Program Tailoring: Slicing by Sequential Criteria[*]

Yue Li[†1], Tian Tan[†2], Yifei Zhang[3], and Jingling Xue[4]

1 **School of Computer Science and Engineering, UNSW Australia**
`yueli@cse.unsw.edu.au`
2 **School of Computer Science and Engineering, UNSW Australia**
`tiantan@cse.unsw.edu.au`
3 **School of Computer Science and Engineering, UNSW Australia**
`yzhang@cse.unsw.edu.au`
4 **School of Computer Science and Engineering, UNSW Australia**
`jingling@cse.unsw.edu.au`

Abstract

Protocol and typestate analyses often report some sequences of statements ending at a program point P that needs to be scrutinized, since P may be erroneous or imprecisely analyzed. Program slicing focuses only on the behavior at P by computing a slice of the program affecting the values at P. In this paper, we propose to restrict our attention to the subset of that behavior at P affected by one or several statement sequences, called a *sequential criterion (SC)*. By leveraging the ordering information in a *SC*, e.g., the temporal order in a few valid/invalid API method invocation sequences, we introduce a new technique, *program tailoring*, to compute a tailored program that comprises the statements in all possible execution paths passing through at least one sequence in *SC* in the given order. With a prototyping implementation, TAILOR, we show why tailoring is practically useful by conducting two case studies on seven large real-world Java applications. For program debugging and understanding, TAILOR can complement program slicing by removing *SC*-irrelevant statements. For program analysis, TAILOR can enable a pointer analysis, which is unscalable to a program, to perform a more focused and therefore potentially scalable analysis to its *SC*-relevant parts containing hard language features such as reflection.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages - Program Analysis, D.2.5 Testing and Debugging - Code inspections and Debugging aids

Keywords and phrases Program Slicing, Program Analysis, API Protocol Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.15

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.2.1.8`

1 Introduction

Program slicing, supported by industry-strength tools, such as WALA [52] and CodeSurfer [18], has found many diverse applications, such as program debugging, comprehension, analysis, testing, verification and optimization [8, 20, 43, 49]. Given a *slicing criterion* consisting of a program point P and several variables used at P [53], program slicing computes a slice of the program that may affect their values at P in terms of data and control dependences. In the

[*] This work is supported by ARC grants, DP130101970 and DP150102109.
[†] These authors contributed equally to this work.

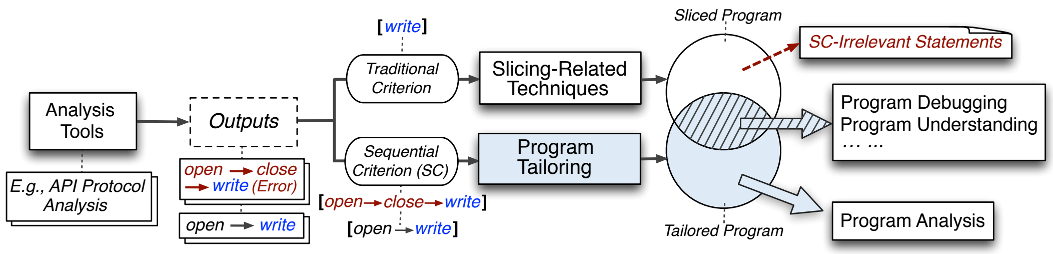


Figure 1 Program tailoring with some of its potential applications highlighted.

past three decades, several variations on this theme of program slicing have been proposed, including static vs. dynamic, backward vs. forward, and closure vs. executable [43,49].

In practice, API protocol analysis [7,37] and typestate analysis [9,16,34] often report some statement sequences ending at a program point P that needs to be scrutinized, since P may be erroneous or imprecisely analyzed. Each sequence can represent a valid or invalid API usage call sequence. Such protocol specifications or violations can also be provided manually or mined automatically [1,3,17,36,42,60]. As such analyses are either conservative or unsound, the temporal order specified in a sequence may or may not be feasible. However, program slicing focuses on P by ignoring this order, which is often essential for analyzing P.

As illustrated in Figure 1, we introduce a new technique, *program tailoring* to reap additional benefits missed by program slicing at a point P (e.g., `file.write()`) by leveraging the temporal order specified in a new criterion, called a *sequential criterion* (*SC*) for P. A SC_P consists of one or several statement sequences ending at P, with each representing, e.g., a valid API call sequence like `file.open()` → `file.write()` or an invalid API call sequence like `file.open()` → `file.close()` → `file.write()`. In what follows, we will drop P from SC_P when the context is clear or when we are not interested in it. Given a SC_P, tailoring aims to obtain a tailored program, denoted $\mathcal{T}(SC_P)$, that comprises the statements in all the possible execution paths passing through at least one statement sequence in SC_P in the given order. By construction, all data and control dependences needed for understanding the behavior at P affected by SC_P are included. Any statement that is not in $\mathcal{T}(SC_P)$ is irrelevant to SC_P, i.e., SC_P*-irrelevant*. For illustration purposes, we write $\mathcal{S}(P)$ to represent the (backward) slice affecting P obtained by program slicing. Note that slicing all the points in SC_P independently still fails to capture their ordering constraint (and is unscalable, too).

Like slicing, tailoring enables software developers or client applications to inspect only the interesting parts of a program. Unlike slicing, which focuses on understanding the program behavior at P, tailoring restricts our attention to the subset of that behavior affected by SC_P only. Due to incompatible criteria used, tailoring can be used either as a complementary technique to slicing or in cases where slicing is ineffective, as discussed below.

1.1 Goals and Motivations

Given a SC_P, we have developed a prototyping implementation of program tailoring, denoted TAILOR, for Java programs, with the following three goals in mind:

Precision TAILOR is designed to sharpen the precision of many client applications, as highlighted in Figure 1, by exploiting the temporal order in a SC_P. One significant class of client applications includes many slicing-related techniques, such as thin slicing [47], program chopping [22] and value slicing [26]. For a given program, $\mathcal{T}(SC_P)$ is shown as the blue circle and $\mathcal{S}(P)$ as the white circle. The statements in $\mathcal{I}(SC_P) = \mathcal{S}(P) - \mathcal{S}(P) \cap \mathcal{T}(SC_P)$ are SC_P-irrelevant and can thus be pruned away to facilitate program debugging and understanding by a human. If $\mathcal{I}(SC_P) = \varnothing$, then Tailor is ineffective at P but no harm is done. If $\mathcal{I}(SC_P) \neq \varnothing$, then Tailor can make slicing more precise, by leveraging the otherwise wasted SC_P information that is widely available. In Section 6.1, we show that Tailor can improve the precision of thin slicing [47], a state-of-the-art practical but unsound slicing technique, for Java programs. In Section 6.2, we show that Tailor can enable a sophisticated pointer analysis for Java, S-2Obj [24], which is unscalable for a program, to perform a focused analysis on its SC-relevant parts containing hard language features such as reflection, where existing slicing techniques are ineffective.

Scalability Tailor is designed to work efficiently for large object-oriented programs, for which traditional slicing [21] is unscalable (even with industry-strength implementations [18, 52]), with the key bottleneck coming from handling of the heap [47]. Like any slicing tool, Tailor is not always scalable. However, Tailor is designed to scale significantly better than traditional slicing, as Tailor avoids handling of the heap by reasoning about essentially the (un)reachability of a statement towards the statement sequences in a SC_P.

Soundness Tailor is designed to be a practical tool to facilitate programming debugging and understanding as well as program analysis (among others) for large object-oriented programs. In this setting, any sound static analysis would be either unscalable or imprecise due to the presence of many hard language features, such as native code, dynamic class loading, reflection and multi-threading [30]. Therefore, Tailor is designed to be sound whenever a sound graph representation is available for capturing all the control flows in a (single- or multi-threaded) program. In other words, Tailor is always sound with respect to part of the program behavior modeled. According to [30], "a soundy analysis aims to be as sound as possible without excessively compromising precision and/or scalability." Therefore, Tailor represents one such soundy analysis.

1.2 Challenges and Solutions

We examine some challenges faced in achieving our three goals and describe our solutions:

Precision/Scalability Tradeoffs How do we compute $\mathcal{T}(SC)$ efficiently and precisely for large object-oriented programs? Due to exponential blowup of program paths, both DFS and BFS are out of question. We formulate the problem of computing $\mathcal{T}(SC)$ by solving an IFDS (Interprocedural Finite Distributive Subset) data-flow analysis problem [38], efficiently on its interprocedural control-flow graph (ICFG) representation of the program. To avoid unrealizable paths with mismatched calls and returns, our analysis must be (fully) context-sensitive in order to achieve useful precision for Java programs. Otherwise, many unrealizable paths, which go through the constructor of `java.lang.Object`, cannot be filtered out, causing $\mathcal{T}(SC)$ to be severely over-approximated. However, distinguishing calling contexts fully with call strings, object-sensitivity [33], and method cloning [54] are known to be unscalable for large programs [12, 44]. We achieve (full) context sensitivity by solving a CFL-reachability problem over a balanced-parentheses language by matching call and return edges also in the IFDS framework as described in [38].
How do we ensure that Tailor works effectively for a SC of any given length, which is defined to be the number of statements in its longest statement sequence? In general, the longer a SC is, the more SC-irrelevant statements will be removed. We propose to lengthen any given SC by leveraging the concept of object-sensitivity [33, 44] developed by the pointer analysis community for Java. As a result, some infeasible paths that would otherwise be introduced are avoided. We will also try to avoid making SC extensions that make our analysis run longer but contribute nothing to precision improvement.

Soundiness How do we make TAILOR as soundly as possible? We decompose the problem of tailoring a program for a given SC into two sub-problems: (1) building an ICFG, G_{ICFG}, for the program and (2) computing $\mathcal{T}(SC)$ from G_{ICFG}. TAILOR is sound if G_{ICFG} is sound (representation of all control flows in the program). In fact, TAILOR is designed to be practically useful for analyzing the program behavior modeled by G_{ICFG} even if G_{ICFG} is unsound. This paper solves (2) while resorting to the state-of-the-art for (1).

1.3 Contributions

- We introduce program tailoring, a new technique for trimming a program based on SCs, which are widely available from other analyses but never exploited by program slicing.
- We describe how to extend a given SC for object-oriented programs in order to improve the precision of program tailoring (by making tailored programs smaller).
- We formulate the problem of computing a tailored program as one of solving a data-flow problem efficiently in the IFDS framework. TAILOR, which is implemented in SOOT [51], is released as an open-source tool at `http://www.cse.unsw.edu.au/~corg/tailor`.
- We describe two case studies to demonstrate why TAILOR is practically useful on a set of seven large real-world Java programs, by (1) assisting program slicing with program debugging and understanding tasks, and (2) enabling a focused pointer analysis on the parts of a program containing hard language features such as reflection.

2 A Motivating Example

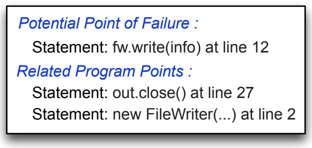
We use an example to describe how TAILOR can assist slicing tools to simplify program debugging and understanding tasks through exploiting the temporal ordering information in a given SC that is otherwise ignored by program slicing. In Section 6.2, we provide additional motivations on why TAILOR can be useful in simplifying program analysis tasks.

Large object-oriented programs are very difficult to debug and understand, due to the pervasive use of heap-allocated data, nested data structures, and large libraries with complex dependences and configurations. Tracing the flow of values via multiple levels of pointer indirection through the heap across many classes in both the application and libraries is unworkable. A practical tool is needed to pinpoint relevant statements for the task at hand.

Our Java example is given in Figure 2. The `Driver` class is used to create and initialize a `Driver` object according to some user input (lines 33 – 36) or by default (lines 37 – 41). Then the corresponding initialization information stored in `info` is dumped to a log at line 42.

This example has a typical error found in Java programs caused by ignoring the fact that closing a wrapper file handler will also close its internal file handler. The internal file handler, `fw` is passed as an argument at line 6 and assigned to `out` at line 25. Later, closing its wrapper file handler, `bw`, at line 9 will also close `out` (i.e., `fw`) at line 27. Then any further access to a closed `fw` (e.g., at line 12) will trigger a runtime exception at line 20.

Now we use a static typestate analysis tool CLARA [9] to analyze this program. Some typestate specifications regarding file operations used will include "accessing a closed file leads to an error state". For our example, CLARA produces an error report shown on the left, marking line 12 as a "*potential point of failure*", together with a sequence of two method calls leading to line 12. We have one $SC_{\text{line 12}}$: line 2 $\rightarrow$ line 27 $\rightarrow$ line 12. As static analyses like CLARA are either conservative or unsound, there may or may not be an error at line 12. Now our debugging task begins.


Potential Point of Failure :
Statement: fw.write(info) at line 12
Related Program Points :
Statement: out.close() at line 27
Statement: new FileWriter(...) at line 2

```
 1  class Driver {
 2      Writer fw = new FileWriter(...);
 3      Driver (String s) {...}
 4      Driver () {...}
 5      void config(String[] args) {
 6          Writer bw = new BufferedWriter(fw);
 7          for(int i = 1; i < args.length; i++)
 8              bw.write(args[i] + "\n");
 9          bw.close();
10      }
11      void log(String info) {
12          fw.write(info);
13      }
14  }
15  class FileWriter {
16      boolean isOpen = true;
17      void close() { isOpen = false; }
18      void write(...) {
19          if (!isOpen)
20              throw new IOException();
21      }
22  }
23  class BufferedWriter {
24      Writer out;
25      BufferedWriter(Writer w) {out = w;}
26      void close() {
27          out.close();
28      }
29  }
30
31  void main(String[] args) {
32      Driver d; String info;
33      if (args[0].equals(...)) {
34          d = new Driver(args[0]);
35          d.config(args);
36          info = getConfigInfo(args);
37      } else {
38          d = new Driver();
39          File f = getSystemFile(...);
40          info = getSystemInfo(f);
41      }
42      d.log(info);
43  }
```

Figure 2 An example showing how TAILOR removes $SC_{\text{line } 12}$-irrelevant statements.

The error at line 12 happens only when a `Driver` object is created in the `if` branch of `main()`. Therefore, a tool that instructs the developer to examine this `if` branch only would enable its cause to be identified quickly. In contrast, marking some lines in its matching `else` branch as also being relevant can increase human analysis effort significantly (especially if the developer has to trace the flow of values across many nested heap structures).

Traditional Slicing. For large object-oriented programs, traditional (sound) slicing [21, 53] is unscalable or yields slices that are too large for human consumption [47]. Given a virtual call `fw.write(info)` at line 12, the set of variables of interest consists of (1) the receiver reference and the arguments of the call [52], and (2) some relevant variables selected manually or recognized automatically [2, 13]. In our example, these variables form the set $\{$`fw`, `info`, `fw.isOpen`$\}$. Then, a (backward) slice that affects their values comprises all the statements except lines 7 – 8 and 19 – 20. This slice, which includes everything in `main()`, contains too many statements that are not all directly relevant to the task at line 12.

Thin Slicing. Thin slicing [47] is introduced to facilitate program debugging and understanding for object-oriented programs by trading soundness for scalability and (direct) relevance. All control dependences and all base pointer data dependences are excluded. Given a point of interest, thin slicing includes only so-called *producer statements* that affect directly the values at the point. Statements that serve to explain why producer statements affect the point are ignored. For example, given $x = p.f$ and $q.f = y$, where p and q may be aliased, $q.f = y$ is a producer statement for $x = p.f$, because there may be a direct value flow from y to x. All statements that help explain or establish why p and q are aliases are ignored.

If we adopt the same slicing criterion at line 12 as above, thin slicing will include only seven statements at lines 2, 12, 16, 17, 36, 39 and 40. Compared with the traditional slice obtained, this smaller slice still retains line 17, a statement for explaining an immediate cause of the error at line 12. However, two $SC_{\text{line } 12}$-irrelevant statements at lines 39 – 40 (in the `else` branch of `main()`) are also present, which can cost human analysis effort unnecessarily.

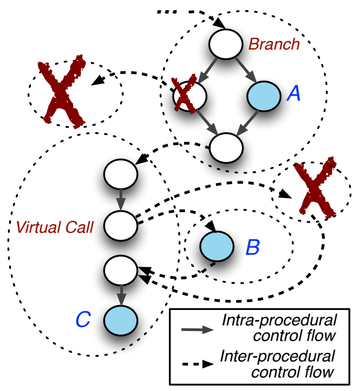


Figure 3 Leveraging the ordering information $SC : A \rightarrow B \rightarrow C$ to trim irrelevant statements away.

Program Tailoring. Given $SC_{\text{line }12}$: line 2→line 27→line 12, Tailor produces a tailored program consisting of all the statements in all execution paths passing through lines 2, 27 and 12 in that order. As line 27 is only reachable from line 9, which resides in the `config` method invoked at line 35, the tailored program includes all the lines in the example *except lines 37 – 41 that appear in the* `else` *branch of* `main()` and lines 19 – 20.

Let us revisit the two slices obtained above, the traditional slice, P_{trad} and the thin slice, P_{thin}. Let our tailored program be identified as P_{tal}. Despite $|P_{\text{thin}}| < |P_{\text{tal}}| < |P_{\text{trad}}|$, tailoring brings several benefits, obtained from exploiting the temporal order of invocation sequences in $SC_{\text{line }12}$. First, P_{tal} is the only one that includes nothing from the `else` branch of `main()`, revealing more clearly to a human debugger that the potential error at line 12 is triggered by a `Driver` object created in the `if` branch of `main()` ("according to some user input") rather than in its matching `else` branch ("according to some default configuration"). Such contextual information enables the developer to locate the cause for the error at line 12 more quickly. In the case of P_{trad} and P_{thin}, the developer may end up wasting a lot of analysis effort on navigating through a lot of $SC_{\text{line }12}$-irrelevant code, highlighted by `getSystemInfo()` and `getSystemFile()`, in the `else` branch. Second, any statement that is not included in P_{tal} is $SC_{\text{line }12}$-irrelevant for understanding the SC-specific behavior at line 12. Thus, Tailor can make, e.g., thin slicing more effective, by removing the $SC_{\text{line }12}$-irrelevant statements $P_{\text{thin}} - P_{\text{thin}} \cap P_{\text{tal}}$, i.e., lines 39 and 40, from P_{thin}. Then P_{thin} is down to only five statements at lines 2, 12, 16, 17 and 36. Starting from line 17 (an immediate cause for the error at line 12), the developer can trace the flow of `isOpen` backwards to find the original cause. Finally, P_{tal} includes all data and control dependences reaching line 12 affected by $SC_{\text{line }12}$, enabling it to be analyzed further by other analyses, e.g., a pointer analysis, as will be discussed in Section 6. However, P_{trad} and P_{thin} will not be applicable since P_{trad} is unobtainable scalably for large programs and P_{thin} is unsound.

Note that P_{tal} still contains lines 7 – 8 that do not affect $SC_{\text{line }12}$. Removing *all* such irrelevant statements for large programs may be neither necessary (due to the first two points made earlier) nor practical, as a sound slicing tool would be unscalable. Thus, we have designed Tailor based on the precision/scalability tradeoffs described in Section 1.

Figure 3 recaps our insight behind tailoring. Given a $SC : A \rightarrow B \rightarrow C$, we focus on the behavior at C affected by a sequential execution of A, B and C. If one point in SC is reached from only one branch (e.g., the one containing A) of a multi-way branching statement, then the statements in the other branches are SC-irrelevant and can be trimmed away (✗). This

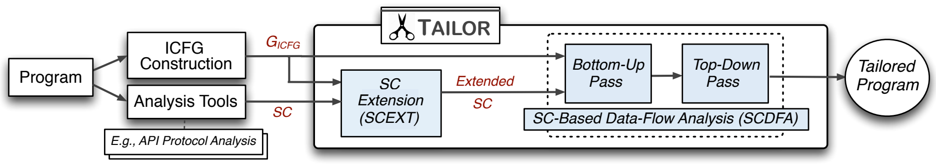


Figure 4 Overview of TAILOR (with all the components implemented in this paper in blue).

is particularly suitable for object-oriented languages, since a virtual call site behaves as a multi-way branch switching to its target methods. For example, B can be regarded as residing in a target method invoked at the marked virtual call on a receiver object created only at the allocation site at A. Thus, the other target methods unreachable to C are trimmed away (✗).

3 Methodology

Figure 4 gives an overview of TAILOR, with all the components implemented in this paper highlighted in blue. Given a program, we rely on the state-of-the-art (shown as "ICFG Construction") to build an inter-procedural control flow graph (ICFG), denoted G_{ICFG}, to represent all the possible control flows in the program. A SC is simply a set of one or more statement sequences ending at the same statement, with all statements identified by their line numbers, i.e., program points only. The *length* of a SC is the number of statements in its longest sequence. SCs can be deduced from the results returned by many analysis tools such as API protocol analysis [7, 37] and typestate analysis [9, 16, 34]. For example, a typestate analysis may report a potential error at line C, `f.write()`, together with two invocation sequences of related methods, A : `f.open()` $\rightarrow B1$: `f1.close()` and A : `f.open()` $\rightarrow B2$: `f2.close()`, leading to line C. Therefore, we may choose to tailor the program at C to investigate its behavior affected by $SC = \{A \rightarrow B1 \rightarrow C, A \rightarrow B2 \rightarrow C\}$.

Given a SC, a tailored program, $\mathcal{T}(SC)$, consists of the statements on all possible execution paths in G_{ICFG} passing through at least one statement sequence in SC. By exploiting the temporal ordering information in SC, it is possible to scale tailoring significantly better than slicing for large object-oriented programs and makes it a practically useful technique.

TAILOR is sound as it computes $\mathcal{T}(SC)$ over-approximately with respect to G_{ICFG}. In contrast, traditional (sound) slicing [21] is unscalable when G_{ICFG} is large and thin slicing [47] is unsound for G_{ICFG} (as it is designed for program debugging and understanding only).

TAILOR computes $\mathcal{T}(SC)$ in two stages, SC Extension and SC-based Data-Flow Analysis. In the first stage (Section 3.2), we exploit "branch correlations" in object-oriented programs to lengthen a given SC in order to avoid some infeasible paths that would otherwise be introduced in the second stage. In the second stage (Section 3.1), we compute $\mathcal{T}(SC)$ by solving a data-flow problem in order to avoid unrealizable paths efficiently. This design allows TAILOR to achieve good precision and scale well to large object-oriented programs.

3.1 SC-Based Data-Flow Analysis

We compute $\mathcal{T}(SC)$ by solving flow- and context-sensitively an IFDS (Interprocedural Finite Distributive Subset) data-flow problem [38], efficiently on G_{ICFG}, via graph reachability. This formulation of our SC-based data-flow analysis, denoted SCDFA, is significant for three reasons. (1) With flow-sensitivity, SCDFA can filter out imprecisely ordered statement sequences in a SC, as many static analyses from which SCs are deduced are flow-insensitive

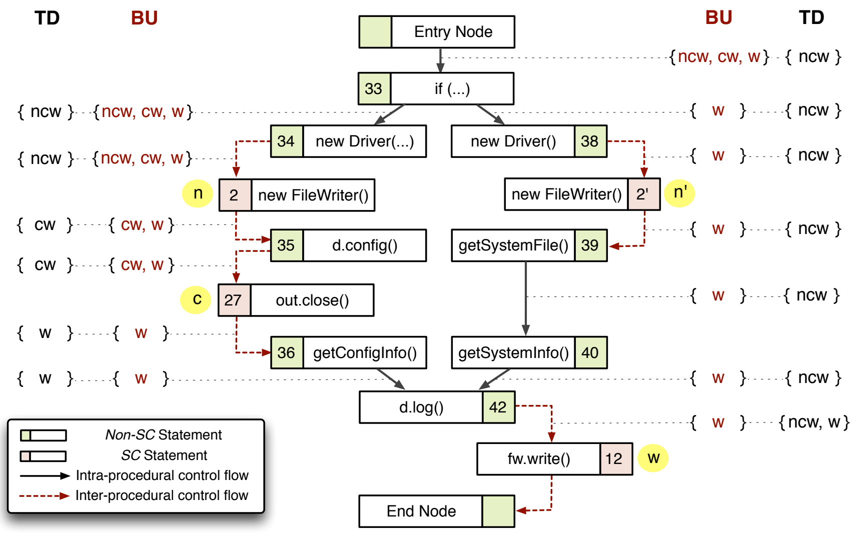


Figure 5 A simplified G_{ICFG} of the program given in Figure 2 for illustrating the bottom-up (BU) and top-down (TD) passes of SCDFA with $SC_w = \{n{\to}c{\to}w, n'{\to}c{\to}w\}$.

in order to be scalable. Conversely, precisely ordered statement sequences in SC also enable SCDFA to filter out irrelevant control flows in tailoring a program. This mutually beneficial process improves the precision of both parts, therefore avoiding the unnecessary complexity faced for solving both as one problem. (2) With context-sensitivity, we avoid introducing unrealizable paths with mismatched calls and returns, which is critical for achieving precision in computing $\mathcal{T}(SC)$. (3) With an IFDS formulation, SCDFA can scale well to reasonably large object-oriented programs. In particular, (full) context-sensitivity can be realized more efficiently by solving a CFL-reachability problem over a simplified balanced-parentheses language. As an IFDS problem, SCDFA has a time complexity of $O(ED^3)$, where E is the number of edges in G_{ICFG} and D is the size of the set of SC-related data-flow facts used (i.e., the set of suffixes of sequences in SC, as will be clear below and defined in Section 4).

SCDFA starts with a bottom-up pass (BU) and finishes with a top-down pass (TD), with both performed (fully) context-sensitively. By using the example program from Figure 2, we first explain the functionalities of two passes and then examine briefly how context-sensitivity is realized efficiently in the IFDS framework. Suppose we are given a SC as line 2 : `fw = new FileWriter()` $\to$ line 27 : `out.close()` $\to$ line 12 : `fw.write()`. For convenience, we write it as $SC : n{\to}c{\to}w$. In its G_{ICFG}, the allocation site for `FileWriter` at line 2 is replicated in the two constructors of `Driver`. We identify the one in the single-arg constructor as line 2 (denoted by n) and the one in the non-arg constructor as line 2′ (denoted by n'). As a result, we finally have a two-sequence $SC_w = \{n{\to}c{\to}w, n'{\to}c{\to}w\}$.

***BU* and *TD* Passes.** Both passes operate on G_{ICFG}, as shown in Figure 5, for our example. As in any data-flow analysis, all control flow edges in G_{ICFG} are treated non-deterministically executable. Let ENTRY be the entry node of G_{ICFG} and EXIT the node marking the point of interest w at line 12 in SC_w. Conceptually, if a sequence $S \in SC_w$, which is ncw or $n'cw$, lies on a path, then BU must see a suffix of S at every node n on the path backwards from EXIT and TD must see the corresponding prefix of S at n forwards from ENTRY.

BU computes a global property, PANTI, for every node n in G_{ICFG}, backwards against the

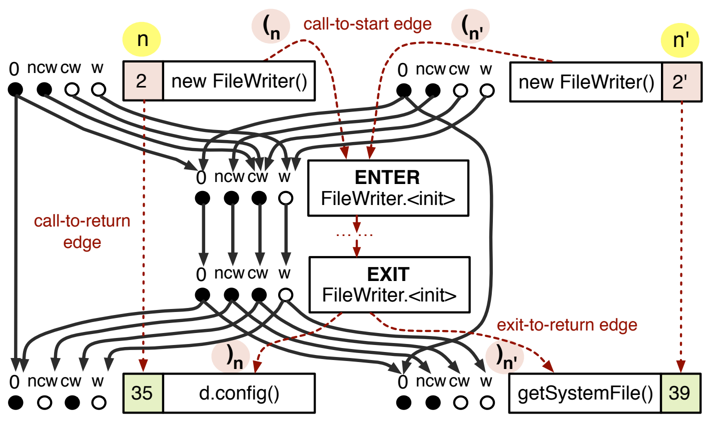


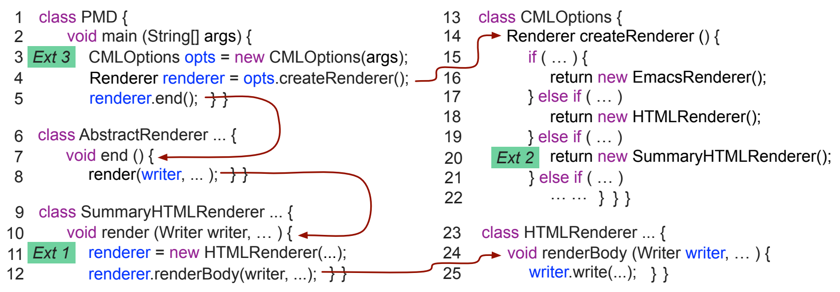
Figure 6 Context-sensitivity via CFL-reachability.

control flow, starting at EXIT. $\text{PANTI}^{\text{out}}(n)$ represents the set of suffixes of some sequences in SC_w that are partially anticipable at the entry of n, i.e., appear on some outgoing path of n ending at EXIT. In our example, $\text{PANTI}^{\text{out}}(\text{ENTRY}) = \{ncw, cw, w\}$. As ncw is partially anticipable (but $n'cw$ is not) at ENTRY, G_{ICFG} contains some paths passing through ncw.

TD computes a global property, PAVAIL, for every node n in G_{ICFG}, forwards along the control flow, starting from ENTRY. $\text{PAVAIL}^{\text{in}}(n)$ specifies the set of suffixes of some sequences in $\text{PANTI}^{\text{out}}(\text{ENTRY})$ to represent implicitly (for efficiency reasons) the fact that their corresponding prefixes are partially available at the entry of n, i.e., appear on some incoming paths of n starting from ENTRY. For example, $\text{PAVAIL}^{\text{in}}(12) = \{ncw, w\}$, indicating that the prefixes ϵ and nc of ncw are partially available at the entry of node 12.

For this example, a node n is included in the tailored program if $\text{PAVAIL}^{\text{in}}(n) \cap \text{PANTI}^{\text{out}}(n) \neq \varnothing$, since a suffix $s \in \text{PAVAIL}^{\text{in}}(n) \cap \text{PANTI}^{\text{out}}(n)$ is partially anticipable at the entry of n and some sequence in SC_w with s removed is partially available at the entry of n. In our example, the tailored program consists of all the lines except lines 38 – 40.

Context Sensitivity. Figure 6 illustrates as an example how the *TD* pass shown in Figure 5 is performed for `FileWriter()` context-sensitively by solving a CFL-reachability-based balanced-parentheses problem, efficiently in the IFDS framework. For technical details, we refer to [38]. There are four data-flow facts, ncw, cw, w, and 0 (for the empty set). There are two call sites to `FileWriter()` at lines 2 and 2′, which are identified as nodes n and n', respectively, with their call-to-start and exit-to-return edges labeled as $(_n$, $)_n$, $(_{n'}$ and $)_{n'}$ appropriately. ENTER and EXIT are the start and exit nodes of `FileWriter()`, whose CFG is irrelevant and thus elided. In SCDFA, the call-to-return edge always serves as a kill edge (to stop the data-flow facts from bypassing a callee). For each node n, $\text{PAVAIL}^{\text{in}}(n)$ is the set of facts, i.e., suffixes of ncw shown in black dots. According to Figure 5, $\text{PAVAIL}^{\text{in}}(2) = \text{PAVAIL}^{\text{in}}(2') = \{ncw\}$. If `FileWriter()` is entered from $(_n$ and exited from $)_n$, then $\text{PAVAIL}^{\text{in}}(\text{ENTER}) = \text{PAVAIL}^{\text{in}}(\text{EXIT}) = \{cw\}$. Hence, $\text{PAVAIL}^{\text{in}}(35) = \{cw\}$. However, if `FileWriter()` is entered from $(_{n'}$ and exited from $)_{n'}$, then $\text{PAVAIL}^{\text{in}}(\text{ENTER}) = \text{PAVAIL}^{\text{in}}(\text{EXIT}) = \{ncw\}$. Hence, $\text{PAVAIL}^{\text{in}}(39) = \{ncw\}$.



```
1        class PMD {
2            void main (String[] args) {
3  Ext 3         CMLOptions opts = new CMLOptions(args);
4                Renderer renderer = opts.createRenderer();
5                renderer.end(); } }

6        class AbstractRenderer ... {
7            void end () {
8                render(writer, ... ); } }

9        class SummaryHTMLRenderer ... {
10           void render (Writer writer, … ) {
11 Ext 1         renderer = new HTMLRenderer(...);
12               renderer.renderBody(writer, ...); } }

13       class CMLOptions {
14           Renderer createRenderer () {
15               if ( … ) {
16                   return new EmacsRenderer();
17               } else if ( … )
18                   return new HTMLRenderer();
19               } else if ( … )
20 Ext 2             return new SummaryHTMLRenderer();
21               } else if ( … )
22                   … …  } } }

23       class HTMLRenderer ... {
24           void renderBody (Writer writer, … ) {
25               writer.write(...);   } }
```

Figure 7 A code snippet from PMD for illustrating SCEXT in a program understanding task.

3.2 SC Extension

TAILOR is designed to work effectively with any *SC*. While an API specification mining tool [17] may generate *SCs* longer than 10, an assertion verifier may produce only single-point *SCs*. In general, the longer a *SC* is, the more *SC*-irrelevant statements will be eliminated.

To improve the precision of SCDFA, we perform first a SC extension pass, denoted SCEXT, as shown in Figure 4. Given a sequence $S \in SC$, with F being its first point, we look for a set of n extension points $E_1, \cdots, E_n$, that collectively dominate F, such that any program execution that passes through F must pass through one E_i. We do so effectively by leveraging the concept of object-sensitivity [33,44] developed by the pointer analysis community for Java. For F, its extension points are chosen as the object allocation sites used for representing the object-sensitive calling contexts for the method containing F. As a result, some infeasible paths are avoided by exploiting branch correlations in object-oriented programs (Figure 3).

To make SC extensions, we use the points-to information obtained by, e.g., the pointer analysis performed earlier for building G_{ICFG}. Consider an ICFG fragment discussed earlier in Figure 3. Suppose we are given $SC : B \rightarrow C$ such that B resides in a target method m for the virtual call site shown and m is only invokable on the objects created at A, which represents an object allocation site. Then we can prepend A to SC to obtain $ESC : A \rightarrow B \rightarrow C$. According to SCDFA, $\mathcal{T}(SC)$ will include both branches of "Branch" shown in Figure 3 since both reach SC but $\mathcal{T}(ESC)$ will include only the branch containing A as the other (infeasible) branch does not reach ESC. However, lengthening a SC increases the number of SC-related data-flow facts used (Figure 5). So a precision/scalability tradeoff needs to be made.

We use a real program understanding task to show why SCEXT is useful for real code. PMD is a static analyzer for analyzing Java programs and can print a range of source code flaws in different formats such as Emacs, CSV, and HTML. In this task, we want to understand how PMD renders its outputs in the commonly used HTML format. Figure 7 shows the simplified code. The only knowledge we have initially is that the HTMLRenderer class is responsible for writing to the file at line 25, but how it is done is unknown. At this stage, we have a single-point, $SC_{\text{line 25}}$: line 25, representing just the write statement at line 25.

If we apply SCDFA to $SC_{\text{line 25}}$, $\mathcal{T}(SC_{\text{line 25}})$ will include all the lines in the code given. Below we show how to extend this to $ESC_{\text{line 25}}$: line 20 $\rightarrow$ line 11 $\rightarrow$ line 25 object-sensitively [33, 44]. If we apply SCDFA to $ESC_{\text{line 25}}$, $\mathcal{T}(ESC_{\text{line 25}})$ will now be smaller, consisting of all the lines except lines 16, 18 and 22 in method createRenderer(). In other words, all the branches except the one enclosing line 20 in method createRenderer() are infeasible for $SC_{\text{line 25}}$, according to the points-to information provided for this program.

Let us see how SCEXT works in growing $SC_{\text{line 25}}$ to become $ESC_{\text{line 25}}$. Initially, SC_{line25} has one statement at line 25, which resides in method renderBody(). The (abstract) receiver object pointed to by `renderer` at line 12 is allocated only at the allocation site at line 11 and is considered as the calling context for line 25 in an object-sensitive pointer analysis. There is another allocation site at line 18 for the same type, `HTMLRenderer`, but this is not considered, since the abstract object created at line 18 does not flow to line 12 (according to the points-to information available). At this stage, we have $SC_{\text{line 25}}$: line 11 $\rightarrow$ line 25.

Similarly, we now look for the allocation sites that can be used as the calling contexts for line 11 contained in method `render()` at line 10, which is called from `this.render()` at line 8 in method `end()` defined in the `AbstractRenderer` class. Thus, the receiver objects on which method `render()` (line 10) is invoked are the ones pointed to by `renderer` at line 5. These objects are returned from the call to `createRenderer()` at line 4. Thus, `renderer` points to the objects created by the allocation sites in all different branches in `createRenderer()`. According to the points-to information, the target method `end()` invoked at the virtual call site at line 5 can only be made on an receiver object of type `SummaryHTMLRenderer`. Thus, we now have an even longer $SC_{\text{line 25}}$: line 20 $\rightarrow$ line 11 $\rightarrow$ line 25.

The allocation site at line 3 is the object-sensitive context for the target method `createRenderer()`, which contains line 20, invoked at line 4. No further extension is possible, since `main()` has been reached. Now, we have $SC_{\text{line 25}}$: line 3 $\rightarrow$ line 20 $\rightarrow$ line 11 $\rightarrow$ line 25.

If we apply SCDFA to this four-point $SC_{\text{line 25}}$, $\mathcal{T}(SC_{\text{line 25}})$ will consist of all the lines except lines 16, 18 and 22. However, including line 3 does not help as it dominates line 20 in G_{ICFG}. By removing it, we obtain $ESC_{\text{line 25}}$: line 20$\rightarrow$line 11$\rightarrow$line 25 as desired. Applying SCDFA to $ESC_{\text{line 25}}$ yields the same tailored program obtained for the three-point $SC_{\text{line 25}}$.

When lengthening *SCs*, we aim to reduce infeasible paths by exploiting branch correlations. However, with longer *SCs*, SCDFA will end up using more data-flow facts, as seen in Figure 5, making it less efficient than before. So a precision/scalability tradeoff has to be made. Our key observation is to keep a *SC* extension point found if it is inside one branch in a multi-way branching statement or a target method invoked at a polymorphic call site (Figure 3), provided that the point is not in a cycle. This way, the other branches (or target methods) are infeasible (with respect to a given *SC*) and will not show up in the tailored program.

Let us return to the program understanding task at hand. From the tailored program $\mathcal{T}(ESC_{\text{line 25}})$ obtained for $ESC_{\text{line 25}}$, we can see clearly that the allocation site at line 20 in method `createRenderer()` is responsible for the rendering-related write at line 25.

4 Formalism

We first formalize SCDFA and SCEXT and then prove some properties about Tailor.

4.1 SC-based Data-Flow Analysis

We provide a context-insensitive formalization of SCDFA as context sensitivity is achieved on top of this formalization via CFL-reachability, as shown in Figure 6. Essentially, we describe the data-flow equations needed for solving its *BU* and *TD* passes in the IFDS framework [38].

The IFDS problem framework is precise and efficient for solving data-flow problems with three properties. First, the set $\mathcal{D}$ of data-flow facts is finite. Second, the domain and range of each flow, i.e., transfer function is the power set of $\mathcal{D}$, denoted $2^{\mathcal{D}}$, the lattice ordering relation $\sqsubseteq$ on $2^{\mathcal{D}}$ is $\subseteq$ or $\supseteq$, and the meet operator $\sqcap$ is $\cap$ or $\cup$. Finally, each fow function f must be distributive over $2^{\mathcal{D}}$: $\forall S_1, S_2 \in 2^{\mathcal{D}}$: $f(S_1 \sqcap S_2) = f(S_1) \sqcap f(S_2)$.

Below we describe our *BU* and *TD* passes, including their finite domain $\mathcal{D}$ and data-flow equations used. In both cases, the ordering relation $\sqsubseteq$ is $\supseteq$ and the meet operator $\sqcap$ is $\cup$. All our transfer functions given below are easily seen to be distributive over $2^{\mathcal{D}}$.

Domain. A SC_P is a set of statement sequences ending at the same statement P, with all statements identified by their line numbers (or labels). The domain $\mathcal{D}$ for BU and TD is:

$$\mathcal{D} = \bigcup_{S \in SC_P} \mathit{Suffix}(S) \tag{1}$$

where $\mathit{Suffix}(S)$ returns the set of all suffixes of S, including ϵ (the empty string). Note that ϵ is necessary when P appears in a control-flow cycle (i.e., a loop or recursion). In the IFDS framework, the data-flow facts used are generated on the fly for efficiency reasons.

We make use of the *car*, *cdr* and *cons* functions operating on sequences in the standard manner. If s and s' are two sequences, then $s + s'$ returns the concatenation of s and s'.

Graph Representation. The BU and TD passes operate on the G_{ICFG} representation of a program. Without loss of generality, we assume that a node, i.e., basic block in G_{ICFG} contains one single statement. Thus, a node and the statement represented by it are used interchangeably. Let ENTRY be the entry node of G_{ICFG} and EXIT be the node for P in SC_P. Given a node n, $\mathsf{succ}(n)$ ($\mathsf{pred}(n)$) is the set of its successors (predecessors) in G_{ICFG}.

***BU*: Bottom-Up Analysis.** BU computes a global property, PANTI, for every node n in G_{ICFG}, backwards against the control flow, starting at EXIT. $\mathsf{PANTI}^{\text{out}}(n)$ represents the set of suffixes in $\mathcal{D}$ that are partially anticipable at the entry of n, i.e., that appear on some outgoing path of n ending at EXIT. Thus, we have the following initialization at EXIT:

$$\mathsf{PANTI}^{\text{out}}(\mathsf{EXIT}) = \{P\} \tag{2}$$

which means that our point of interest P in SC_P is partially anticipable at the entry of EXIT.

The transfer equations at a node n in G_{ICFG} are given by:

$$\mathsf{PANTI}^{\text{in}}(n) = \bigcup_{n' \in \mathsf{succ}(n)} \mathsf{PANTI}^{\text{out}}(n') \tag{3}$$

$$\mathsf{PANTI}^{\text{out}}(n) = \mathsf{GEN}_{BU}(n) \cup \mathsf{PANTI}^{\text{in}}(n) \tag{4}$$

Due to the presence of $\mathsf{PANTI}^{\text{in}}(n)$ in (4), whatever is anticipable at the exit of n are also anticipable at the entry of n. No old data-flow facts (i.e., known suffixes) are killed, because some sequences in SC_P may share some common suffixes but have different prefixes. GEN_{BU} is applied to generate all the new suffixes partially anticipable at a node:

$$\mathsf{GEN}_{BU}(n) = \bigcup_{s \,\in\, \mathsf{PANTI}^{\text{in}}(n)} \mathsf{ADD\text{-}SUFFIX\text{-}SEEN}_{BU}(n, s) \tag{5}$$

Given a partially anticipable suffix s at the exit of a node n, $\mathit{cons}(n, s)$ will also be partially anticipable at the entry of n if $\mathit{cons}(n, s) \in \mathcal{D}$. Hence, we have:

$$\mathsf{ADD\text{-}SUFFIX\text{-}SEEN}_{BU}(n, s) = \begin{cases} \{\mathit{cons}(n, s)\} & \textbf{if } \mathit{cons}(n, s) \in \mathcal{D} \\ \varnothing & \textbf{otherwise} \end{cases} \tag{6}$$

The following lemma follows immediately from the data-flow equations (2) – (6).

▶ **Lemma 1.** *For any node n, $\epsilon \notin \mathsf{PANTI}^{\text{in}}(n)$ and $\epsilon \notin \mathsf{PANTI}^{\text{out}}(n)$ always hold.*

▶ **Example 2.** Figure 5 illustrates the BU pass with the data-flow results shown. We start with $\mathsf{PANTI}^{\text{out}}(\mathsf{EXIT}) = \mathsf{PANTI}^{\text{out}}(12) = \{w\}$ and finish with $\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY}) = \{ncw, cw, w\}$.

***TD*: Top-Down Analysis.** TD computes a global property, PAVAIL, for every node n in G_{ICFG}, forwards along the control flow, starting from ENTRY and visiting only the nodes reachable in BU. The basic idea is simple. A statement sequence $S \in SC_P$ starts at ENTRY and flows forwards along the control flow and ends up with its first statement $car(S)$ removed on encountering the node representing $car(S)$. Therefore, a node n is included in the tailored program $\mathcal{T}(SC_P)$ if the remaining suffix of S that flows to the entry of a node n appears also in $\text{PANTI}^{\text{out}}(n)$ or the entire sequence S has been removed upon reaching n (when P in SC_P appears in a control-flow cycle). Formally, $\text{PAVAIL}^{\text{in}}(n)$ computes the set of suffixes $s \in \mathcal{D}$ to represent implicitly (for efficiency reasons) the fact that their corresponding prefixes p, such that $p +\!\!+\, s = S$ for some $S \in SC_P$, are partially available at the entry of n.

As all the sequences in $SC_P \setminus \text{PANTI}^{\text{out}}(\text{ENTRY})$ have been filtered out by BU, we only need to focus on the ones partially anticipable at ENTRY. Hence, our initialization is:

$$\text{PAVAIL}^{\text{in}}(\text{ENTRY}) = SC_P \ \cap \ \text{PANTI}^{\text{out}}(\text{ENTRY}) \tag{7}$$

The transfer equations at a non-ENTRY node in G_{ICFG} are given by:

$$\text{PAVAIL}^{\text{in}}(n) = \bigcup_{n' \in \text{pred}(n)} \text{PAVAIL}^{\text{out}}(n') \tag{8}$$

$$\text{PAVAIL}^{\text{out}}(n) = \begin{cases} \text{GEN}_{TD}(n) & \textbf{if } \text{PANTI}^{\text{out}}(n) \neq \varnothing \\ \varnothing & \textbf{otherwise} \end{cases} \tag{9}$$

During the top-down pass, we only need to visit a node n if n is reachable during the bottom-up pass, which happens when $\text{PANTI}^{\text{out}}(n) \neq \varnothing$.

Unlike GEN_{BU}, GEN_{TD} may preserve/kill an old data-fact and generate some new ones:

$$\text{GEN}_{TD}(n) = \bigcup_{s \,\in\, \text{PAVAIL}^{\text{in}}(n)} \text{ADD-SUFFIX-EXPECTED-TO-SEE}_{TD}(n, s) \tag{10}$$

As a suffix $s \in \text{PAVAIL}^{\text{in}}(n)$ represents the fact that its corresponding prefix p, such that $p +\!\!+\, s = S$ for some $S \in SC_P$, is partially available at the entry of n, we have:

$$\text{ADD-SUFFIX-EXPECTED-TO-SEE}_{TD}(n, s) = \begin{cases} \{cdr(s)\} & \textbf{if } car(s) = n \\ \{s\} & \textbf{otherwise} \end{cases} \tag{11}$$

There are two cases. If $car(s) = n$, then $cdr(s)$ is generated, indicating that $p +\!\!+\, n$ is partially available at the exit of n. At the same time, s is killed (for efficiency not correctness), as s would be redundantly propagated otherwise. If $car(s) \neq n$, then s is simply preserved.

▶ **Example 3.** Figure 5 illustrates the TD pass with the data-flow results shown. We start with $\text{PAVAIL}^{\text{in}}(\text{ENTRY}) = SC_w \cap \text{PANTI}^{\text{out}}(\text{ENTRY}) = \{ncw\}$ and finish with $\text{PAVAIL}^{\text{in}}(\text{EXIT}) = \text{PAVAIL}^{\text{in}}(12) = \{ncw, w\}$. At node 27, $\text{PAVAIL}^{\text{out}}(27) = \{w\}$, since $\text{PAVAIL}^{\text{in}}(27) = \{cw\}$.

Tailored Program. Finally, a node n is included in $\mathcal{T}(SC_P)$ if TAILORED(n) holds:

$$\text{TAILORED}(n) = \text{PAVAIL}^{\text{in}}(n) \cap \text{PANTI}^{\text{out}}(n) \neq \varnothing \vee \epsilon \in \text{PAVAIL}^{\text{in}}(n) \tag{12}$$

The first disjunct suffices if P in SC_P is not in a cycle (in G_{ICFG}). If $s \in \text{PAVAIL}^{\text{in}}(n) \cap \text{PANTI}^{\text{out}}(n)$, then a prefix p, such that $p +\!\!+\, s = S$ for some $S \in \text{PAVAIL}^{\text{in}}(\text{ENTRY})$, is partially available at the entry of n, then n must be in $\mathcal{T}(SC_P)$, since it lies on a path passing through all statements in S. If P is in a cycle, then the whole sequence S that starts at ENTRY ends up being removed eventually, resulting in $\epsilon \in \text{PAVAIL}^{\text{in}}(n)$. Then n should be included in $\mathcal{T}(SC_P)$ as well. Note that $\epsilon \notin \text{PANTI}^{\text{out}}(n)$ by Lemma 1.

▶ **Example 4.** According to the data-flow facts shown for the program given in Figure 5, the tailored program consists of all the lines except lines 38 – 40 according to (12).

4.2 SC Extension

We make use of the points-to information provided by a pointer analysis to extend a *SC* to reduce infeasible paths that would otherwise be introduced by SCDFA. Given a statement sequence $S \in SC$, SCEXT will lengthen it recursively by prepending the object allocation sites representing the calling contexts for the method containing its first statement, as demonstrated in Section 3.2. The general algorithm for lengthening a sequence $S : L_1 \rightarrow \cdots \rightarrow L_n$ is as follows. Suppose that L_1 resides in a method m invoked at a virtual call site. Let $A_1, \cdots, A_r$ be its all r allocation sites for creating the receiver objects on which m is invoked. Then S grows into $A_1 \rightarrow L_1 \rightarrow \cdots \rightarrow L_n, \cdots, A_r \rightarrow L_1 \rightarrow \cdots \rightarrow L_n$, and the same process continues.

With longer *SCs*, SCDFA will be less efficient due to more data-flow facts introduced. We will keep a SC extension point if it is *embedded* in a branch and ignore it otherwise. This way, SCEXT can enable SCDFA to avoid infeasible paths more effectively by exploiting branch correlations. A statement is said to be embedded in a branch if it appears intraprocedurally (directly) or interprocedurally (indirectly) inside a multi-way branching statement or a target method invoked at a *polymorphic call site* (i.e., a virtual call site with at least two target methods). Let *Stmts* be the set of all statements in G_{ICFG}. We use the following function $\mathit{InBranchOrVC} : \mathit{Stmts} \rightarrow \textbf{boolean}$ to capture formally this branch-embedding relation:

$$\mathit{InBranchOrVC}(s) = \begin{cases} \textbf{true} & \textbf{if } \mathit{InIntraBranch}(s) \\ \textbf{false} & \textbf{else if } \mathit{InMain}(s) \\ \mathit{InInterBranchOrVC}(s) & \textbf{otherwise} \end{cases} \tag{13}$$

where $\mathit{InIntraBranch}(s)$ determines if s appears directly in a branch or not and $\mathit{InMain}(s)$ tells us whether s appears directly in `main()` or not. $\mathit{InInterBranchOrVC} : \mathit{Stmts} \rightarrow \textbf{boolean}$ checks to see whether s is embedded in a branch interprocedurally or not:

$$\mathit{InInterBranchOrVC}(s) = \bigvee_{c \in \mathit{Caller}(m)} \Big(\mathit{InBranchOrVC}(c) \ \vee \ |\mathit{Callee}(c)| > 1 \Big) \\ \textbf{where } m \textbf{ is the method containing } s \tag{14}$$

where $\mathit{Caller}(m)$ returns the set of call sites at which m is invoked. For each call site c, there are two disjuncts. One represents a recursive application of *InBranchOrVC* defined in (13) to c. The other one, $|\mathit{Callee}(c)| > 1$, evaluates to **true** if the call site c is polymorphic.

▶ **Example 5.** For the program in Figure 7, as discussed in Section 3.2, SCEXT starts with $SC_{\text{line 25}}$: line 25, grows it to $SC_{\text{line 25}}$: line 3 $\rightarrow$ line 20 $\rightarrow$ line 11 $\rightarrow$ line 25, and finally settles with $ESC_{\text{line 25}}$: line 20 $\rightarrow$ line 11 $\rightarrow$ line 25. Let us see why a SC extension point is kept or ignored. Line 3 should be ignored since $\mathit{InBranchOrVC}(\text{line 3}) = \neg \mathit{InMain}(\text{line 3}) =$ **false**. Line 20 is retained since $\mathit{InInterBranchOrVC}(\text{line 20}) = \mathit{InIntraBranch}(\text{line 20}) =$ **true**. Finally, line 11 is also retained because we have $\mathit{InInterBranchOrVC}(\text{line 11}) = \mathit{InInterBranchOrVC}(\text{line 8}) = \mathit{InInterBranchOrVC}(\text{line 5}) = |\mathit{Callee}(\text{line 5})| > 1 =$ **true**, where the call site at line 5 is polymorphic according to the points-to information provided.

In practice, SCDFA does not usually benefit from a SC extension point if it appears in a control-flow cycle (a loop or a recursion cycle) in G_{ICFG}. Such cycle-related points are identified and ignored as well. To detect recursion cycles effectively, we apply Tarjan's algorithm [48] to find strongly connected components on the call graph of the program. To detect (natural) loops, we resort to a textbook loop detection algorithm. The statements reachable directly or indirected from a loop are also considered as being part of the loop.

statement	$s, s_1, s_2, ..., s_m \in \mathbb{S}$
execution	$e : s_1 s_2 ... s_m \in \mathbb{E}^G$
sequence	$sq : s_1 s_2 ... s_m \in SC$
relation $\mathbb{E}^G \times SC$	$e \rightsquigarrow sq$
relation $\mathbb{E}^G \times \mathbb{S}$	$e \to s$
execution set	$\mathbb{E}^{sq} = \{e \in \mathbb{E}^G \mid e \rightsquigarrow sq\}$
statement set	$\mathbb{S}^{sq} = \{s \in \mathbb{S} \mid e \in \mathbb{E}^{sq}, e \to s\}$

Figure 8 Notations used in proofs.

4.3 Properties

We prove that TAILOR is sound (Theorem 3), by showing that SCDFA and SCEXT are sound with respect to G_{ICFG} (Theorems 1 and 2), and consequently, that every tailored program is *SC-executable*, i.e., executable along all execution paths passing through a given *SC*.

We make use of the notations given in Figure 8 in our proofs. $\mathbb{S}$ is a set of statements in G_{ICFG}. $\mathbb{E}^G$ represents all runtime executions of the program represented by G_{ICFG} and each execution e consists of a sequence of statements in $\mathbb{S}$. We write $e \rightsquigarrow sq$ if execution e contains all the statements in a statement sequence $sq = s_1 s_2 \cdots s_m$ in exactly the same order, ending at s_m. We write $e \to s$ if execution e contains statement s. $\mathbb{E}^{sq}$ is the set of all executions that pass through a given sequence sq. Finally, $\mathbb{S}^{sq}$ is the set of all statements that appear in all possible executions e (passing through sq), where $e \in \mathbb{E}^{sq}$.

The following theorem states that SCDFA is sound with respect to G_{ICFG}.

▶ **Theorem 1** (Soundness of SCDFA). *Let G_{ICFG} be the ICFG of a program. Let SC be given (as defined in Section 3). If $sq \in SC$, then $s \in \mathbb{S}^{sq} \implies$ TAILORED(s).*

Proof. We show that for every execution e such that $e \rightsquigarrow sq$, where $sq \in SC$, TAILORED(s), which is given in (12), holds for all statements s such that $e \to s$. Let $sq = s_1 s_2 ... s_m$. Since $e \rightsquigarrow sq$, every statement s_i must appear at least once in e or more in the presence of control-flow cycles. For convenience, let s_e^0 be a fictitious statement at the beginning of e. Let s_e^{m+1} be the last occurrence of s_m in e, i.e., the last statement in e. Let s_e^i be the first occurrence of s_i in e after s_e^{i-1}, where $1 \leqslant i \leqslant m$. We can now divide e into $m+1$ sub-executions, $e_1, e_2, \cdots, e_{m+1}$, where e_i, represents the sub-sequence between s_e^{i-1} (exclusive) and s_e^i (inclusive). We will still write $e_i \to s$ if e_i contains a statement s.

As $e \rightsquigarrow sq$ is an execution, then e represents a realizable path, which must appear in G_{ICFG}. In SCDFA, its *BU* and *TD* passes are distributive over $2^{\mathcal{D}}$ given in (1), performed context-sensitively. Thus, we only need to focus on this path. Note that in our formulation of SCDFA, a statement s_i and its corresponding node n_i in G_{ICFG} are used interchangeably.

During the *BU* pass in terms of (2) – (6), we must have:

$$\forall\ 1 \leqslant i \leqslant m : \forall\ n_i\ s.t.\ e_i \to n_i : s_i s_{i+1} \cdots s_m \in \mathsf{PANTI}^{\text{out}}(n_i) \tag{15}$$

$$\forall\ n_{m+1}\ s.t.\ e_{m+1} \to n_{m+1} : s_m \in \mathsf{PANTI}^{\text{out}}(n_{m+1}) \tag{16}$$

which implies $sq \in \mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})$ (since $\forall\ n_1\ s.t.\ e_1 \to n_1 : s_1 s_2 \cdots s_n \in \mathsf{PANTI}^{\text{out}}(n_1)$).

During the *TD* pass, $sq \in \mathsf{PAVAIL}^{\text{in}}(\mathsf{ENTRY})$ by (7). By (8) – (11), we must have:

$$\forall\ 1 \leqslant i \leqslant m : \forall\ n_i\ s.t.\ e_i \to n_i : s_i s_{i+1} \cdots s_m \in \mathsf{PAVAIL}^{\text{in}}(n_i) \tag{17}$$

$$\forall\ n_{m+1}\ s.t.\ e_{m+1} \to n_{m+1} : \epsilon \in \mathsf{PAVAIL}^{\text{in}}(n_{m+1}) \tag{18}$$

(Note that (18) is needed only if e_{m+1} is non-empty, which happens when s_m is in a control-flow cycle.) Hence, TAILORED(s) holds for every statement s such that $e \to s$. ◀

For a SC, we write $\alpha(G_{\text{ICFG}}, SC)$ to represent the set of all executions in G_{ICFG} that pass at least one sequence in SC, i.e., $\alpha(G_{\text{ICFG}}, SC) = \bigcup_{sq \in SC} \mathbb{E}^{sq}$. Theorem 1 can also be stated equivalently as follows: TAILORED(s) holds for every $s \in \{s \mid e \in \alpha(G_{\text{ICFG}}, SC), e \to s\}$.

The following theorem states that SCEXT is sound since only infeasible paths are ignored.

▶ **Theorem 2** (Soundness of SCEXT). *Let ESC be extended from a given SC by applying* SCEXT *in* G_{ICFG}*, then* $\alpha(G_{\text{ICFG}}, SC) = \alpha(G_{\text{ICFG}}, ESC)$*.*

Proof. According to the algorithm of SCEXT operating in G_{ICFG} (Section 4.2), all possible object-sensitive calling contexts for a method containing the first point F in a sequence of SC are considered as the extension points of F. According to (13) and (14), no feasible paths with respect to SC are excluded if an extension point of F is not selected. Thus, only infeasible paths with respect to SC are ignored when SC is extended into ESC this way. ◀

▶ **Theorem 3** (Soundness of TAILOR). TAILOR *is sound with respect to* G_{ICFG}*.*

Proof. Follows from Theorems 1 and 2. ◀

▶ **Theorem 4** (SC-Executability). $\mathcal{T}(SC)$ *obtained in* G_{ICFG} *is* SC*-executable.*

Proof. By Theorem 3, $\alpha(G_{\text{ICFG}}, SC)$ is included in the tailored program. ◀

5 Implementation

We have implemented TAILOR (`http://www.cse.unsw.edu.au/~corg/tailor`) in SOOT, a framework for analyzing and optimizing Java programs [51]. To build the ICFG for a program, we apply SOOT's SPARK pointer analysis [27]. During the ICFG construction, SOOT models the effects of native methods by using abstract Java code and creates the corresponding control-flow edges. In addition, SOOT considers both explicit and implicit exceptions and treats the exceptional edges as normal control-flow edges. Finally, SOOT models thread creation and running as method calls by assuming that threads execute in a sequential order. How to build ICFGs to support multi-threading soundly, precisely and scalably for Java programs is a big challenge in its own right.

TAILOR has two main components, SCEXT and SCDFA. When extending SCs, we make use of the points-to information provided by SPARK to find the required object allocation sites object-sensitively. To perform SCDFA, we choose HEROS [10] as the IFDS solver for its BU and TD data-flow problems, because it can be easily plugged into the SOOT framework.

6 Evaluation

Program tailoring is designed to be sound for a program (with respect to its ICFG, as proved in Section 4), with useful precision and good scalability for large object-oriented programs. This section serves to evaluate the last two goals by answering three research questions:
RQ1: Is TAILOR useful to support program debugging and understanding, in practice?
RQ2: Is TAILOR useful to support program analysis, in practice?
RQ3: Is TAILOR scalable for large object-oriented programs, in practice?

To address RQ1 and RQ2, we conduct two real-world case studies. In one study, we demonstrate that TAILOR can assist a state-of-the-art slicing tool, a thin slicer [47] implemented in WALA [52], to simplify debugging and understanding tasks. In the other study, we demonstrate that TAILOR can enable a sophisticated pointer analysis algorithm, S-2OBJ [24],

Table 1 Program characteristics. For each program, the numbers are produced for both the application and library code, including only reachable classes, methods and statements by Spark [27].

Application	Description	#Classes	#Methods	#ByteCodes	LOC
ANTLR (2.7.2)	a recognizer and parser generator	2049	13,751	261,727	90,404
Avrora (1.7.117)	an assembly program simulator	3196	17,186	276,340	92,505
Eclipse (4.5)	IDE	2517	16,953	305,575	106,640
Apache™ FOP (0.20.5)	a formatting-objects processor	4681	28,105	492,686	171,087
JBoss AS (4.0.2)	an application server	4039	25,634	448,163	154,290
PMD (4.0)	a source code analyzer	4234	26,623	467,249	161,300
Apache Tomcat™ (8.0.24)	a Java Servlet container	3920	25,157	432,652	150,074

provided in a state-of-the-art pointer analysis tool for Java, Doop [14], to investigate the multi-object typestate (reflective) behavior in programs for which S-2Obj is unscalable as a whole-program analysis. In both studies, all *SCs* used are deduced from the results generated by state-of-the-art clients, Clara [9] and Solar [29], rather than injected manually.

To address RQ3, we perform a stress test on Tailor by using a large number of randomly generated *SCs*. Tailor scales well to relatively large Java programs, suggesting that program tailoring represents an attractive option as a practical tool.

Our experiments are carried out on an Xeon E5-2650 2GHz machine with 64GB RAM. We have selected seven large and diverse real-world Java programs under a large library, JDK 1.6.0_45, described in Table 1. For each program, with its bytecode representation generated by Soot [51], all the statistics are calculated by using Soot's Spark pointer analysis [27].

6.1 RQ1: Program Debugging and Understanding

Wala [52] includes a traditional slicer [21] and a thin slicer [47] (denoted TSlicer), with industry-strength implementations. Traditional slicing does not scale to large object-oriented programs due to the key bottleneck in handling of the heap [47]. Thin slicing [47] alleviates the bottleneck by including only producer statements that affect directly the values at a program point. This unsound design can facilitate program debugging and understanding [25, 50, 59].

By focusing on producer statements, TSlicer is surprisingly effective, by producing often small (or thin) slices quickly. In this study, we show that Tailor can make TSlicer even more effective, as highlighted in Figure 1, by exploiting the temporal order of statements in *SCs* to prune away more statements irrelevant to *SCs*. In addition, Tailor achieves this improved precision by trimming a program more efficiently than TSlicer does. These results are significant given TSlicer's unsoundness (even for G_{ICFG}) and well-tuned implementation in Wala, demonstrating clearly the practical benefits of our *SC*-oriented program tailoring.

To ensure a fair comparison between Tailor (implemented in Soot) and TSlicer (implemented in Wala), we have configured Soot to minimize the differences in the ICFGs constructed for a program in both frameworks. There are four main contributing factors: (1) pointer analysis, (2) reflection analysis, (3) exception handling, and (4) native code handling. Our bottom-line is to make sure that Tailor is never more precise than TSlicer in dealing with (1) – (4). For (1), we select Wala's *VanillaZeroOneCFA* option to perform its allocation-site-sensitive pointer analysis, which is more precise than Soot's Spark pointer analysis, as Spark merges some `java.lang.String` allocation sites for improving performance. For (2), we use Wala's *no_flow_to_casts* option to resolve reflective calls. In Soot, we have taken advantage of Solar [29] to perform reflection resolution in the same way. For (3) and (4), Wala and Soot model the same native methods in JDK 1.3 and handle both explicit and implicit exceptions with some differences. However, these differences do not affect the

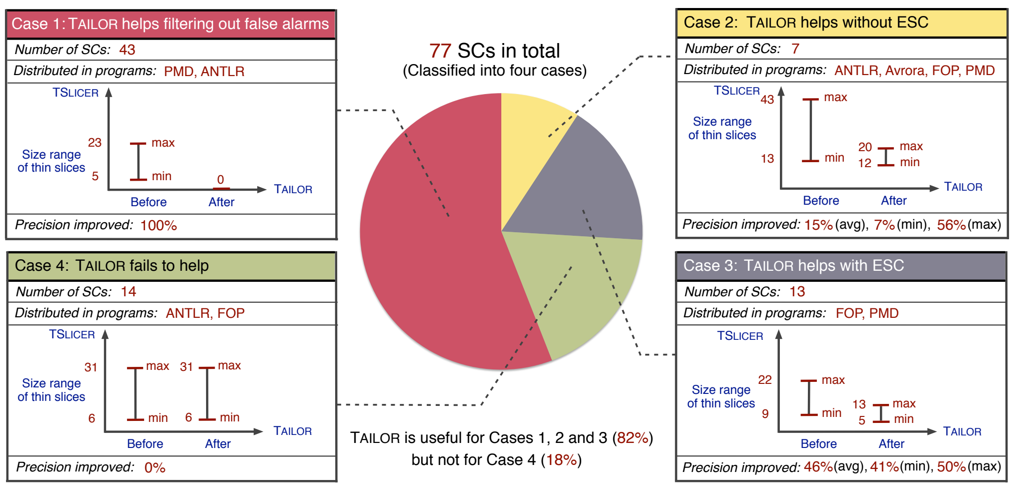


Figure 9 A case study demonstrating how TAILOR enables TSLICER to remove more *SC*-irrelevant statements based on the *SCs* deduced from the results reported by a typestate analysis, CLARA [9].

precision achievements obtained, validated by having inspected all results manually.

Both TAILOR and TSLICER trim a program based on the writer/OutputStream-related *SCs* deduced from the results reported by a state-of-the-art typestate analysis tool, CLARA [9]. TSLICER is run context-sensitively by using the last point in each *SC* as its slicing criterion with the variables of interest selected automatically. However, the results in the following cases are excluded: (1) CLARA crashes due to runtime exceptions in the case of `Eclipse`, `JBoss` and `Tomcat`, (2) TSLICER is unscalable for a criterion (within 1 hour), and (3) the size of a thin slice is less than 5 (small enough for human consumption). Finally, 77 *SCs* are considered in total, involving 66 errors and 11 program points (like the one in Figure 7) for our debugging and understanding tasks. All these *SCs* are provided in our artifact.

Results and Analysis Figure 9 presents the final results, with all the 77 *SCs* classified into four cases. In each case, we give the number of *SCs* included, the names of *SC*-contributing programs, the size ranges of TSLICER's thin slices before and after the *SC*-irrelevant statements detected by TAILOR are removed, as well as the minimum, maximum and average precision improvements achieved. Note that WALA's traditional slicer is unscalable for any *SC* (within 1 hour). Below we first analyze each case and then make some remarks.

Case 1. There are 43 *SCs* distributed in two programs, `ANTLR` and `PMD`. TSLICER has produced thin slices ranging from 5 to 23 statements, requiring further human analysis efforts. In contrast, TAILOR has produced only zero-sized tailored programs, declaring all the 43 errors as false alarms with respect to the given *SCs*. Furthermore, as CLARA is sound (with respect to G_{ICFG}), all the 43 errors are false alarms for the 43 reported locations.

Let us consider a *SC* from `PMD`. CLARA reports a "write after close" typestate error for a call to `writer.write()` at line 33 in class `net.sourceforge.pmd.renderers.TextRenderer`, together with four two-statement sequences of method calls leading to this potential error location: line 290 → line 325, line 290 → line 337, line 292 → line 325, and line 292 → line 337. These five calls are distributed in classes `net.sourceforge.pmd.PMD` and the afore-mentioned class `TextRenderer`. Given this four-sequence *SC* ending at line 33, TAILOR recognizes that all these sequences are infeasible since line 33 is not reachable from lines 325 and 337.

CLARA reports such false errors as it is partially context-sensitive and intraprocedurally but not interprocedurally flow-sensitive. This is a typical trade-off made by static analyses, which must, for example, reason about complicated typestate or protocol information as well. Otherwise, full context- and flow-sensitivity is unattainable scalably for large programs [9]. Unlike these client analyses, TAILOR reasons about the (un)reachability of a statement towards a *SC*, making it substantially more amenable to a fully context- and flow-sensitive analysis. TAILOR's success in this case is potentially replicable for other analysis tools [9, 29, 37].

Case 2. There are 7 *SCs* spread across `ANTLR`, `Avrora`, `FOP` and `PMD`. In this case, SCEXT is not useful. These *SCs* share the same characteristic as $SC_{\text{line }12}$: line 2→line 27→line 12 from our motivating example in Figure 2 (Section 2). For each *SC*, TAILOR has succeeded in removing some *SC*-irrelevant statements in some branches from TSLICER's thin slices.

Case 3. There are 13 *SCs* found in `FOP` and `PMD`. In this case, TAILOR will be ineffective unless SCEXT is turned on. We take a *SC* in `FOP` ending at line 101 in class `CommandLineStarter` to show how TAILOR can simplify debugging tasks for object-oriented programs enormously. Given this point of interest, TSLICER returns a thin slice containing nine statements, which are distributed in five classes, including `AWTStarter`, `PrintStarter` and `CommandLineStarter` in package `org.apache.fop.apps`, where the first two are the subclasses of the last one. The first statement of this *SC* resides in `CommandLineStarter`'s `run()` method, which is overridden in the two subclasses. During the SC extension, the object allocation site at line 522 in the `CommandLineOptions` class is found to be the sole object-sensitive context for `CommandLineStarter`'s `run()` method. Therefore, with this extension point, TAILOR is able to remove four irrelevant statements, line 94 in `AWTStarter`, line 91 in `PrintStarter`, and lines 494 and 508 in `CommandLineOptions`, from TSLICER's thin slice, saving a human a lot of debugging effort on navigating through many irrelevant classes unnecessarily.

Case 4. There are 14 *SCs* found in `ANTLR` and `FOP`. TAILOR fails to reduce TSLICER's thin slices any further. By including producer statements (and ignoring the others unsoundly), TSLICER has happened to eliminate all the *SC*- irrelevant statements removed by TAILOR.

Remarks. First, TAILOR has succeeded in making 82% (56%) of TSLICER's 77 thin slices smaller (empty), as shown in Figure 9, even though TSLICER is known to return small slices unsoundly. Second, TAILOR aims to eliminate *SC*-irrelevant statements. As discussed in Section 2 and elaborated in Case 3, removing several or even just one *SC*-irrelevant statement can save a lot of debugging effort, particularly for large object-oriented programs. Finally, TAILOR is fast, as compared with TSLICER in Figure 10. To make the analysis times for TAILOR visible, the longest analysis times spent by TSLICER on several *SCs* are depicted at the top-left corner. In Case 4, TAILOR is ineffective, but no harm is done, as TAILOR is fast. Without TAILOR, the practical benefits reaped from exploiting the temporal order in the other *SCs* in Cases 1 – 3 will be missed.

6.2 RQ2: Program Analysis

In this second case study, we demonstrate that TAILOR can be invaluable for pointer analysis (the foundation for virtually all other analyses). In particular, we show how TAILOR can enable today's most sophisticated pointer analysis algorithms, which are unscalable for a program, to perform a more focused and thus potentially scalable analysis to its specific

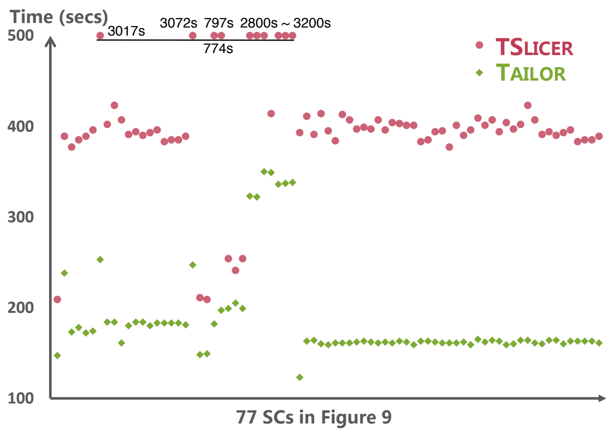


Figure 10 Efficiency of TAILOR vs. TSLICER.

parts that contain usually hard-to-analyze language features such as reflection [30]. For a Java program, a pointer analysis requires a reflection analysis to resolve part of its call graph representing reflective calls, and conversely, a reflection analysis requires the points-to information from a pointer analysis to discover the reflective targets at a reflective call site.

Reflection analysis finds many real-world applications, such as bug detection and security analysis [5, 11, 28, 31], in its own right. Despite recent advances [28, 29, 45], a sophisticated reflection analysis does not co-exist well with a sophisticated pointer analysis, since the latter is unscalable for large programs [28, 29, 31, 45]. If a scalable but imprecise points-to analysis is used instead, the reflection analysis may introduce many false call graph edges [29, 45], making its underlying client applications to be too imprecise to be practically useful.

We show how TAILOR can alleviate this problem. We choose DOOP [14], a state-of-the-art pointer analysis framework for Java, and focus on its three pointer analyses, 1-CALL, S-2TYPE and S-2OBJ. 1-CALL is usually the most scalable but most imprecise. S-2OBJ is the most precise but the most unscalable. S-2TYPE enjoys both precision and scalability, with its precision being always not better than S-2OBJ [24]. For reflection analysis, we choose SOLAR [29], a state-of-the-art reflection analysis built on top of DOOP. Our program analysis task is to investigate the multi-object typestate behavior at some reflective call sites as precisely as possible, by running SOLAR with S-2OBJ. If S-2OBJ is scalable for a program, we are done. Otherwise, we run SOLAR together with 1-CALL, and if that is unscalable, with S-2TYPE on the same program. If either is scalable, we treat SOLAR as a client analysis for producing the required *SCs*. To investigate the behavior at a reflective call site P, say, `m.invoke()`, we let its SC_P be all possible sequences of API method calls ending at P, such as `Class.forName()` $\to$ `c.getMethod()` $\to$ `m.invoke()`, found by SOLAR. Then we run S-2OBJ on the tailored program $\mathcal{T}(SC_P)$. If SC_P represents all possible sequences of method invocations for P, then $\mathcal{T}(SC_P)$ exhibits the same behavior at P as in the whole program (Theorem 3). Otherwise, a SC_P-specific behavior at P is analyzed.

For the seven applications listed in Table 1, S-2OBJ is only unscalable for `Eclipse`, `Tomcat` and `JBoss`, which will therefore be the focus of our second study. Figure 11 shows the number of reflective targets resolved at all call sites to `Class.newInstance()`/`Method.invoke()` (the most commonly used [28], in practice) in the application code of `Eclipse`, `Tomcat` and `JBoss`, by 1-CALL (the red dots) and S-2TYPE (the green dots from `JBoss` as 1-CALL is unscalable).

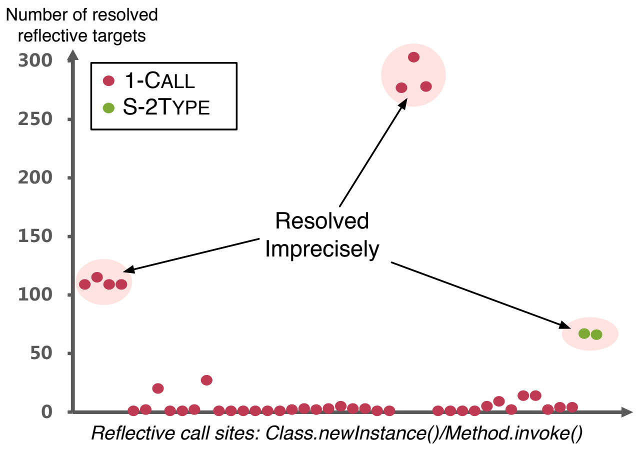


Figure 11 Reflection resolution for `Eclipse`, `Tomcat` and `JBoss`.

Table 2 A case study demonstrating how TAILOR enables a sophisticated pointer analysis to scale better by analyzing the reflective behavior at the specific parts of a program.

SC	Efficiency (Analysis Times)					Precision (Reflective Targets Resolved)				
						Number of Targets Resolved			Precision Improved Over	
	Before TAILOR			TAILOR	After TAILOR	Before TAILOR		After TAILOR		
	1-CALL	S-2TYPE	S-2OBJ		S-2OBJ	1-CALL	S-2TYPE	S-2OBJ	1-CALL	S-2TYPE
Eclipse-1	43m45s	129m39s	>10h	3m1s	37m25s	109	27	12	88.9%↑	55.6%↑
Eclipse-2				6m20s	>10h	109	28	—	—	—
Eclipse-3				2m29s	>10h	115	33	—	—	—
Eclipse-4				4m7s	>10h	109	28	—	—	—
Tomcat-1	42m0s	80m35s	>10h	4m11s	61m9s	303	116	1	99.7%↑	99.1%↑
Tomcat-2				3m49s	61m24s	278	1	0	100%↑	100%↑
Tomcat-3				3m42s	>10h	277	41	—	—	—
JBoss-1	>10h	171m38s	>10h	10m59s	6m14s	N/A	67	4	N/A	94%↑
JBoss-2				4m19s	6m12s	N/A	66	1	N/A	98.5%↑

There are a total of nine reflective call sites with a high number of reflective targets resolved, possibly imprecisely by 1-CALL or S-2TYPE, which result in a total of nine *SCs* deduced from their corresponding SOLAR analysis.

Table 2 shows our results. For the nine *SCs* given (in Column 1), TAILOR enables S-2OBJ to run scalably on the tailored programs obtained for five *SCs*, *Eclipse-1*, *Tomcat-1*, *Tomcat-2*, *JBoss-1* and *JBoss-2* (in 6 – 62 minutes) with significant precision improvements in terms of the number of reflective targets reduced. Without TAILOR, no existing reflection analyses [28, 29, 31, 45] can achieve such precise results automatically for their corresponding reflective calls. Note that traditional slicing [21] is unscalable given the last points in the nine *SCs* (out of (64GB) memory after 2 hours each) while thin slicing [47] (designed for program debugging and understanding only) is unsound and is thus non-applicable in this setting.

S-2OBJ remains unscalable for four *SCs*, *Eclipse-2*, *Eclipse-3*, *Eclipse-4* and *Tomcat-3*, partly because S-2OBJ is precise but slow (being basically 2-object-sensitive [24]) and partly because reflection-rich object-oriented programs such as `Eclipse` and `Tomcat` are difficult to analyze both precisely and scalably, despite recent advances [28, 29, 45]. TAILOR produces the tailored programs for the nine *SCs* in less than 11 minutes each (Column 5), making it possible for the five out of nine *SCs* to be analyze precisely than before (Columns 7 – 11).

Let us examine the *SC* denoted by *Tomcat-1*. Under 1-CALL, SOLAR identifies an imprecisely resolved `newInstance()` call (at line 268 in class `Bootstrap`) with 303 targets.

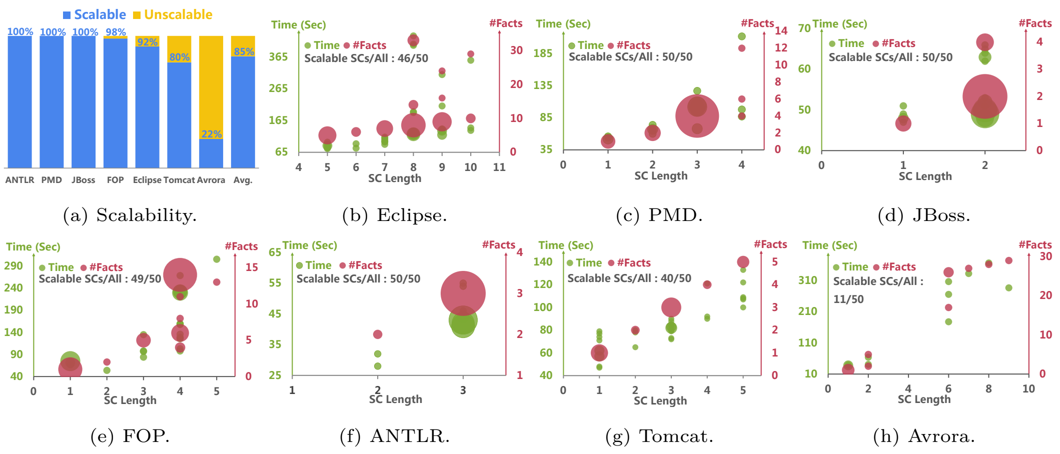

(a) Scalability. (b) Eclipse. (c) PMD. (d) JBoss.
(e) FOP. (f) ANTLR. (g) Tomcat. (h) Avrora.

Figure 12 Scalability of SCDFA on 50 randomly generated *SCs* per program (with a 10-minute budget per *SC*). “#Facts” represents the number of data-flow facts in $\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})$.

For this call, Solar reports four related `Class.forName()` and `loadClass()` calls, at lines 1232, 1265, 1299 in class `WebappClassLoaderBase` and line 265 in class `Bootstrap`, forming a four-sequence *SC* (of length 2 each). When analyzing `Tomcat` as a whole, S-2Obj does not terminate in 10 hours. Given *Tomcat-1*, Tailor produces a tailored program, $\mathcal{T}$(*Tomcat-1*), in about 4 minutes. Given $\mathcal{T}$(*Tomcat-1*), S-2Obj finishes in about 61 minutes, enabling Solar to resolve the `newInstance()` call (line 268) precisely to be class `Catalina` as its target.

We have inspected manually the five *SCs* scalably analyzed and found that no true reflective targets are missed for the call sites at their last points. For *Tomcat-2*, Column 9 has a 0, because its last point, the `newInstance()` call (at line 595 in class `RewriteValue`), reported by 1-Call is not reachable (from `main()`) and thus filtered out by S-2Obj.

6.3 RQ3: Scalability

Tailor has two main components, SCEXT and SCDFA, with the latter dominating the total analysis time. We perform a stress test to investigate how well SCDFA scales, in practice. For each of our seven applications listed in Table 1, we select randomly 50 statements as the potential points of interest, then apply SCEXT to these points, and finally, run SCDFA on the 50 extended *SCs*. For a total of seven applications, a total of 350 *SCs* are generated.

Figure 12 gives the final results. According to Figure 12a, SCDFA is scalable for 296 (85%) out of the 350 *SCs* generated (with a 10-minute budget per *SC*). Figures 12b – 12h provide more details. For each application, we give the number of scalable *SCs*, and for each scalable *SC*, its length (in the x-axis), its analysis time (in the left y-axis) and the maximum number of data-flow facts (suffixes) reaching ENTRY, i.e., $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$ (in the right y-axis). In the legend for each *SC*, there is a green dot representing its analysis time and a red dot representing its $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$. The data plotted for a fixed *SC* length can be understood as follows. First, if there are n *SCs* with the same analysis time ($|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$), then its green (red) dot is n times as large as the green (red) dot in the legend. Second, the analysis time of a *SC* always increases as $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$ increases (allowing one to find its associated red dot given a green dot). This is expected as the time complexity of SCDFA is $O(ED^3)$, where D is the size of the data-flow facts used, i.e., $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$. In general, $|\mathsf{PANTI}^{\text{out}}(\mathsf{ENTRY})|$ increases as the length of its underlying *SC* increases. However, this is not absolute, as the length of a *SC* is defined to be the length of its longest sequence (with the other shorter sequences ignored).

Finally, we explain why SCDFA is unscalable for many *SCs* in `Tomcat` and `Avrora` and discuss some possible solutions. For `Tomcat`, SCDFA is unscalable for 10 *SCs*, because many of their extension points introduced by SCEXT could have been avoided if a more precise pointer analysis (than Soot's Spark pointer analysis) is used.

For `Avrora`, SCDFA is unscalable for 39 *SCs*, due to a special programming pattern used in this (simulator) application. One factory class `cck.util.ClassMap` is used to create all its simulation and platform classes, e.g., `SensorSimulation`, residing in a total of 13 packages. As a result, most of the SC extension points introduced by SCEXT happen to land in this factory class. In addition, there are many object allocation sites for this class in the program, making all of them eligible as SC extension points and consequently increasing the number of data-flow facts used, i.e., $|\mathsf{PANTI}^{\mathrm{out}}(\mathsf{ENTRY})|$. One possible improvement is to make SCEXT pattern-aware to avoid some SC extensions that would otherwise be introduced.

6.4 Limitations

We observe that program tailoring scales better than program slicing, as the overall design of tailoring (with its object-sensitive conceptualization of SCEXT and its IFDS formulation of SCDFA laid out in Sections 1.1 and 1.2) is more amenable to efficient implementation with useful precision, as validated with our prototyping system, Tailor. However, there are still spaces for performance improvement. According to our experimental results presented in Section 6.3, a more intelligent SCEXT is needed to deal with programs such as `Avrora` more effectively. Applying a pattern-aware pre-analysis to recognize some unscalability-inducing SC extension points may be a viable solution worth trying in future work.

Tailor is practically useful in program debugging and understanding as well as program analysis, as demonstrated with two case studies. However, Tailor is expected to be more effective if we can avoid introducing irrelevant statements in loops. As explained in Section 4.2, SCEXT presently gives up a SC extension point inside a control-flow cycle but may miss an opportunity for avoiding infeasible paths at this point, with a tradeoff made favoring scalability over precision. One possible improvement is to leverage slicing to remove irrelevant statements that do not produce any data dependence for the statements in a *SC*. How to combine tailoring and slicing scalably is non-trivial but will be an interesting future work.

7 Related Work

In addition to the related work mentioned earlier, we review some other related research.

Program Slicing. There are dozens of slicing techniques proposed, including amorphous slicing [19], parametric slicing [15], interface slicing [6] and specification slicing [4]. However, none of these is close to program tailoring, as tailoring is the first to exploit a sequential criterion and designed to scale for today's large object-oriented programs. For past slicing techniques and their design goals, we refer to some survey articles [8, 20, 43, 49]. Below we examine the most closely related ones, by focusing on their connections with this work.

Traditional slicing [53] and thin slicing [47] have been introduced and compared in Section 2 and evaluated in Section 6. In general, most existing static slicing methods [20, 43, 49] make use of the traditional slicing criterion (defined in [53] and used in Section 2) to express their points of interest but may slice a program differently according to different goals. For example, thin slicing [47] considers only producer statements that may have direct effects at a point. Although unsound, thin slicing may exclude many distracting statements, easing significantly program debugging and understanding (which is its goal aimed for).

Program tailoring focuses on a totally different criterion, known as sequential criterion, which captures the temporal order of invocation sequences of related methods naturally inherent in a program. As explained in Section 2 and demonstrated in Section 6, such ordering information enables tailoring to improve the effectiveness of a modern thin slicing tool, and potentially other slicing tools, e.g., a recently proposed value slicer [26].

Path slicing [23] takes as input a program path (or trace) to a target point and tries to eliminate the statements that are irrelevant towards the reachability of the point. Given a SC_P lying on a path with P as a target point, path slicing may remove all points in SC_P except P as long as they do not affect the reachability, i.e., feasibility of the path to P (from `main()`). In contrast, program tailoring focuses on determining the reachability of statements towards a SC so that the temporal order specified in the sequences in SC is respected.

Program chopping [22,39] considers two sets of variables, one at a source point and one at a sink point, as its slicing criterion. It identifies the statements that transmit the effects from the source to the sink, by applying a forward slicing at the source and a backward slicing at the sink. Tailoring is different in three aspects. First, chopping focuses on two points but tailoring focuses on the temporal order specified by a sequential criterion consisting of possibly many statement sequences of arbitrary lengths (sinking at the same point). Second, chopping, which relies on traditional slicing, does not scale to large object-oriented programs. Finally, SCEXT is unique as it enables tailoring to work effectively, in practice. In fact, these differences are also what distinguish tailoring from other slicing techniques [8,20,43,49].

API Protocol Analysis. We consider typestate analysis [9, 16, 34] as a special kind of API protocol analysis as the abstract states of an object are usually affected by API calls. API protocol analysis [7,37] reports related invocation sequences of API methods, indicating a kind of semantic (or ordering) information in the program, which is ignored by program slicing. Such API invocation sequences can be specified manually or mined automatically [1,3,17,36,42,60]. SCEXT can be seen as a new method for mining protocols in object-oriented programs automatically, but at a coarser granularity, based on the object-sensitive calling contexts of a method. We expect Tailor to become more effective when longer sequences (with richer semantic information) are considered as *SCs*.

Pointer Analysis. SCEXT exploits the concept of object-sensitivity [33] to extend *SCs* in order to avoid infeasible paths that would otherwise be introduced by SCDFA. We refer to [44] on why object-sensitivity is more effective than call-site-sensitivity in representing calling contexts when analyzing object-oriented programs context-sensitively. Over the years, many whole-program pointer analyses for Java have been developed [24,27,44,54,55]. In Section 6.2, we show how Tailor enables a precise whole-program pointer analysis [24] to scale better on a SC-relevant part of the program in order to analyze the reflection behavior at a program point affected by SC precisely. In principle, the reflective behavior at a program point can also be answered by raising points-to queries on-demand instead. In practice, however, existing demand-driven pointer analyses for Java [32,40,41,46,58] either ignore hard language features such as reflection and dynamic class loading or assume that they have been handled by a pre-analysis (e.g., when G_{ICFG} is built). In fact, how to analyze such hard language features is a big challenge in its own right [30], despite recent advances on static handling of reflection and dynamic class loading [5,28,29,31,35,45,56,57]. In this setting, we are not aware of any SC-aware demand-driven pointer analysis, not to mention its scalability to large object-oriented programs flow- and context-sensitively.

8 Conclusions and Future Work

This paper brings a new dimension to program slicing by introducing a sequential criterion. The temporal ordering constraints that appear in a sequential criterion, which can be, e.g., one or several sequences of API usage calls leading to a program point, are naturally inherent in many real-world applications, but have not been exploited in program slicing before. Accordingly, we propose program tailoring, a new technique to trim a program by a sequential criterion soundly (with respect to a given ICFG) and scalably (for reasonably large object-oriented programs). Regarding this new research work — program tailoring, in theory, we have formalized it, proved its soundness (with respect to a given ICFG), and discussed its advantages and limitations. In practice, we have produced a soon-to-be released open-source implementation, Tailor, and demonstrated its usefulness for improving the effectiveness of existing slicing techniques in program debugging and understanding and for supporting program analysis with large Java programs due to its good scalability.

Program tailoring builds on a natural connection with several research fields: program slicing, API protocol (or specification) mining, and program analysis. Therefore, a lot of interesting future work is anticipated to investigate their interplay. One possibility is to combine tailoring and slicing to eliminate irrelevant statements that cannot be eliminated by either alone scalably for a given task. Another is to combine API protocol analysis with SC extension to enable the latter to exploit richer semantic information available. Finally, program tailoring also provides new opportunities for developing program analyses (e.g., pointer analysis) that are focused and partial, paying closer attention to specific parts of the program, where some hard language features need to be analyzed precisely and scalably.

Acknowledgements. The authors wish to thank the anonymous reviewers for their valuable comments.

References

1 Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. FSE '07.

2 Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. SAS '02.

3 Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. POPL '02.

4 Min Aung, Susan Horwitz, Rich Joiner, and Thomas Reps. Specialization slicing. *ACM Trans. Program. Lang. Syst.*, 2014.

5 Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and android intents. ASE '15.

6 Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. ICSE '93.

7 Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. ECOOP '09.

8 David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers.*, 2004.

9 Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. ICSE '10.

10 Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. SOAP '12.

11 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. ICSE '11.

12 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. OOPSLA '09.

13 Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. PLDI '02.

14 DOOP. `http://doop.program-analysis.org`.

15 John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. POPL '95.

16 Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 2008.

17 Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. FSE '08.

18 GrammaTech. GrammaTech static analysis. `http://www.grammatech.com`.

19 Mark Harman and Sebastian Danicic. Amorphous program slicing. IWPC '97.

20 Mark Harman and Rob Hierons. An overview of program slicing. *Software Focus.*, 2001.

21 Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 1990.

22 Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. SIGSOFT '94.

23 Ranjit Jhala and Rupak Majumdar. Path slicing. PLDI '05.

24 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. PLDI '13.

25 Andrew J. Ko and Brad A. Myers. Extracting and answering why and why not questions about Java program output. *ACM Trans. Softw. Eng. Methodol.*, 2010.

26 Shrawan Kumar, Amitabha Sanyal, and Uday P. Khedker. Value slice: A new slicing concept for scalable property checking. TACAS '15.

27 Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. CC '03.

28 Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for Java. ECOOP' 14.

29 Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. SAS'15.

30 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 2015.

31 Benjamin Livshits, John Whaley, and Monica Lam. Reflection analysis for Java. APLAS'05.

32 Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with CFL-reachability. CC '13.

33 Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 2005.

34 Nomair A. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. OOPSLA '08.

35 Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. ACSC '05.

36 Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. FSE '09.

37 Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. ICSE '12.

38 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL '95.

39 Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. SIGSOFT '95.

40 Lei Shang, Yi Lu, and Jingling Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation. ASE '12.

41 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. CGO '12.

42 Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. ISSTA '07.

43 Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 2012.

44 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. POPL '11.

45 Yannis Smaragdakis, George Kastrinis, George Balatsouras, and Martin Bravenboer. More sound static handling of Java reflection. APLAS '15.

46 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. PLDI '06.

47 Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. PLDI '07.

48 Robert Tarjan. Depth first search and linear graph algorithms. *SICOMP*, 1972.

49 Frank Tip. A survey of program slicing techniques. *J Program Lang*, 1995.

50 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. PLDI '09.

51 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. CASCON '99.

52 WALA. T.J. Watson libraries for analysis. `http://wala.sf.net`.

53 Mark Weiser. Program slicing. ICSE '81.

54 John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. PLDI '04.

55 Xiao Xiao and Charles Zhang. Geometric encoding: Forging the high performance context sensitive points-to analysis for Java. ISSTA '11.

56 Jingling Xue and Phung Hua Nguyen. Completeness analysis for incomplete object-oriented programs. CC '05.

57 Jingling Xue, Phung Hua Nguyen, and John Potter. Interprocedural side-effect analysis for incomplete object-oriented software modules. *Journal of Systems and Software*, 2007.

58 Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. ISSTA '11.

59 Sai Zhang and Michael D. Ernst. Which configuration option should I change? ICSE '14.

60 Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. ECOOP '09.