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

Hsu Myat Win
University of Technology Sydney
Australia
HsuMyat.Win@student.uts.edu.au

ABSTRACT

Existing static and dynamic slicing techniques for Android applications exhibit limitations when the location of the fault is in application resources such as layout definitions and user interface strings. This paper proposes 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.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Android, slicing

ACM Reference Format:
Hsu Myat Win. 2022. Complement of Dynamic Slicing for Android Applications with Def-Use Analysis for Application Resources. In *IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MOBILESoft '22), May 17–24, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3524613.3527808

1 PROBLEM AND MOTIVATION

Program slicing [8] 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 [2]. 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.

MOBILESoft '22, May 17–24, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9301-0/22/05…$15.00
https://doi.org/10.1145/3524613.3527808

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 [6]. The dynamic PDG, which is a subgraph of the static PDG [5], 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 [2] [1]. While AndroidSlicer [3] performs dynamic slicing by modeling asynchronous data and control dependences of Android apps, [4] 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.

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) [7]. 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 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 1 shows an example bug [1], which is the wrong text (i.e., `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 (i.e., `Themen`). In Figure 1b, AndroidSlicer generated the slice from the point of interest (i.e., $r8$ holding wrong value `Themen`), and it included Line 4 (i.e., Instruction number 36612) and Line 3 (i.e., Instruction number 36611 and 36610) and missed the location of the fault in application resources (i.e., Line 1 and 2). In Figure 1c, SfR generated the complement with the help of RDG. Specifically, the variable (*tv*) used at Line 3 is defined at Line 2.

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

$$R = \frac{|\{I_c\} \cap \{I_m\}|}{|\{I_c\}|} \qquad P = \frac{|\{I_c\} \cap \{I_m\}|}{|\{I_m\}|} \qquad F\text{-}measure = 2\,\frac{R * P}{R + P}$$

Table 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.

5 ACKNOWLEDGEMENT

I wish to thank Prof. Yulei Sui and Prof. Shin Hwei Tan for their guidance. This work was supported by the National Natural Science Foundation of China (Grant No. 61902170).

REFERENCES

[1] Hiralal Agrawal, Richard A DeMillo, and Eugene H Spafford. 1991. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the symposium on Testing, Analysis, and Verification*. 60–73.
[2] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices* 25, 6 (1990), 246–256.
[3] Tanzirul Azim, Arash Alavi, Iulian Neamtiu, and Rajiv Gupta. 2019. Dynamic slicing for android. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1154–1164.
[4] Mayee Chen, Karan Goel, Nimit S Sohoni, Fait Poms, Kayvon Fatahalian, and Christopher Ré. 2021. Mandoline: Model Evaluation under Distribution Shift. In *International Conference on Machine Learning*. PMLR, 1617–1629.
[5] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[6] Susan Horwitz, Thomas Reps, and David Binkley. 1988. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. 35–46.
[7] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Information processing letters* 29, 3 (1988), 155–163.
[8] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

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