (e.g., `?x`). Below is an example showing the syntax:

$$\texttt{Person(?p)} \wedge \texttt{hasAge(?p,?age)} \wedge$$
$$\texttt{swrlb:greaterThan(?age,17)} \rightarrow \texttt{Adult(?p)}$$

This rule states that if there is a person `p` whose age is greater than seventeen, then this person is an adult. In the example, `swrlb:greaterThan()` is a widely used built-in function supported in SWRL to increase its expressiveness.

Related Work. Prior studies focus on multiple aspects of ethical concerns for several domains.

Studies on ethical concerns in Software Engineering. Several studies focus on ethical concerns for empirical studies in software engineering [26, 32, 152]. Badampudi conducted a study about the reports of the ethical considerations in Software Engineering publications [32]. Andrews et al. illustrated some of the common approaches to encourage ethical behavior and their limits for demanding ethical behavior between researchers' duty and their publishing as well as the companies' and individuals' integrity [26]. Singer et al. introduced their work as a practical guide to ethical research involving humans in

software engineering [152]. Our study is complementary to these studies as the types of unethical behavior discovered in our study points to potential violations of ethical principles that software engineering researchers should consider when their evaluations of automated tools use OSS projects.

Studies on ethical concerns in OSS. Existing studies of OSS projects focus on issues related to gender bias [91, 159], fairness of the code review process [64], similar code in Stack Overflow and GitHub [35, 181], and software licensing [111] [166] [165]. Studies relating to gender bias in GitHub [91, 159] aims to address the obstacles in improving gender diversity. Meanwhile, a study of a large industrial open source ecosystem (OpenStack) shows that unfairness is "starting to be perceived as an issue" in OSS [64]. Several studies investigated code clones between code snippets from Stack Overflow and projects on GitHub and found a considerable number of non-trivial clones [35, 181]. Although these studies also explored how GitHub stakeholder's reference code was copied or adapted from Stack Overflow answers without giving proper credits to the authors (who wrote the code), they did not consider the scenario where the stakeholder of the code snippets used in GitHub is the same as the owner of the code in Stack Overflow (in this case, a credit is not needed). Several techniques have been proposed for the automated detection of license incompatibility [63, 95, 179]. Although our study identifies license incompatibility as one of the types of unethical behavior, our study includes more diverse types of unethical issues related to licensing (missing license, license incompatibility, and uninformed license change). Nevertheless, all existing studies on ethical concerns in OSS projects only focus on a few aspects of ethical principles, and they did not conduct an analysis of the diverse types of ethical violations in OSS projects in GitHub.

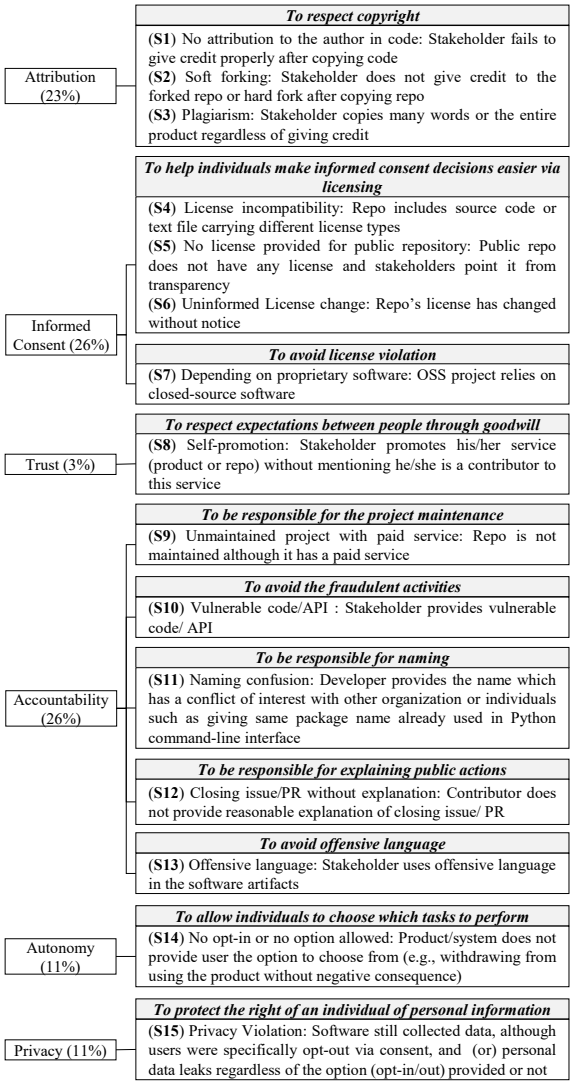


Figure 6.2: Taxonomy of unethical behavior in OSS projects.

6.3 Study of unethical behavior

To understand different types of unethical behavior in OSS projects, we conducted a study guided by common principles from prior work [26] [102] [130] [61] [68]. We manually inspect stakeholders' discussions on GitHub issues to identify a common set of unethical behavior or characteristics seen in OSS projects in GitHub. We crawled issues in GitHub by searching using the keyword "ethic", and its concepts related to ethics, and synonyms to "un/ethical" (i.e., "unprofessional", "unfair", "right", "proper", and "principle"). We

Table 6.1: Number of issues and affected artifacts of unethical behavior in OSS projects

Type	**Issues (#)**	**Affected artifacts**
S1	49	42 Source code, 6 Configuration files, 1 API
S2	20	20 Projects
S3	6	2 Source code, 2 Data, 1 UI, 1 Project
S4	27	10 Legalese, 7 Source code, 4 `README`/ `CONTRIBUTING.md`, 3 Configuration files, 1 Image, 1 OS, 1 Website
S5	31	31 Legalese
S6	9	9 `CHANGELOG`s
S7	16	16 APIs
S8	8	8 PR/Issue comments
S9	11	11 Release histories
S10	26	22 Source code, 4 APIs
S11	21	10 Product names, 8 Source code, 1 UI, 1 Data, 1 Script
S12	16	10 PR/Issue comments, 6 PR/Issue reviews
S13	8	3 UIs, 2 Product names, 1 Source code, 1 `README`/ `CONTRIBUTING.md`, 1 Website
S14	36	15 UIs, 11 Software features, 6 Source code, 4 Configuration files
S15	36	12 Source code, 10 APIs, 5 UIs, 5 Software features, 3 Configuration files, 1 Website
Total	**320**	**320**

manually (1) analyzed the discussion by stakeholders if it included unethical behavior in the collected data, (2) filtered out invalid issues (e.g., issues that mention "ethic"' but only involve updating terms and conditions in document[2]), and (3) further analyzed subsequent links if we found additional links in the discussion (GitHub links, such as issue, PRs, repositories, Google Play link, or Stack Overflow link), resulting in 75

[2]https://github.com/Pryaxis/handbook/issues/3

additional links. The final output of our crawler is a list of GitHub issues/PRs that discuss unethical behavior in OSS projects. We started from 1642 GitHub issues/PRs of 1339 projects submitted by stakeholders. After filtering invalid issues, we obtained a total of 320 issues related to unethical behavior. For each issue, we attempted to answer the following two questions:

RQ1: What are the common types of unethical behavior in OSS projects? What are the ethical principles and ethical guidelines that these unethical behavior violates?

RQ2: For each of these unethical issues, what types of software artifacts are affected?

RQ1 aims to explore the types of unethical behavior in OSS projects in GitHub, whereas RQ2 investigates the impacted software artifacts to model them for automated detection. Figure 6.1 shows an overview of our study. When categorizing an issue, we used an open coding approach [112], a widely-used approach for qualitative research. Two authors of the work independently analyze the issues by reading their discussion, and meeting to discuss to resolve any disagreement.

Ethical considerations. Although most empirical studies in Software Engineering include a survey that asks developers for their opinions on the studied topics, we choose to observe unethical behavior passively by reading developers' discussions to avoid spamming developers [34].

6.3.1 RQ1: Types of unethical behavior

We use a three-step approach to identify the types of unethical behavior in OSS projects. First, we identified six common ethical principles guiding the action of stakeholders in OSS inspired by prior work [102] (we exclude "Welfare" because it concerns fair wages and job security which is generally not discussed in OSS projects). The six ethical principles include Attribution, Informed Consent, Trust, Accountability, Autonomy, and

Privacy. Second, we derived ethical guidelines by exploring terms used in existing studies [71] [64] [162] [62] [42] [130] [101] [75] [8] [137] [39] [148] [96] [57] [143] [135] [129] [171][52] [133] [49] [56] [120] [117] [107] [53]. This results in 11 ethical guidelines. Third, we classified unethical behavior by reviewing previous literature on ethical guidelines and reading comments on GitHub issues. Figure 6.2 shows the 15 types of unethical behavior in our study. The boxes on the left (e.g., "Attribution") describes the ethical principles behind each unethical behavior, whereas the grey heading for the boxes on the right (e.g., "To respect copyright") includes the 11 ethical guidelines for the principles, and the contents present the related types of unethical behavior. We explain the 11 ethical guidelines and the corresponding type of unethical behavior found in our study below:

1. *To respect copyright.* We define an issue to be related to copyright if it contains "copy", "plagiarism", "credit", or "fork". There are three types of unethical behavior related to copyright, described below:

We classify an unethical behavior issue as **S1: No attribution to the author in code** if the stakeholders failed to credit properly after copying a piece of code because all derived work must be credited. An example discussion for **S1** is *"... copied from stack overflow ... we can ignore the license if we want but it seems a bit **unethical** to just ignore it."*[3] Likewise, if the copied item is a repository and has not given credit to the forked repository, we classify it as **S2: Soft forking** because GitHub encourages forking as the core of social coding and crediting the original developers [137]. An example discussion for **S2** is *(S2) "Unauthorised copy of... **unethical**... You must delete this repo and fork from the original..."*[4]

We consider an issue as **S3: Plagiarism** if the stakeholders were involved in copying texts (non-source code) or the entire product regardless of giving credit or not [8]. For

[3]https://github.com/OpenTreeMap/otm-core/issues/400

[4]https://github.com/biddyweb/yes-cart/issues/33

example, replicating the website's text and using it as the content of the interactive book, which is the majority of the product. An example discussion for **S3** is *"Interactive book should be free of plagiarism. By replicating the content used by...**unethical**."*[5]

2. *To help individuals make informed consent decisions easier via licensing.* For each GitHub issue/PR, we define the issue to be related to licensing if the issue contains "license". Specifically, we classify an issue as **S4: License incompatibility** if the repository included source code or text files carrying different license types compared to the project's license because stakeholders must ensure license compatibility of the repository. Example for **S4** is *"To continue distributing when we know they have incompatible licenses is **unethical**."*[6].

If the public repository does not have any license and the stakeholders request it, we classify an issue as **S5: No license provided in public repository** because licenses state the official permissions to use a repository, and project owners should provide the license if the OSS is public for greater transparency. An example comment for **S5** is *"The repository is public which implies an intent of being open-source but no license is specified making review of the code an issue...People get...at the end of the day, but they are funding this stuff instead of the... developers. That's **unethical** but legal."*[7].

In terms of transparency concerns, developers of OSS projects should inform the stakeholders about the license change (via `CHANGELOG` or PR) prior to changing the license. We categorize this unethical behavior as **S6: Uninformed License change** if the contributors fail to do so. An example comment for **S6** is *"Sudden, Silent License change?...I find this silent change **unethical**."*[8]

3. *To avoid license violation*. For each GitHub issue/PR, we define the issue to be related to license violations if the issue contains "proprietary", and "closed source" because

[5]https://github.com/CircuitVerse/Interactive-Book/issues/80
[6]https://github.com/mpdf/mpdf/issues/15
[7]https://github.com/pkalogiros/AudioMass/issues/1
[8]https://github.com/minio/minio/issues/12143

stakeholders must obey the OSS license agreement and avoid integrating prohibited licenses that cause violations in license dependency chains [96]. Specifically, we classify an issue as **S7: Depending on proprietary software** if the OSS projects rely on closed-source software. An example comment for **S7** is *"Since ... is fully open source software, I believe depending on closed source software is* ***unethical****"*[9].

4. *To respect expectations between people through goodwill*. Trust is an ethical principle that refers to respecting expectations between people through goodwill. One of the unethical behavior that we studied could lead to broken trust between stakeholders in OSS projects. Specifically, we use the keywords "promote", "advertise" and "promotion" to identify **S8: Self-promotion** (i.e., the stakeholder advertises his or her service, such as product and repository, without mentioning that he or she is a contributor to this service). An example comment for **S8** is *"Seeing him leverage his notability and following to promote and increase the adoption of ..., which he just released a few days ago, is* ***unethical****."*[10]

5. *To be responsible for the project maintenance.* For each GitHub issue/PR, we define the issue to be related to project maintenance if the issue contains "maintain", "support", "pay", and "purchase". We consider an issue to be **S9: Unmaintained project with paid service** if the project repository is not actively maintained when it has a paid service. It is unethical because one of the responsibilities of the project's owner is to listen to the users who escalated the bugs, provide support and fix the bugs within a reasonable time. An example comment for **S9** is *"I just bought the pro version, and now I'm having this same problem...definitely* ***unethical****."*[11]

6. *To avoid fraudulent activities.* We identify an issue to be related to fraudulent activities if it contains the keywords "malware", "vulnerable", and "hacking". The issue is classified as **S10: Vulnerable code/API** if it describes stakeholders involving in

[9] https://github.com/wger-project/wger/issues/266
[10] https://github.com/eslint/eslint/pull/15102
[11] https://github.com/tranleduy2000/javaide/issues/236

malicious activities (e.g., contributing malicious code or API). An example comment for **S10** is *"Given that ... 2.1.7 has...unfixed security vulnerability, ...continuing to release it is **unethical**"*[12].

7. *To be responsible for naming.* We define an issue to be related to naming if it contains the keyword "name", and we classify it as **S11: Naming confusion** if it involves the stakeholders' duty to give unique names for their artifacts (e.g., packages, variables, and libraries). Project owners are responsible for identifying unique names before using the name. An example comment for **S11** is *"research on the package names before assigning to the package. There is already a package 'click' for creating command-line interfaces. I am using ... package which turns out that it tries to import `click` package:... your library does not have a style component and python throws an error...**unethical**"*[13]. In this example, developers select the same package name as the `Click` package, causing a program failure due to naming conflicts.

8. *To be responsible for explaining public actions.* We consider **S12: Closing issue/PR without explanation**" to be an unethical behavior if the issue/PR is being closed without providing any explanation because all stakeholders need to receive reasonable explanations to support informational fairness [49]. An example comment for **S12** is *"...It's a bit **unfair** to just close something without explaining why?..."*[14].

9. *To avoid offensive language.* We define an issue to be related to offensive language if it contains the keyword "offensive", and we classify it as **S13: Offensive language** if it involves the stakeholders using offensive language because words with subjects that are considered offensive language might represent unethical behavior [52]. Prior study states that hate speech (offensive words) might not necessarily be a criminal offense but it can still harm [133]. An example comment for **S13** is *"Rename the Scroll of Genocide*

[12]https://github.com/flyingsaucerproject/flyingsaucer/pull/123
[13]https://github.com/click-llc/click-integration-django/issues/1
[14]https://github.com/twbs/bootstrap/issues/5632

*to something else...It was never **ethical** name"*[15]. In this example, the stakeholder thinks that using inappropriate words (e.g., "Genocide") to name a scroll in software is unethical.

10. *To allow individuals to choose which tasks to perform.* We define the issue to be related to **S14: No opt-in or no option allowed** if it has the keywords "opt-in", and "option". Specifically, the system does not provide users options such as withdrawing from using the product and disallowing the permission to save personal data without negative consequences. For example, no option is available for uninstalling the third-party library. We focus on issues with "no option" or "no opt-in" because they provide stronger protections than opt-out [39]. An example comment for **S14** is *"There should be an option if someone wants to completely remove ... from the system...I think it's **unethical** to not provide an easy way for a program to be uninstalled"*[16].

11. *To protect the right of an individual of personal information*. We define an issue to be related to privacy if it contains the keyword "privacy", and we classify it as **S15: Privacy Violation** under two common scenarios: (1) if the software still collects data despite the fact that users have opted-out via consent, and (2) if there exist personal data leaks regardless of the option (opt-in/out) being provided or not. An example comment for **S15** is *"Form submitted even if opt-in checkbox is unchecked...Signing people up when they haven't opted in is a major enough bug...at least **unethical**"*[17].

Table 6.1 presents the numbers of issues we found for each type of unethical behavior. The "Type" and "Issues (#)" columns represent the types of unethical behavior issue and the number of issues we found in GitHub, respectively. Overall, our study identifies 15 types of unethical behavior where *the most common types of unethical behavior are related to copyright (**S1**, **S2**, and **S3**) and licensing (**S4**, **S5**, and **S6**)*. As our study shows that illegal copying of code (**S1**) or copying the entire repository (**S2**), or copying texts (**S3**) are common in OSS projects, we hope to raise awareness to stakeholders of OSS

[15]https://github.com/NetHack/NetHack/issues/359

[16]https://github.com/EasyEngine/easyengine/issues/488

[17]https://github.com/katzwebservices/Contact-Form-7-Newsletter/issues/79

projects that such behavior is considered unethical.

6.3.2 RQ2: Affected software artifacts

Software artifacts are objects made intentionally to achieve some purposes [81]. In our work, we consider all artifacts appearing in software repositories as software artifacts. Specifically, we define *affected software artifacts* as artifacts that violate ethical principles. Based on prior studies [90, 142], we identify 18 types of artifacts, including (1) source code, (2) legalese (i.e., licenses, copyright notes or patents), (3) application programming interface (API), (4) user interface (UI), (5) project, (6) release history, (7) software feature (i.e., functional or non-functional requirements of a system), (8) product name (i.e., the affected artifact concerns the product, project, or app name), (9) configuration file, (10) PR/Issue code review, (11) PR/Issue comment, (12) README / CONTRIBUTING.md, (13) CHANGELOG, (14) data (i.e., data from database systems like SQLite), (15) image, (16) operating system (OS), (17) website, and (18) script (i.e., source code in languages executed by an interpreter). As some types of the artifacts are more difficult to understand, we define these artifacts in the following paragraph:

Source code: If the API is internal, like method within the own package, it should be source code

Legalese: Legalese refers to licenses, copyright notes, or patents. (e.g., LICENSE, LICENCE COPYRIGHT, or PATENTS).

API: API is about third party (external) library or service.

Software feature: Software feature refers to an observable system behavior (feature) [59, 89]. An example feature is the ability to unsubscribe from newsletters.

The third column in Table 6.1 presents the affected artifacts for each unethical behavior. Each number in the column represents the number of GitHub issues with a particular type of artifact (e.g., “42 Source code” means that there are 42 issues where **S1**

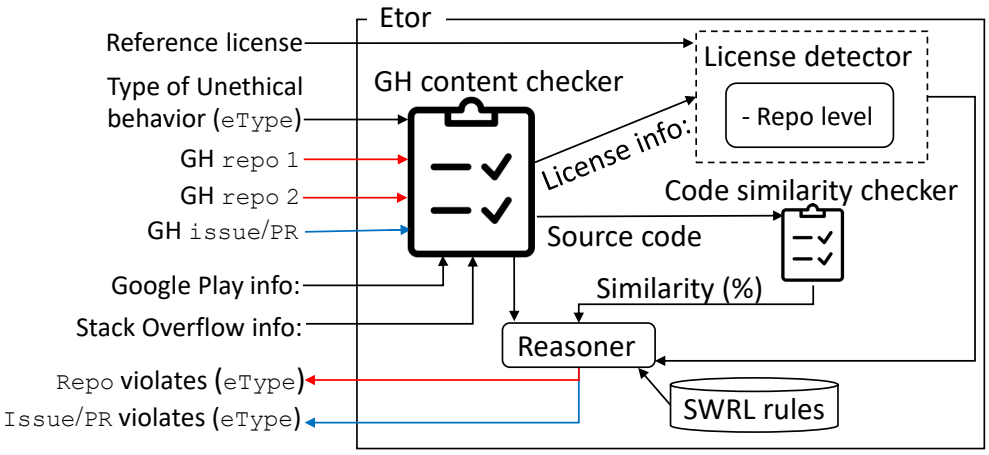


Figure 6.3: Overall architecture of Etor (GH denotes GitHub).

is affected by source code). Theoretically, one GitHub issue might contain the discussion for multiple artifacts but during our analysis, each issue only discusses about one artifact because we notice that developers prefer to discuss ethical concerns for a particular type of artifact in a separate GitHub issue. Overall, Table 6.1 shows that *source code is still the most common type of artifacts for unethical behavior in OSS projects* (i.e., it affects eight types of unethical behavior). However, apart from applying program analysis techniques for detecting unethical behavior involving source code, *future research needs to work on designing a robust tool that can parse and analyze 18 types of artifacts for automated detection of unethical behavior* discovered in our study.

6.4 Methodology

Our study shows that diverse types of unethical behavior exist in OSS projects, and these unethical behaviors usually involve many different types of software artifacts. The diversity and the complexity of the rules governing the ethics-related activities in GitHub motivate the need for a modeling approach that can abstract this complexity and facilitate its automatic detection. In this section, we first describe how we model unethical

Table 6.2: GitHub attributes and types for auto-detection

Attribute	**Type**	**Description**
GHRepository		
licenseFile	GHContent	repo's license file
readmeFile	GHContent	readme file
fileCount	int	# of files in repo
fileContent	GHContent	file's content
commitCountByPath	int	# of commits for specific file path
commitByPath	GHCommit	commit for file path
fork	GHRepository	fork of a repo
forkCount	int	# of forks of repo
contributor	GHUser	stakeholder taking part in GitHub repo
pullRequestCountByCommit	int	# of PRs which contain specific commit
latestRelease	GHRelease	the last release in GitHub history
GHUser		
user	String	GitHub username
GHIssue		
issueMessageBody	String	description of issue
issueOwner	GHUser	stakeholder who reports issue
GHCommit		
commitCodeChange	String	code change in commit
GHContent		
contentCount	int	# of contents stored in file's content
content	String	content
path	String	file path
pathCount	int	# of file paths
GHRelease		
publishedDate	Date	date of release in GitHub history

behavior in OSS projects using SWRL rules in Section 6.4.1. Then, in Section 6.4.2, we explain the overall design and architecture of Etor that incorporates the SWRL rules for its automatic detection of unethical behavior.

6.4.1 Modeling unethical behavior via SWRL rules

As there are a large number of software artifacts that are stored in repositories in GitHub [142], we propose using SWRL rules to represent unethical behavior in an OSS project together with the data stored in the project. SWRL rules allow us to model software artifacts affected by unethical behavior as hierarchies of classes and properties to represent the relationships between the affected software artifacts and stakeholders. Table 6.2 shows the list of GitHub attributes that we used for our modeling. The columns under "Attribute", and "Type" explain each attribute and its types (e.g., $latestRelease$ attribute of `GHRepository` has the `GHRelease` type).

For each OSS project, we model it as `GHRepository`. By referring to the GitHub Repositories API [18], we extract 11 data properties (e.g., `latestRelease` and `licenseFile`) that belong to a `GHRepository` by excluding properties that are irrelevant for modeling unethical behavior (e.g., `avatar_url` that points to the icon for a repository). Apart from the `GHRepository` main class, we have derived the following classes to model data properties of a repository:

GHUser: A GitHub user identified via its username. While GitHub users usually play different roles in an OSS project, we only model two kinds of users: (1) contributors (users who are official contributors of a repository) and (2) issue owners (users who report an issue).

GHCommit: The code changes in a commit.

GHContent: The content (including source code) of a file and its location (file path).

[18] https://docs.github.com/en/rest/repos

GHIssue: A GitHub issue that describes a bug or a feature. We reuse the same convention in GitHub Search API by modeling a PR (`GHPullRequest`) as a subclass of `GHIssue` (i.e., GitHub Issue Search API will search for issues and pull requests, essentially treating a PR as a type of GitHub issue).

GHRelease: The latest releases are represented by the published date of the release.

6.4.2 Automatic detection of unethical behavior

Among 15 types, we identify six types of unethical behavior that can be automatically detected. Specifically, we exclude nine unethical behavior because (1) they involve artifacts (e.g., product names, software features, and data files) that are difficult to automatically isolate from other information (we exclude "Plagiarism", "Naming confusion" and "Offensive language"), (2) they require sophisticated analysis of configuration files, API or source code (we exclude "License incompatibility", "Depending on proprietary software", "Vulnerable code/API", "Privacy Violation", and "No opt-in or no option allowed"), and (3) their detection requires advanced natural language processing (we exclude Closing issue/PR without explanation as it is difficult to automatically judge whether the explanation for the decision to close PR/issue exists), and (3) automated approaches for "License incompatibility" [63, 95, 179] exist so we exclude it to avoid reinventing the wheels.

Overview of Etor. Figure 6.3 presents the overall architecture of our automatic detection tool, Etor. Our approach supports detection of unethical behavior for two levels, including: (1) repository (denoted as `repo`), and (2) GitHub issue/pull request (we will denote an issue as `issue` and a pull request as `PR`). Given a `repo` or an `issue`/`PR`, and the type of unethical behavior `eType` to be checked, the Etor relies on its set of SWRL rules for its detection, and produces as output whether there is a violation of `eType` in the given input. Apart from GitHub attributes listed in Table 6.2 that can be detected using the

GitHub API, our SWRL rule reasoner uses two additional components for its detection: (1) license detector that checks for licenses at the repository level, and (2) code similarity checker that identifies similar code.

Auto-detectable issues. Below are the six types of issues that can be detected automatically:

(S1) No attribution to the author in code. This type checks if an issue or a PR has a Stack Overflow link representing a reference code, and the code snippet copied from Stack Overflow cites the reference link. Given an `issue/PR` as input, Etor checks if a comment `b` in the `issue/PR` posted by a stakeholder `u1` contains the Stack Overflow link (`w`) (we use regular expression to extract `w`). Etor reports a potential violation if: (1) `u1` is not the owner of the Stack Overflow comment, (2) the code snippets from Stack Overflow is found in one of the files in the repository (`F`) with at least 10% similarity (copyright law permits the use of up to 10% of work without permission [19]), and (3) `w` is not found in `F`. Note that we only check for Stack Overflow links in Etor because we learned from our study that contributors are required to give credit to authors for the copied code as code snippets in Stack Overflows are protected by the CC-BY-SA Creative Commons license. The SWRL rule for **S1** is listed below:

$$\texttt{GHIssue(?i)} \wedge \texttt{isIssueMessageBody(?i,?b)} \wedge \texttt{swrlb:contains(?b,"https://stackoverflow.com/")} \wedge \texttt{isWebLink(?b,?w)} \wedge \texttt{isIssueOwner(?i,?u1)} \wedge \texttt{openStream(?w,?s)} \wedge \texttt{swrlb:booleanNot(contains(?s,?u1))} \wedge \texttt{isFileCount(?r,?fc)} \wedge \texttt{swrlb:greaterThan(?fc,0)} \wedge \texttt{isFileContent(?r,?ffct)} \wedge \texttt{isContentCount(?ffct,?ffctc)} \wedge \texttt{swrlb:greaterThan(?ffctc,0)} \wedge \texttt{isContent(?ffct,?ffcc)} \wedge \texttt{isSimilarCode(?ffcc,?s)} \wedge \texttt{swrlb:booleanNot(contains(?ffcc,?w))} \wedge \rightarrow \texttt{S1(?i)}$$

(S2) Soft forking. Given two repositories `r1` and `r2`, Etor compares the contents of all

[19] https://www.legislation.gov.au/Details/C2017C00180

source files in the two repositories to check if one repository is a *soft-fork* (the repository has the same content but it is not listed as an official fork of another repository) of another repository. Specifically, we use `AC2`[20] to detect the similarities between files. AC2 is a source code plagiarism detection tool that has been widely used by instructors and graders to detect plagiarism within a group of assignments. We select AC2 for checking code similarity because (1) it supports many programming languages (e.g., C, C++, Java, and PHP), (2) it can be run in a local environment without needing to send data to remote servers, (3) it includes information visualization which makes it easy to visualize the detection results, and (4) it is quite robust as it incorporates multiple algorithms found in the scientific literature. Etor reports a violation if it detects: (1) 100% similarity between `r1` and `r2`, and (2) `r2` is not in the fork list of `r1`. The SWRL rule for **S2** is listed below:

```
GHRepository(?r1)∧GHRepository(?r2)∧swrlb:isSimilarCode(?r1,?r2)∧
isForkCount(?r1,?fc)∧swrlb:greaterThan(?fc,0)∧isFork(?r2,null)→S2(?r2)
```

(S5) No license provided in public repository. Given a repository `r`, Etor detects the repo-level license by checking if it exists in the: (1) `LICENSE` file[21] in the main directory of `r`, (we check only in the main directory to avoid mistakenly finding API license or package license) or (2) `README.md` file with license information (we use the list of licenses provided by GitHub [22] for repo-level license detection). Etor reports a potential violation if no license is found after searching for the two files. The SWRL rule for **S5** is listed below:

```
GHRepository(?r)∧isFileCount(?r,?fc)∧swrlb:greaterThan(?fc,0)∧
isLicenseFile(?r,null)∧isReadmeFile(?r,?rmf)∧isContentCount(?rmf,?rmcc)
```

[20] https://github.com/manuel-freire/ac2

[21] https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/adding-a-license-to-a-repository

[22] https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository

∧swrlb:greaterThan(?rmcc,0)∧isContent(?rmf,?c)∧

swrlb:booleanNot(contains(?c,[License]))→S5(?r)

(S6) Uninformed License change. We consider a change to be *uninformed* if (1) it is not being announced in the CHANGELOG.md or (2) the license change is done via PR. Given a repository r, Etor checks if the repo-level license has been changed by: (1) extracting commit lists of the license file, and (2) checking if commit changes include license updates. If the license changes occur in more than one commit (we ignore the first commit as it is the initial license creation), Etor checks whether the changes have been announced in the CHANGELOG.md by checking whether the CHANGELOG.md mentions license information. If no license information is found in CHANGELOG.md, Etor checks the PR count for the commit (pullRequestCountByCommit). If the count is less than one, Etor marks it as a potential violation. The SWRL rule for **S6** is listed below:

GHRepository(?r)∧isFileCount(?r,?fc)∧swrlb:greaterThan(?fc,1)∧

isLicenseFile(?r,?lf)∧isContentCount(?lf,?fctc)∧swrlb:greaterThan(?fctc,

0)∧isContent(?lf,?c)∧isPathCount(?lf,?pc)∧swrlb:greaterThan(?pc,0)∧

isPath(?lf,?p)∧isCommitCountByPath(?r,?ctp)∧swrlb:greaterThan(?ctp,1)∧

isCommitByPath(?r,?cl)∧isCommitCodeChange(?cl,?cc)∧swrlb:contains(?cc,

[License])∧swrlb:booleanNot(contains(CHANGELOG,?cc))∧

isPullRequestCountByCommit(?r,?prc)∧swrlb:lessThan(?prc,1)→S6(?r)

(S8) Self-promotion. We consider *self-promotion* to be the scenario where a contributor u opens a GitHub issue/PR where the content of the issue/PR includes links to another repository in GitHub (swrlb:contains(?b,"https://github.com/")) to promote his or her own repository. Given an issue/PR for r1 as input, Etor first (1) checks that the issue/PR includes a link L to another repository r2, and (2) identifies the stakeholder u who opens the issue/PR. Then, it reports a violation if: (1) r1 is not r2, (2) u is not a

contributor of r1 (i.e., u is an outsider for r1), and (3) u is a contributor of r2. To reduce false positives, Etor also checks if L includes specific keywords ("\issues\", "\pull\", "\commit\", "\tree\", "\releases\", "\blob\", and "\runs\"). These keywords usually indicate that the contributor is sharing the link L for demonstration purposes ([DEMO]) instead of promoting a repository/library. The SWRL rule for **S8** is listed below:

GHIssue(?i)∧isIssueMessageBody(?i,?b)∧swrlb:contains(?b, "https://github.com/")∧isWebLink(?b,?r2)∧swrlb:notEqual(?r1,?r2)∧ isIssueOwner(?i,?u)∧swrlb:booleanNot(isContributor(?r1,?u))∧ swrlb:booleanNot(contains(?b,[DEMO]))∧isContributor(?r2,?u)→S8(?i)

(S9) Unmaintained Android Project with Paid Service. This type checks whether an Android project offered paid service in Google Play, but stop actively maintaining the GitHub repository. On average, 115 APIs are updated per month [125], and 49% of app updates have at least one update within 47 days [127]. Based on the frequency of app updates, we define an *unmaintained Android project* to be an Android project where the latest update released date is less than 0.5 year. Given a r as input, Etor first checks for unmaintained Android projects by examining whether (1) the latest release date (D) of r is less than 0.5 year, and (2) r is an original repository (i.e., not forked from other repositories). Then, it checks whether the app offers a paid service by (1) identifying the Google Play link l from r, and (2) searching for the "in-app purchase". The SWRL rule for **S9** is listed below:

GHRepository(?r)∧isLatestRelease(?r,?lr)∧isPublishedDate(?lr,?d)∧ durationGreaterThan(0.5,?d,"Years")∧isFork(?r,null)∧isFileCount(?r,?fc)∧ swrlb:greaterThan(?fc,0)∧isReadmeFile(?r,?rm)∧isContentCount(?rm,?rmcc) ∧swrlb:greaterThan(?rmcc,0)∧isContent(?rm,?c)∧swrlb:contains(?c, "https://play.google.com")∧isWebLink(?c,?l)∧openStream(?l,?s)∧ swrlb:contains(?s,"in-apppurchase")→S9(?r)

Table 6.3: Number of issues detected and TP/FP rate

Type	# Unethical Issues	True Positive		False Positive		Time (s)
	# repos or issues / Total	# repos or issues / Total	%	# repos or issues / Total	%	
(S1) No attribution to the author in code	80 / 195,621 issues	59 / 80 issues	74	21 / 80 issues	26	5.4
(S2) Soft forking	10 / 100 repos	10 / 10 repos	**100**	0 / 10 repos	**0**	343.1
(S5) No license provided in public repository	476 / 1,765 repos	426 / 476 repos	89	50 / 476 repos	11	3.1
(S6) Uninformed License change	18 / 1,765 repos	16 / 18 repos	88	2 / 18 repos	11	9.2
(S8) Self-promotion	116 / 195,621 issues	37 / 116 issues	32	79 / 116 issues	68	4.3
(S9) Unmaintained Android Project with Paid Service	4 / 1,765 repos	4 / 4 repos	**100**	0 / 4 repos	**0**	5.3
Average	-	-	**80.5**	-	**19.3**	-

6.5 Evaluation

We applied Etor on 195,621 GitHub issues and PRs of 1,765 GitHub repos to address the following research questions:

RQ3: How many unethical issues can Etor find in OSS projects?

RQ4: What is the effectiveness of Etor in detecting unethical behavior?

By counting the number of unethical issues in OSS projects, RQ3 provides a rough estimation of the prevalence of each type of unethical behavior in OSS projects. For RQ4, we measure the effectiveness of Etor by checking for the accuracy and the efficiency of its detection using the following metrics:

True Positive (TP): Etor classifies an unethical behavior as a potential violation, and it is a true violation.

False Positive (FP): Etor incorrectly classifies an unethical behavior as a potential violation, and it is a false violation.

Time: The average time taken (in seconds) to detect a type of unethical behavior across all the evaluated repositories/issues.

Selection of projects/issues. As there is no prior benchmark for evaluating the detection of unethical behavior, we construct a dataset by crawling GitHub. Our goal is to select the most recent popular (most stars and most forks) OSS projects and the GitHub

issues/PRs from these popular projects for evaluation. We first obtain the list of the top 2,000 OSS projects (we first get the top 1,000 projects with the greatest number of stars, and then the top 1,000 projects with the greatest number of forks) created last year (2021). After eliminating duplicated projects, there are 195,621 GitHub issues/PRs of 1,765 projects in our evaluation set. As soft forking requires two repositories as input, we obtain the pair of repositories ($repo1$,$repo2$) by getting $repo1$ from the top 200 projects (first 100 from most stars, subsequent 100 from most forks) from the initial list of 2,000 projects. From these 200 projects, our crawler automatically identifies $repo2$ by searching GitHub for projects with similar names using the name of $repo1$ as the query. At this step, our crawler found only 10 out of the 200 projects that have at least one repository with similar names. For each of these 10 projects, our crawler retrieves the first 10 repositories from the search results as $repo2$, leading to a total of 10*10=100 projects for evaluating soft forking.

Ethical considerations. As calling out stakeholders for violations of unethical behavior could potentially lead to similar ethical concerns in prior work [177], we choose to evaluate Etor by manually inspecting the reported issues. To reduce author bias in the manual classification of TP/FPs, we ask for help from a non-author to independently label each issue.

All experiments are conducted on a machine with Intel(R) Core (TM) i7-7500 CPU @2.7 GHz and 16 GB RAM. **Implementation.** We use Protégé 5.5.0 [134] [136] to define the knowledge model for our work. Our crawler uses the GitHub API (Java version [23], Python version [24]).

[23]https://github-api.kohsuke.org/

[24]https://github.com/PyGithub/PyGithub

6.5.1 RQ3: Number of detected issues

Table 6.3 summarizes the results of our evaluation. The "Type" column denotes the unethical issue type that Etor detected, whereas the second column is of the form x / y where x represents the number of repositories/issues with the unethical behavior detected and y denotes the total number of repositories/issues in our evaluation dataset. Overall, Etor has successfully detected at least one violation for all types of unethical behavior that we studied. As our evaluation dataset is different from the study dataset, and we have observed the occurrences of unethical behavior in both datasets, this indicates that *different types of unethical behavior is prevalent in OSS projects*. Table 6.3 also shows that "No license provided in public repository" is the most common types among the six types of detected issues. This means that *a relatively high percentage of the evaluated repositories are missing license files* (around 24% of the evaluated repositories if we exclude the false positives). For the issue-level detection, we observe that "No attribution to the author in code" and "Self-promotion" are the most common ones among all evaluated issues/PRs. This means that *contributors of OSS projects tend to (1) forget to give credit to the author in their copied code snippets, or (2) promote their own repositories when contributing code*.

6.5.2 RQ4: Effectiveness of Etor

Accuracy. To evaluate the effectiveness of Etor, two raters (one author, and one non-author who is an undergraduate CS student working as a part-time student assistant) independently inspect its output. Specifically, for each violation reported by Etor, each rater determines if the violation is a true violation (TP) or a false violation (FP). We use Cohen's Kappa to assess inter-rater agreement, specifically for measuring inter-rater reliability scores [106] [105] [25]. The initial Cohen's Kappa was 0.82, which indicates a high level of agreement. The two raters then meet to resolve any disagreement to

reach Cohen's Kappa of 1.0. The "True positive" and "False positive" columns in Table 6.3 show the results for the inspection. On average, the true positive rate is 80.5%, and the false positive rate is 19.3%. For repository-level detection, although Etor can only detect a small number of violations for "Soft forking" and "Unmaintained Android Project with Paid Service", it can detect these unethical issues with high accuracy (0% FP rate). For "Soft forking", as we consider a repository a *soft-fork* only if all the contents of the two repositories are the same (100% similarity), this design decision may lead to fewer violations being found but increase the accuracy of its detection. In future, it is worthwhile to study the effect of the similarity threshold on the accuracy of its detection. For issue-level detection, Etor can detect "No attribution to the author in code" with reasonable accuracy (26% FP rate).

Efficiency. The "Time" column in Table 6.3 shows the average time taken to detect an unethical behavior. Overall, the average time to analyze a repository is 3.1–343.1 seconds and the average time taken to analyze an issue is 4.3–5.4 seconds. This indicates that *Etor can detect a type of unethical behavior relatively fast*. We also observe that "Soft forking" is the most time-consuming type to detect because Etor needs to check for code similarities for all source files within the repository.

Reasons behind the inaccurate detection. We manually inspect the reasons behind the FPs reported by Etor.

Etor reports the highest FP rate for "Self-promotion". Recall that Etor checks that a stakeholder `St` opens an issue/PR `I` at repository `R1`, and includes the other repository (`R2`) link (`L`). A true "Self-promotion" only occurs if `St` did not mention about being a contributor of `R2`. We need to manually verify this condition by reading the comments written in natural language. Hence, FPs may occur if (1) `St` mentioned that he or she is a contributor of `R2` (e.g., "I am working on a project called the ..." in comment [25]) or (2) `St` wanted to ask for suggestion in using `R1` for `R2` (e.g., "I'd like to try your ... module in a

[25] https://github.com/Anarios/return-youtube-dislike/issues/401

non-mmdetection repo (...)" [9]).

For "No attribution to the author in code", we found three main reasons for FPs; (1) there is no code or idea copied (e.g., the Stack Overflow link was mentioned as references [10]), (2) Etor checks the exact Stack Overflow link and is unable to detect if the citation uses the short link of Stack Overflow, and (3) Etor detects the exact GitHub user name against with Stack Overflow user name, and is unable to detect if the user name is different (e.g., GitHub user name is `devinrhode2` and Stack Overflow user name is `DevinRhode` [11]). For "No license provided in public repository", FPs occur because the repository (1) has a license file that is not in the main directory (e.g., `LICENSE` file in the inner folder [12]), (2) has a disclaimer in `README.md` (e.g., "This repository is for personal study and research purposes only. Please DO NOT USE IT FOR COMMERCIAL PURPOSES." in `README.md` [13]), (3) is used for education purposes (we need to manually exclude repositories for the public schools where the license is not required), (4) has no source code or data, and (5) is under an organization license and no separate license is defined for the repository [14].

For "Uninformed License change", FPs occur because the scenario where the repository has restored the old license should not be considered an unethical violation (e.g., the stakeholder changed the license from "Apache License Version 2.0" to "GNU GENERAL PUBLIC LICENSE Version 3" on Feb 17 2022, and he/she restored back to "Apache License Version 2.0" on Feb 18 2022 [15]).

6.6 Discussion and Implications

We discuss practical takeaways and suggestions below:

Types of unethical behavior in OSS projects. Our study revealed several types of unethical behavior that occur in general setting (e.g., "Plagiarism" and "Offensive language"), and several types that are unique in the context of OSS projects (e.g., "Soft

forking" represents ethical concerns when forking, "Closing issue/PR without explanation" are related to closing GitHub issues/PRs). Although Etor can only detect six out of the 15 studied types using an ontology-based approach, *our study points to future direction of research that designs more sophisticated techniques to automatically detect unethical behavior in OSS projects*.

Software artifacts affected by unethical behavior. Our study shows that modeling unethical behavior requires checking for conditions over diverse types of software artifacts. Although source code is still the most common type of artifacts affected by unethical behavior (Table 6.1), other artifacts in natural language (e.g., PR/Issue comments, product names, and website) are also common. In future, *it is worthwhile to study how to apply natural language processing techniques to accurately detect unethical behavior affected by these artifacts*.

Challenges in automated detection of unethical behavior. To provide guidelines for future research on the automated detection of unethical behavior, we discuss several challenges identified in our study and evaluation:

- As shown in our study in Section 6.3.2, the *types of artifacts affected by the unethical behavior are too diverse*. An accurate detection technique needs to support analysis of various types of artifacts, including source code, data, and websites.
- Within GitHub, we notice that *discussion and announcement in GitHub spread across multiple web pages* (issues, PRs, wikis, discussions, and commit logs). With the rapid growth of different types of web pages in GitHub, it poses additional challenges for automated approaches to exhaustively analyze all relevant web pages.
- We notice that *some discussions of unethical behavior are conducted outside of GitHub* (e.g., personal emails, slack channel). For example, for "Self-promotion", we cannot check whether the stakeholder has communicated with the developers in advance through emails. Without complete information about the discussion, the detection is bound to be

inaccurate.

• *The scope for the detection can be too broad for some types of unethical behavior* (e.g., "Naming confusion"). Without a predefined scope of detection (package name collision versus app name collision), we cannot accurately detect the behavior.

• There exist *ambiguities for certain unethical behavior*, which makes it difficult even for human beings to reach consensus (e.g., whether the language used is offensive). In this case, an automated tool can present all relevant information to help stakeholders in OSS to make more grounded decisions about whether a behavior is ethical or not.

6.7 Threats to validity

External. Our findings of unethical behavior may not generalize beyond the studied OSS projects and issues/PRs. While other types of unethical behavior discovered in our study is important, Etor can only detect six of them, and our evaluation is limited to these six types. Nevertheless, our experiments show that Etor can detect unethical behavior with relatively high accuracy, which shows the feasibility of having an approach for automated detection of unethical behavior.

Internal. Our code and scripts may have bugs that can affect our results. To mitigate this threat, we make our tool and data publicly available for inspection.

6.8 Conclusion and future work

To better understand activities in OSS projects that can lead to ethical concerns, we conduct a study of the types of unethical behavior in OSS projects. By reading and analyzing the discussion of stakeholders in OSS projects, our study of 320 GitHub issues identifies 15 types of unethical behavior. These unethical behaviors are affected by various types of software artifacts. Inspired by our study, we propose Etor, an ontology-

based approach that can automatically detect unethical behavior. Our evaluation of Etor on 195,621 GitHub issues (1,765 GitHub repositories) shows that Etor can automatically detect 552 issues with 19.3% FP rate on average. As the first study that investigates the types of unethical behavior in OSS projects, we hope to raise awareness of the importance of understanding ethical issues in OSS projects. Our tool that uses software artifacts and data available in GitHub API also lays the foundation for future approaches on automated detection of unethical behavior. In the future, we plan to enhance Etor to detect more issue types, and reduce false positives using machine learning techniques.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this dissertation, we propose the following two approaches for automated debugging for Android apps.

1. **Event-Aware Precise Dynamic Slicing for Automatic Debugging of Android Applications**. 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- and event-level by leveraging the simplified input event sequence that triggers the same bug using segment-based delta debugging. Our tool (ESDroid) can produce a more precise but smaller dynamic PDG with up to 72% (27% on average) fewer falsely 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 intend to enhance ESDroid to handle non-crashing bugs by exercising more strategies (e.g., hierarchical delta debugging), and including test cases with complex interactions such as GUI text input and system

events.

2. **Complement of Dynamic Slicing for Android Applications with Def-Use Analysis for Application Resources**. We propose a novel approach called SfR (Slicing for Resources), which identifies the dependences between the program statements and the application resources to complete the slice for Android applications. We observed improvements in the slices' quality and evaluated for 3 apps. On average, the accuracy is 90% for AndroidSlicer and 96% for SfR. This shows that the data flow between statements and application resources influences the accuracy of slicing tools for Android applications. We plan to build the tool supporting the parent-child tag element in the future.

Inspired by automated analysis, as our third contribution, we study unethical behavior in OSS projects and propose the following approach.

- **Towards Automated Detection of Unethical Behavior in Open-source Software Projects**. We, for the first time, conduct a study of the types of unethical behavior in OSS projects and propose a novel approach that can automatically detect unethical behavior. By reading and analyzing the discussion of stakeholders in OSS projects, our study of 320 GitHub issues identifies 15 types of unethical behavior. These unethical behaviors are affected by various types of software artifacts. Inspired by our study, we implemented a tool called Etor (an ontology-based approach) that can automatically detect unethical behavior. Our evaluation of Etor on 195,621 issues (1,765 repositories) reveals that Etor can automatically detect 552 issues with 80.5% TP rate on average. As the first study that investigates the types of unethical behavior in OSS projects, we hope to raise awareness among OSS stakeholders regarding the importance of understanding ethical issues in OSS projects. While Etor indicates promising results in automated detection of

unethical behavior in OSS projects, we intend to enhance Etor in the future to detect more types and reduce false positives using machine learning techniques.

CHAPTER 8

ETHICAL ISSUES

I have read HREC Guidelines for undergraduate and postgraduate students carefully. According to this guidance, this research does not need HREC approval. The theory, model, algorithms and experiments are not involved human. The experimental datasets are from open source platforms.

Bibliography

[1] *Research by Cambridge MBAs for tech firm Undo finds software bugs cost the industry $316 billion a year*. https://www.jbs.cam.ac.uk/insight/2013/research-by-cambridge-mbas-for-tech-firm-undo-finds-software-bugs-cost-the-industry-316-billion-a-year/. Accessed: 16/03/2023.

[2] *Android vs. Apple Market Share: Leading Mobile Operating Systems (OS) (Mar 2023)*. https://www.bankmycell.com/blog/android-vs-apple-market-share/. Accessed: 05/05/2021.

[3] *Android*. https://source.android.com/. Accessed: 16/03/2023.

[4] *GooglePlay*. https://play.google.com/store/apps. Accessed: 01/07/2021.

[5] *Report on University of Minnesota Breach-of-Trust Incident*. https://lwn.net/ml/linux-kernel/202105051005.49BFABCE@keescook/. Accessed: 05/05/2021.

[6] *OWL Web Ontology Language*. https://www.w3.org/2001/sw/#owl. Accessed: 05/05/2022.

[7] *SWRL*. http://www.w3.org/Submission/SWRL/. Accessed: 05/05/2021.

[8] *What is Plagiarism?*. https://www.plagiarism.org/article/what-is-plagiarism. Accessed: 16/03/2023.

[9] *CUDA vs Naive Speedup?*. https://github.com/d-li14/involution/issues/1. Accessed: 12/03/2021.

[10] *Squeeze tooltip in the sections panel*. https://github.com/livebook-dev/livebook/pull/536. Accessed: 02/09/2021.

[11] *Are we correctly handling console.Console in node objectKeys(console)?*. https://github.com/sindresorhus/ts-extras/issues/50. Accessed: 16/03/2023.

[12] *ailab*. https://github.com/bilibili/ailab. Accessed: 21/02/2022.

[13] *VIP*. https://github.com/Oreomeow/VIP. Accessed: 21/02/2022.

[14] *DogeBot2*. https://github.com/DGXeon/DogeBot2. Accessed: 21/02/2022.

[15] *xmm*. https://github.com/heiyeluren/xmm. Accessed: 21/02/2022.

[16] R. ABREU, P. ZOETEWEIJ, R. GOLSTEIJN, AND A. J. VAN GEMUND, *A practical evaluation of spectrum-based fault localization*, Journal of Systems and Software, 82 (2009), pp. 1780–1792.

[17] R. ABREU, P. ZOETEWEIJ, AND A. J. VAN GEMUND, *On the accuracy of spectrum-based fault localization*, in Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007), IEEE, 2007, pp. 89–98.

[18] H. AGRAWAL, R. A. DEMILLO, AND E. H. SPAFFORD, *Dynamic slicing in the presence of unconstrained pointers*, in Proceedings of the symposium on Testing, Analysis, and Verification, 1991, pp. 60–73.

[19] H. AGRAWAL AND J. R. HORGAN, *Dynamic program slicing*, ACM SIGPlan Notices, 25 (1990), pp. 246–256.

[20] H. AGRAWAL, J. R. HORGAN, S. LONDON, AND W. E. WONG, *Fault localization using execution slices and dataflow tests*, in Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95, IEEE, pp. 143–151.

[21] M. AHMADZADEH, D. ELLIMAN, AND C. HIGGINS, *An analysis of patterns of debugging among novice computer science students*, in Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, 2005, pp. 84–88.

[22] K. AHMED, M. LIS, AND J. RUBIN, *Mandoline: Dynamic slicing of android applications with trace-based alias analysis*, in 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, 2021, pp. 105–115.

[23] A. ALAVI, I. NEAMTIU, AND R. GUPTA, *Dynamic slicing for android*, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), (2019), pp. 1154–1164.

[24] E. ALVES, M. GLIGORIC, V. JAGANNATH, AND M. D'AMORIM, *Fault-localization using dynamic slicing and change impact analysis*, in 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), IEEE, 2011, pp. 520–523.

[25] A. M. ANDRÉS AND P. F. MARZO, *Delta: A new measure of agreement between two raters*, British journal of mathematical and statistical psychology, 57 (2004), pp. 1–19.

[26] A. A. ANDREWS AND A. S. J. E. S. E. PRADHAN, *Ethical issues in empirical software engineering: the limits of policy*, 6 (2001), pp. 105–110.

[27] A. ANG, A. PEREZ, A. VAN DEURSEN, AND R. ABREU, *Revisiting the practical use of automated software fault localization techniques*, in 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2017, pp. 175–182.

[28] G. ANTONIOU AND F. V. HARMELEN, *Web ontology language: Owl*, in Handbook on ontologies, Springer, 2004, pp. 67–92.

[29] C. ARTHO, *Iterative delta debugging*, International Journal on Software Tools for Technology Transfer, 13 (2011), pp. 223–246.

[30] P. ARUMUGA NAINAR AND B. LIBLIT, *Adaptive bug isolation*, in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, 2010, pp. 255–264.

[31] S. ARZT, S. RASTHOFER, C. FRITZ, E. BODDEN, A. BARTEL, J. KLEIN, Y. LE TRAON, D. OCTEAU, AND P. MCDANIEL, *Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps*, Acm Sigplan Notices, 49 (2014), pp. 259–269.

[32] D. BADAMPUDI, *Reporting ethics considerations in software engineering publications*, in 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, pp. 205–210.

[33] T. BALL, M. NAIK, AND S. K. RAJAMANI, *From symptom to cause: localizing errors in counterexample traces*, in Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2003, pp. 97–105.

[34] S. BALTES AND S. DIEHL, *Worse than spam: Issues in sampling software developers*, in Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement, 2016, pp. 1–6.

[35] S. BALTES AND S. DIEHL, *Usage and attribution of stack overflow code snippets in github projects*, Empirical Software Engineering, 24 (2019), pp. 1259–1295.

[36] A. BANERJEE, L. K. CHONG, C. BALLABRIGA, AND A. ROYCHOUDHURY, *Energypatch: Repairing resource leaks to improve energy-efficiency of android apps*, IEEE Transactions on Software Engineering, 44 (2017), pp. 470–490.

[37] BANKMYCELL, *Android vs. apple market share: Leading mobile operating systems (os) (jul 2023)*. `https://www.bankmycell.com/blog/android-vs-apple-market-share/`, 2023. Accessed: 2023-07-15.

[38] D. BEIMEL AND M. PELEG, *Using owl and swrl to represent and reason with situation-based access control policies*, Data & Knowledge Engineering, 70 (2011), pp. 596–615.

[39] S. R. BERGERSON, *E-commerce privacy and the black hole of cyberspace*, Wm. Mitchell L. Rev., 27 (2000), p. 1527.

[40] H. BOUSSI RAHMOUNI, T. SOLOMONIDES, M. CASASSA MONT, AND S. SHIU, *Modelling and enforcing privacy for medical data disclosure across europe*, in Medical Informatics in a United and Healthy Europe, IOS Press, 2009, pp. 695–699.

[41] R. BRUMMAYER AND A. BIERE, *Fuzzing and delta-debugging smt solvers*, in Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, 2009, pp. 1–5.

[42] M. CENITE, B. H. DETENBER, A. W. KOH, A. L. LIM, N. E. J. N. M. SOON, AND SOCIETY, *Doing the right thing online: a survey of bloggers' ethical beliefs and practices*, 11 (2009), pp. 575–597.

[43] M. CHEN, K. GOEL, N. S. SOHONI, F. POMS, K. FATAHALIAN, AND C. RÉ, *Mandoline: Model evaluation under distribution shift*, in International Conference on Machine Learning, PMLR, 2021, pp. 1617–1629.

[44] T. M. CHILIMBI, B. LIBLIT, K. MEHRA, A. V. NORI, AND K. VASWANI, *Holmes: Effective statistical debugging via efficient path profiling*, in 2009 IEEE 31st International Conference on Software Engineering, IEEE, pp. 34–44.

[45] J.-D. CHOI AND J. FERRANTE, *Static slicing in the presence of goto statements*, ACM Transactions on Programming Languages and Systems (TOPLAS), 16 (1994), pp. 1097–1113.

[46] S. R. CHOUDHARY, A. GORLA, AND A. ORSO, *Automated test input generation for android: Are we there yet?(e)*, in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 429–440.

[47] L. CLAPP, O. BASTANI, S. ANAND, AND A. AIKEN, *Minimizing gui event traces*, in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 422–434.

[48] H. CLEVE AND A. ZELLER, *Finding failure causes through automated testing*, arXiv preprint cs/0012009, (2000).

[49] J. A. COLQUITT, *On the dimensionality of organizational justice: a construct validation of a measure.*, Journal of applied psychology, 86 (2001), p. 386.

[50] W. J. CONOVER, *Practical nonparametric statistics*, vol. 350, john wiley & sons, 1999.

[51] M. L. CORLISS, E. C. LEWIS, AND A. ROTH, *Low-overhead interactive debugging via dynamic instrumentation with dise*, in 11th International Symposium on High-Performance Computer Architecture, IEEE, 2005, pp. 303–314.

[52] D. A. DA SILVA, H. D. B. LOURO, G. S. GONCALVES, J. C. MARQUES, L. A. V. DIAS, A. M. DA CUNHA, AND P. M. TASINAFFO, *Could a conversational ai identify offensive language?*, Information, 12 (2021), p. 418.

[53] L. DABBISH, C. STUART, J. TSAY, AND J. HERBSLEB, *Social coding in github: transparency and collaboration in an open software repository*, in Proceedings of the ACM 2012 conference on computer supported cooperative work, 2012, pp. 1277–1286.

[54] R. A. DEMILLO, H. PAN, AND E. H. SPAFFORD, *Critical slicing for software fault localization*, ACM SIGSOFT Software Engineering Notes, 21 (1996), pp. 121–134.

[55] M. DILHARA, H. CAI, AND J. JENKINS, *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, 2018, pp. 67–71.

[56] N. DOORN, D. SCHUURBIERS, I. VAN DE POEL, AND M. E. GORMAN, *Early engagement and new technologies: Opening up the laboratory*, vol. 16, Springer, 2014.

[57] R. DUAN, A. BIJLANI, M. XU, T. KIM, AND W. LEE, *Identifying open-source license violation and 1-day security risk at large scale*, in Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security, 2017, pp. 2169–2185.

[58] E. DUESTERWALD, R. GUPTA, AND M. L. SOFFA, *Distributed slicing and partial re-execution for distributed programs*, Lecture Notes In Computer Science, 757 (1993), pp. 497–511.

[59] T. EISENBARTH, R. KOSCHKE, AND D. SIMON, *Locating features in source code*, IEEE Transactions on software engineering, 29 (2003), pp. 210–224.

[60] J. FERRANTE, K. J. OTTENSTEIN, AND J. D. WARREN, *The program dependence graph and its use in optimization*, ACM Transactions on Programming Languages and Systems (TOPLAS), 9 (1987), pp. 319–349.

[61] C. FLICK, *Informed consent in information technology: Improving end user licence agreements*, Professionalism in the information and communication technology industry, (2013), p. 127.

[62] B. FRIEDMAN, P. H. KAHN, A. BORNING, AND A. HULDTGREN, *Value sensitive design and information systems*, Springer, 2013, pp. 55–95.

[63] D. M. GERMAN, Y. MANABE, AND K. INOUE, *A sentence-matching method for automatic license identification of source code files*, in Proceedings of the IEEE/ACM international conference on Automated software engineering, 2010, pp. 437–446.

[64] D. M. GERMAN, G. ROBLES, G. POO-CAAMAÑO, X. YANG, H. IIDA, AND K. INOUE, *"was my contribution fairly reviewed?" a framework to study the perception*

of fairness in modern code reviews, in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 523–534.

[65] C. GIBLER, J. CRUSSELL, J. ERICKSON, AND H. CHEN, *Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale*, in Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5, Springer, 2012, pp. 291–307.

[66] N. E. GOLD AND J. KRINKE, *Ethical mining: A case study on msr mining challenges*, in Proceedings of the 17th International Conference on Mining Software Repositories, pp. 265–276.

[67] L. GOMEZ, I. NEAMTIU, T. AZIM, AND T. MILLSTEIN, *Reran: Timing-and touch-sensitive record and replay for android*, in 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 72–81.

[68] M. C. GRACE, W. ZHOU, X. JIANG, AND A.-R. SADEGHI, *Unsafe exposure analysis of mobile in-app advertisements*, in Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks, 2012, pp. 101–112.

[69] A. GROCE, M. A. ALIPOUR, C. ZHANG, Y. CHEN, AND J. REGEHR, *Cause reduction for quick testing*, in 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, IEEE, 2014, pp. 243–252.

[70] A. GROCE, D. KROENING, AND F. LERDA, *Understanding counterexamples with explain*, in Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16, Springer, 2004, pp. 453–456.

[71] F. S. GRODZINSKY, K. MILLER, AND M. J. WOLF, *Ethical issues in open source software*, Journal of Information, Communication and Ethics in Society, (2003).

[72] N. GUPTA, H. HE, X. ZHANG, AND R. GUPTA, *Locating faulty code using failure-inducing chops*, in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 2005, pp. 263–272.

[73] R. GUPTA, M. J. HARROLD, AND M. L. SOFFA, *An approach to regression testing using slicing.*, in ICSM, vol. 92, 1992, pp. 299–308.

[74] R. GUPTA, M. L. SOFFA, AND J. HOWARD, *Hybrid slicing: Integrating dynamic information with static analysis*, ACM Transactions on Software Engineering and Methodology (TOSEM), 6 (1997), pp. 370–397.

[75] F. HABIBZADEH AND K. SHASHOK, *Plagiarism in scientific writing: words or ideas?*, Croatian Medical Journal, 52 (2011), p. 576.

[76] B. HAILPERN AND P. SANTHANAM, *Software debugging, testing, and verification*, IBM Systems Journal, 41 (2002), pp. 4–12.

[77] M. HAMMOUDI, B. BURG, G. BAE, AND G. ROTHERMEL, *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, 2015, pp. 333–344.

[78] D. HAO, L. ZHANG, L. ZHANG, J. SUN, AND H. MEI, *Vida: Visual interactive debugging*, in 2009 IEEE 31st International Conference on Software Engineering, IEEE, 2009, pp. 583–586.

[79] M. J. HARROLD AND M. L. SOFFA, *Interprocedual data flow testing*, ACM SIGSOFT software engineering notes, 14 (1989), pp. 158–167.

[80] R. HILDEBRANDT AND A. ZELLER, *Simplifying failure-inducing input*, in Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, 2000, pp. 135–145.

[81] R. HILPINEN, *On artifacts and works of art 1*, Theoria, 58 (1992), pp. 58–82.

[82] R. HODOVÁN AND Á. KISS, *Modernizing hierarchical delta debugging*, in Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, 2016, pp. 31–37.

[83] R. HODOVÁN AND A. KISS, *Practical improvements to the minimizing delta debugging algorithm.*, in ICSOFT-EA, 2016, pp. 241–248.

[84] R. HODOVÁN, Á. KISS, AND T. GYIMÓTHY, *Coarse hierarchical delta debugging*, in 2017 IEEE international conference on software maintenance and evolution (ICSME), IEEE, 2017, pp. 194–203.

[85] R. HODOVÁN, Á. KISS, AND T. GYIMÓTHY, *Tree preprocessing and test outcome caching for efficient hierarchical delta debugging*, in 2017 IEEE/ACM 12th

International Workshop on Automation of Software Testing (AST), IEEE, 2017, pp. 23–29.

[86] J. HOFFMANN, M. USSATH, T. HOLZ, AND M. SPREITZENBARTH, *Slicing droids: program slicing for smali code*, in Proceedings of the 28th Annual ACM Symposium on Applied Computing, 2013, pp. 1844–1851.

[87] S. HORWITZ, J. PRINS, AND T. REPS, *Integrating noninterfering versions of programs*, ACM Transactions on Programming Languages and Systems (TOPLAS), 11 (1989), pp. 345–387.

[88] S. HORWITZ, T. REPS, AND D. BINKLEY, *Interprocedural slicing using dependence graphs*, in Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, 1988, pp. 35–46.

[89] I. HSI AND C. POTTS, *Studying the evolution and enhancement of software features.*, in icsm, 2000, p. 143.

[90] S. F. HUQ, A. Z. SADIQ, AND K. SAKIB, *Understanding the effect of developer sentiment on fix-inducing changes: An exploratory study on github pull requests*, in 2019 26th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2019, pp. 514–521.

[91] N. IMTIAZ, J. MIDDLETON, J. CHAKRABORTY, N. ROBSON, G. BAI, AND E. MURPHY-HILL, *Investigating the effects of gender bias on github*, in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp. 700–711.

[92] B. JIANG, Y. WU, T. LI, AND W. K. CHAN, *Simplydroid: Efficient event sequence simplification for android application*, in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 297–307.

[93] J. A. JONES AND M. J. HARROLD, *Empirical evaluation of the tarantula automatic fault-localization technique*, in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 2005, pp. 273–282.

[94] J. A. JONES, M. J. HARROLD, AND J. STASKO, *Visualization of test information to assist fault localization*, in Proceedings of the 24th international conference on Software engineering, 2002, pp. 467–477.

[95] G. M. KAPITSAKI, F. KRAMER, AND N. D. TSELIKAS, *Automating the license compatibility process in open source software with spdx*, Journal of systems and software, 131 (2017), pp. 386–401.

[96] G. M. KAPITSAKI, N. D. TSELIKAS, AND I. E. FOUKARAKIS, *An insight into license tools for open source software systems*, Journal of Systems and Software, 102 (2015), pp. 72–87.

[97] A. KAYES, W. RAHAYU, T. DILLON, AND E. CHANG, *Accessing data from multiple sources through context-aware access control*, in 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), IEEE, 2018, pp. 551–559.

[98] B. W. KERNIGHAN AND P. PLAUGER, *Programming style*, in Proceedings of the fourth SIGCSE technical symposium on Computer science education, 1974, pp. 90–96.

[99] A. KISS, *Generalizing the split factor of the minimizing delta debugging algorithm*, IEEE Access, 8 (2020), pp. 219837–219846.

[100] D. KOB AND F. WOTAWA, *Introducing alias information into model-based debugging*, in ECAI, vol. 16, Citeseer, 2004, p. 833.

[101] D. KOCSIS, *Exploring Intention to Return to a Crowdsourcing Platform Through Ethical Considerations*, PhD thesis, University of Nebraska at Omaha, 2018.

[102] D. KOCSIS AND G.-J. DE VREEDE, *Towards a taxonomy of ethical considerations in crowdsourcing*, (2016).

[103] P. KONG, L. LI, J. GAO, T. F. BISSYANDÉ, AND J. KLEIN, *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, 2019, pp. 78–89.

[104] B. KOREL AND J. LASKI, *Dynamic program slicing*, Information processing letters, 29 (1988), pp. 155–163.

[105] T. O. KVALSETH, *A coefficient of agreement for nominal scales: An asymmetric version of kappa*, Educational and psychological measurement, 51 (1991), pp. 95–101.

[106] X.-B. D. LE, L. BAO, D. LO, X. XIA, S. LI, AND C. PASAREANU, *On reliability of patch correctness assessment*, in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 524–535.

[107] S. B. LEACOCK, *Self-promotion in the age of electronic media*, ETHICS IN PSYCHOLOGY AND THE MENTAL HEALTH PROFESSIONS, p. 349.

[108] Y. LEI AND J. H. ANDREWS, *Minimization of randomized unit test cases*, in 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05), IEEE, 2005, pp. 10–pp.

[109] A. LEITNER, M. ORIOL, A. ZELLER, I. CIUPA, AND B. MEYER, *Efficient unit test case minimization*, in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, pp. 417–420.

[110] R. LENCEVICIUS, U. HÖLZLE, AND A. K. SINGH, *Dynamic query-based debugging of object-oriented programs*, Automated Software Engineering, 10 (2003), pp. 39–74.

[111] J. LERNER AND J. TIROLE, *The scope of open source licensing*, Journal of Law, Economics, and Organization, 21 (2005), pp. 20–56.

[112] S. LEWIS, *Qualitative inquiry and research design: Choosing among five approaches*, Health promotion practice, 16 (2015), pp. 473–475.

[113] X. LI, W. LI, Y. ZHANG, AND L. ZHANG, *Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization*, in Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 169–180.

[114] X. LI AND A. ORSO, *More accurate dynamic slicing for better supporting software debugging*, in 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), IEEE, 2020, pp. 28–38.

[115] B. LIBLIT, *Cooperative bug isolation: winning thesis of the 2005 ACM doctoral dissertation competition*, vol. 4440, Springer, 2007.

[116] C. LIU, L. FEI, X. YAN, J. HAN, AND S. P. MIDKIFF, *Statistical debugging: A hypothesis testing-based approach*, IEEE Transactions on software engineering, 32 (2006), pp. 831–848.

[117] C. LIU, J. T. MARCHEWKA, J. LU, AND C.-S. YU, *Beyond concern: a privacy–trust–behavioral intention model of electronic commerce*, Information & Management, 42 (2004), pp. 127–142.

[118] C. LIU, X. YAN, L. FEI, J. HAN, AND S. P. MIDKIFF, *Sober: statistical model-based bug localization*, ACM SIGSOFT Software Engineering Notes, 30 (2005), pp. 286–295.

[119] J. LIU, T. WU, J. YAN, AND J. ZHANG, *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, 2016, pp. 342–352.

[120] P. LUARN AND H.-H. LIN, *A customer loyalty model for e-service context.*, J. Electron. Commer. Res., 4 (2003), pp. 156–167.

[121] P. MACHADO, J. CAMPOS, AND R. ABREU, *Mzoltar: automatic debugging of android applications*, in Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile, ACM, 2013, pp. 9–16.

[122] X. MAO, Y. LEI, Z. DAI, Y. QI, AND C. WANG, *Slice-based statistical fault localization*, Journal of Systems and Software, 89 (2014), pp. 51–62.

[123] J. MARAS, J. CARLSON, AND I. CRNKOVIĆ, *Client-side web application slicing*, in 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), IEEE, pp. 504–507.

[124] A. MARGINEAN, J. BADER, S. CHANDRA, M. HARMAN, Y. JIA, K. MAO, A. MOLS, AND A. SCOTT, *Sapfix: Automated end-to-end repair at scale*, in Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, IEEE Press, 2019, pp. 269–278.

[125] T. MCDONNELL, B. RAY, AND M. KIM, *An empirical study of api stability and adoption in the android ecosystem*, in 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 70–79.

[126] D. L. MCGUINNESS, F. VAN HARMELEN, ET AL., *Owl web ontology language overview*, W3C recommendation, 10 (2004), p. 2004.

[127] S. MCILROY, N. ALI, AND A. E. HASSAN, *Fresh apps: an empirical study of frequently-updated mobile apps in the google play store*, Empirical Software Engineering, 21 (2016), pp. 1346–1370.

[128] A. MCNAMARA, J. SMITH, AND E. MURPHY-HILL, *Does acm's code of ethics change ethical decision making in software development?*, in Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp. 729–733.

[129] V. MIDHA AND S. SLAUGHTER, *Mitigating the effects of structural complexity on open source software maintenance through accountability*, (2011).

[130] L. I. MILLETT, B. FRIEDMAN, AND E. FELTEN, *Cookies and web browser design: Toward realizing informed consent online*, in Proceedings of the SIGCHI conference on Human factors in computing systems, 2001, pp. 46–52.

[131] G. MISHERGHI AND Z. SU, *Hdd: hierarchical delta debugging*, in Proceedings of the 28th international conference on Software engineering, 2006, pp. 142–151.

[132] B. MITTELSTADT, *Principles alone cannot guarantee ethical ai*, Nature Machine Intelligence, 1 (2019).

[133] M. MONDAL, L. A. SILVA, AND F. BENEVENUTO, *A measurement study of hate speech in social media*, in Proceedings of the 28th ACM conference on hypertext and social media, 2017, pp. 85–94.

[134] M. A. MUSEN, *The protégé project: a look back and a look forward*, AI matters, 1 (2015), pp. 4–12.

[135] H. NISSENBAUM, *Computing and accountability*, in The Ethics of Information Technologies, Routledge, 2020, pp. 351–358.

[136] N. F. NOY, D. L. MCGUINNESS, ET AL., *Ontology development 101: A guide to creating your first ontology*, 2001.

[137] L. NYMAN AND T. MIKKONEN, *To fork or not to fork: Fork motivations in sourceforge projects*, International Journal of Open Source Software and Processes (IJOSSP), 3 (2011), pp. 1–9.

[138] C. OEZBEK ET AL., *Research ethics for studying open source projects*, 4th Research Room FOSDEM: Libre software communities meet research community, (2008).

[139] C. PARNIN AND A. ORSO, *Are automated debugging techniques actually helping programmers?*, in Proceedings of the 2011 international symposium on software testing and analysis, 2011, pp. 199–209.

[140] P. PATEL, G. SRINIVASAN, S. RAHAMAN, AND I. NEAMTIU, *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, 2018, pp. 34–37.

[141] S. PEARSON, J. CAMPOS, R. JUST, G. FRASER, R. ABREU, M. D. ERNST, D. PANG, AND B. KELLER, *Evaluating and improving fault localization*, in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 609–620.

[142] R.-H. PFEIFFER, *What constitutes software? an empirical, descriptive study of artifacts*, in Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 481–491.

[143] S. QIU, D. M. GERMAN, AND K. INOUE, *Empirical study on dependency-related license violation in the javascript package ecosystem*, Journal of Information Processing, 29 (2021), pp. 296–304.

[144] M. RENIERES AND S. P. REISS, *Fault localization with nearest neighbor queries*, in 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings., IEEE, 2003, pp. 30–39.

[145] F. RICCA AND P. TONELLA, *Construction of the system dependence graph for web application slicing*, in Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation, IEEE, 2002, pp. 123–132.

[146] J. ROBLER, G. FRASER, A. ZELLER, AND A. ORSO, *Isolating failure causes through test case generation*, in Proceedings of the 2012 international symposium on software testing and analysis, pp. 309–319.

[147] T. ROEHM, S. NOSOVIC, AND B. BRUEGGE, *Automated extraction of failure reproduction steps from user interaction traces*, in 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 121–130.

[148] S. R. SALBU, *The european union data privacy directive and international relations*, Vand. J. Transnat'l L., 35 (2002), p. 655.

[149] J. SCHUTTE, D. TITZE, AND J. M. DE FUENTES, *Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps*, in 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, 2014, pp. 370–379.

[150] C. SCOTT, V. BRAJKOVIC, G. NECULA, A. KRISHNAMURTHY, AND S. SHENKER, *Minimizing faulty executions of distributed systems*, in 13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16), 2016, pp. 291–309.

[151] G. SHU, B. SUN, A. PODGURSKI, AND F. CAO, *Mfl: Method-level fault localization with causal inference*, in 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, IEEE, 2013, pp. 124–133.

[152] J. SINGER AND N. G. J. I. T. O. S. E. VINSON, *Ethical issues in empirical studies of software engineering*, 28 (2002), pp. 1171–1180.

[153] F. STEIMANN AND M. FRENKEL, *Improving coverage-based localization of multiple faults using algorithms from integer linear programming*, in 2012 IEEE 23rd International Symposium on Software Reliability Engineering, IEEE, 2012, pp. 121–130.

[154] M. STOERZER, B. G. RYDER, X. REN, AND F. TIP, *Finding failure-inducing changes in java programs using change classification*, in Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 57–68.

[155] T. SU, L. FAN, S. CHEN, Y. LIU, L. XU, G. PU, AND Z. SU, *Why my app crashes understanding and benchmarking framework-specific exceptions of android apps*, IEEE Transactions on Software Engineering, (2020).

[156] A. SZEGEDI AND T. GYIMÓTHY, *Dynamic slicing of java bytecode programs*, in Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05), IEEE, 2005, pp. 35–44.

[157] S. TALLAM, C. TIAN, R. GUPTA, AND X. ZHANG, *Enabling tracing of long-running multithreaded programs via dynamic execution reduction*, in Proceedings of

the 2007 international symposium on Software testing and analysis, 2007, pp. 207–218.

[158] S. H. TAN, Z. DONG, X. GAO, AND A. ROYCHOUDHURY, *Repairing crashes in android apps*, Proceedings of the 40th International Conference on Software Engineering, (2018), pp. 187–198.

[159] J. TERRELL, A. KOFINK, J. MIDDLETON, C. RAINEAR, E. R. MURPHY-HILL, AND C. PARNIN, *Gender bias in open source: Pull request acceptance of women versus men.*, PeerJ Prepr., 4 (2016), p. e1733.

[160] D. TITZE AND J. SCHÜTTE, *Apparecium: Revealing data flows in android applications*, in 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, IEEE, 2015, pp. 579–586.

[161] P. TONELLA AND F. RICCA, *Web application slicing in presence of dynamic code generation*, Automated Software Engineering, 12 (2005), pp. 259–288.

[162] M. TURILLI AND L. FLORIDI, *The ethics of information transparency*, Ethics and Information Technology, 11 (2009), pp. 105–112.

[163] R. VALLÉE-RAI, P. CO, E. GAGNON, L. HENDREN, P. LAM, AND V. SUNDARESAN, *Soot: A Java bytecode optimization framework*, 2010, pp. 214–224.

[164] A. VARGHA AND H. D. DELANEY, *A critique and improvement of the cl common language effect size statistics of mcgraw and wong*, Journal of Educational and Behavioral Statistics, 25 (2000), pp. 101–132.

[165] C. VENDOME, M. LINARES-VÁSQUEZ, G. BAVOTA, M. DI PENTA, D. GERMAN, AND D. POSHYVANYK, *License usage and changes: a large-scale study of java projects on github*, in 2015 IEEE 23rd International Conference on Program Comprehension, IEEE, pp. 218–228.

[166] C. VENDOME, M. LINARES-VÁSQUEZ, G. BAVOTA, M. DI PENTA, D. GERMAN, AND D. POSHYVANYK, *Machine learning-based detection of open source license exceptions*, in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 118–129.

[167] D. VRANDEČIĆ, *Ontology evaluation*, in Handbook on ontologies, Springer, 2009, pp. 293–313.

[168] T. WANG AND A. ROYCHOUDHURY, *Using compressed bytecode traces for slicing java programs*, in Proceedings. 26th International Conference on Software Engineering, IEEE, 2004, pp. 512–521.

[169] T. WANG AND A. ROYCHOUDHURY, *Dynamic slicing on java bytecode traces*, ACM Transactions on Programming Languages and Systems, 30 (2008), pp. 1–49.

[170] M. WEISER, *Program slicing*, IEEE Transactions on software engineering, (1984), pp. 352–357.

[171] M. J. WOLF, K. BOWYER, D. GOTTERBARN, AND K. MILLER, *Open source software: intellectual challenges to the status quo*, ACM SIGCSE Bulletin, 34 (2002), pp. 317–318.

[172] E. WONG, T. WEI, Y. QI, AND L. ZHAO, *A crosstab-based statistical method for effective fault localization*, in 2008 1st international conference on software testing, verification, and validation, IEEE, 2008, pp. 42–51.

[173] W. E. WONG, V. DEBROY, AND B. CHOI, *A family of code coverage-based heuristics for effective fault localization*, Journal of Systems and Software, 83 (2010), pp. 188–208.

[174] W. E. WONG, V. DEBROY, Y. LI, AND R. GAO, *Software fault localization using dstar (d*)*, in 2012 IEEE Sixth International Conference on Software Security and Reliability, IEEE, 2012, pp. 21–30.

[175] W. E. WONG, R. GAO, Y. LI, R. ABREU, AND F. WOTAWA, *A survey on software fault localization*, IEEE Transactions on Software Engineering, 42 (2016), pp. 707–740.

[176] F. WOTAWA, M. STUMPTNER, AND W. MAYER, *Model-based debugging or how to diagnose programs automatically*, in Developments in Applied Artificial Intelligence: 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE 2002 Cairns, Australia, June 17–20, 2002 Proceedings, Springer, 2002, pp. 746–757.

[177] Q. WU AND K. LU, *On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits*, in Proc. Oakland, 2021.

[178] B. XU, Z. CHEN, AND H. YANG, *Dynamic slicing object-oriented programs for debugging*, in Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation, IEEE, 2002, pp. 115–122.

[179] S. XU, Y. GAO, L. FAN, Z. LIU, Y. LIU, AND H. JI, *Lidetector: License incompatibility detection for open source software*, ACM Transactions on Software Engineering and Methodology, (2021).

[180] T. XU, *Improving automated program repair with retrospective fault localization*, in 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE, 2019, pp. 159–161.

[181] D. YANG, P. MARTINS, V. SAINI, AND C. LOPES, *Stack overflow in github: any snippets there?*, in 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 280–290.

[182] K. YU, M. LIN, J. CHEN, AND X. ZHANG, *Practical isolation of failure-inducing changes for debugging regression faults*, in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 20–29.

[183] Y. YUAN, L. XU, X. XIAO, A. PODGURSKI, AND H. ZHU, *Rundroid: recovering execution call graphs for android applications*, in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp. 949–953.

[184] A. ZELLER, *Yesterday, my program worked. today, it does not. why?*, in ACM SIGSOFT Software engineering notes, vol. 24, 1999, pp. 253–267.

[185] A. ZELLER, *Why programs fail: a guide to systematic debugging*, Elsevier, 2009.

[186] A. ZELLER AND R. HILDEBRANDT, *Simplifying and isolating failure-inducing input*, IEEE Transactions on Software Engineering, 28 (2002), pp. 183–200.

[187] X. ZENG, D. LI, W. ZHENG, F. XIA, Y. DENG, W. LAM, W. YANG, AND T. XIE, *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, 2016, pp. 987–992.

[188] M. ZHANG AND H. YIN, *Appsealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications.*, in NDSS, 2014.

[189] X. ZHANG, N. GUPTA, AND R. GUPTA, *Pruning dynamic slices with confidence*, Acm Sigplan Notices, 41 (2006), pp. 169–180.

[190] X. ZHANG, R. GUPTA, AND Y. ZHANG, *Precise dynamic slicing algorithms*, in 25th International Conference on Software Engineering, 2003. Proceedings., IEEE, 2003, pp. 319–329.

[191] X. ZHOU, X. PENG, T. XIE, J. SUN, W. LI, C. JI, AND D. DING, *Delta debugging microservice systems*, in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 802–807.