Precise Data-flow Summaries with Synchronized Pushdown Systems

by

David Seekatz

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Static data-flow analysis is a method of reasoning about program values without executing the program. A data-flow analysis that is context-sensitive, field-sensitive, flow-sensitive, and alias-aware can precisely and soundly answer points-to queries (e.g. what heap objects can variable `v` reference at line `n`?). For large modern software systems, answering these queries is time- and resource-intensive due to the size and complexity of library code. If an application is heavily library-dependent, a typical data-flow analysis spends the majority of its computation time analyzing library code. However, library code does not change each time an application is compiled, so library data-flow can be summarized and re-used to efficiently analyze application code.

This thesis details a novel method of computing precise data-flow summaries for libraries using Synchronized Pushdown Systems (SPDS) and heap graphs, which we implement in a tool called BluJ. To construct the summaries, BluJ instantiates an SPDS instance at every point that an object may enter the library. BluJ then computes forward data-flow reachability information for each SPDS instance and uses a heap graph to model the heap relationships between the respective objects tracked by each SPDS instance. The resulting data-flow summaries take into account any aliasing that occurs in the library, they are context- and field-sensitive, and they are flow-sensitive with the exception of a few rare cases. The generated summaries are bi-directional and can be consumed by either a forward or backward client analysis. We implement our summaries within the Boomerang points-to analysis framework

and compare the relative performance and precision of BOOMERANG with and without our generated summaries.

Preface

This thesis contains wholly original work, and we intend to publish portions of this thesis in the future. I formulated the concept and theory of this work, designed and implemented the software component, and carried out the empirical evaluation. Karim Ali was the supervisory author and contributed to the concept and theory of this work, as well as the empirical evaluation design and thesis edits.

Acknowledgements

I would like to thank my supervisor, Karim Ali, for his patience, guidance, and support over the last few years. In addition to his invaluable contributions to this work, he provided focus and direction in the early stages when I was unsure of my research topic, and he taught me the research skills I needed to be successful. I would also like to thank Ondřej Lhoták and Frank Tip for providing me with feedback and advice early in my Master's, which was invaluable to me at the time.

Thank you to my amazing partner, Sarah, and our wonderful daughter for providing the support and encouragement I needed to stay grounded and motivated over the course of my graduate studies.

Thank you to my friends and colleagues in the Maple lab for the productive discussions we had and the mentorship you provided, and thank you to everyone else at the University of Alberta who helped me along the way.

Thank you to the folks at the Amazon Prime Video Automated Reasoning team, especially Bor-Yuh Evan Chang, Daniel Schoepe, and Pauline Bolignano, for providing me with the chance to spend six months learning as an intern. Thanks to my time with the team, I was able to return to my Master's studies equipped with a greater breadth of static analysis knowledge and several new ideas for this work.

I would like to thank the University of Alberta Department of Computing Science and Faculty of Graduate Studies and Research for partially funding this work. Additionally, I acknowledge that this work was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

I respectfully acknowledge that during the completion of this thesis I have lived and worked on Treaty 6 and Treaty 7 territory, the traditional lands

of the nêhiyaw, Dené, Anishinaabe, Nakota Sioux, Tsuut'ina, and Niitsitapi peoples, as well as a homeland of the Métis.

Contents

List of Tables

List of Figures

Glossary

API
: application programming interface

BSA
: BluJ summary adapter

CFG
: control-flow graph

ESG
: exploded supergraph

HGASS
: heap graph augmented synchronized summary

IDE
: interprocedural distributive environment

IDE
: integrated development environment

IFDS
: interprocedural finite distributive subset

IR
: intermediate representation

JAR
: java archive

JSON
: JavaScript object notation

JVM
: Java virtual machine

MFP
: maximal fixed point

MOP
: meet-over-all-path

PDS
: pushdown system

SPDS
: synchronized pushdown system

TAL
: tree-adjoining language

Chapter 1

Introduction

Static data-flow analysis leverages program semantics to reason about the flow of information in a program at compile-time, without running the program. Optimizing compilers use static data-flow analysis to determine which program transformations they can safely apply [51], integrated development environments (IDEs) use static data-flow analysis to detect possible bugs [37], and software security tools use static data-flow analysis to verify that programs are free from data leaks and injection attacks [7]. Unfortunately, computing exact semantic properties of arbitrary programs is undecidable, per Rice's theorem [43], so static data-flow analysis techniques must either *over-* or *under-*approximate the true runtime behaviour of programs.[1]

Generally, compilers and security tools must over-approximate program behaviour because they aim to provide guarantees about the programs they analyze, i.e., that a compiled program is semantically equivalent to its original source code, and that a program is free from security vulnerabilities, respectively. An analysis that provides these guarantees is *sound*, but it is also necessarily *imprecise* due to the undecidability of static analysis. This imprecision manifests in the form of *false positives*. For a security analysis, this means that it may report data leaks that may not actually occur at run time. For a compiler, this means that it may avoid applying an optimization that would be perfectly safe in practice.

Because data-flow analyses reason about the flow of information in a pro-

[1] Hereafter, we use "data-flow analysis" to refer to "static data-flow analysis"

gram, they are generally *flow-sensitive*, meaning they compute information at every individual program statement. For large programs, precise flow-sensitive analyses are not always feasible due to constraints on memory usage or running time. For example, a flow-sensitive points-to analysis calculates the set of possible heap allocations that every program variable can reference at each individual program statement. It is therefore difficult to efficiently compute flow-sensitive points-to information for large programs with many statements and many heap-allocated objects.

The difficulty of obtaining flow-sensitive information for large, complex programs motivates the development of techniques to analyze data-flow in *partial* programs. The goal of partial program data-flow analysis is to decrease the scale of an analysis problem by analyzing individual compilation units and then composing the results of the individual analyses. Most partial program data-flow analyses persist their analysis results by writing data-flow summaries to disk, allowing subsequent analysis runs to re-use the summaries. Consequently, this analysis approach is most applicable to compilation units that see a high amount of re-use across multiple programs. For the Java language, the highest-use compilation units are *libraries* such as the Java Standard Library or those available through repositories such as Maven Central.[2] Java libraries are often significantly larger and more complex than the application code that uses them, motivating extensive research on generating analysis summaries for Java libraries [1], [2], [6], [45], [60], [61].

There are inherent difficulties in generating precise library data-flow summaries for Java. In the absence of application code, the library only has access to type information for objects created in the application as declared in the signatures of its application programming interface (API) methods, making it difficult to accurately resolve virtual method calls. Application classes can override library methods, so library summaries must accurately model the flow of objects back into the application at overridable method calls. Because Java programs often exhibit deeply nested (and sometimes recursive) field accesses on objects, library-only analyses must track the flow of fields of objects passed

[2] `https://mvnrepository.com/repos/central`

into the library in addition to the objects themselves. Analyses that track fields of objects are called *field-sensitive*. The current state-of-the-art for Java library data-flow summaries uses k-limited access paths to abstract fields of objects at library entry and exit points. k-limited access paths only track data-flow up to k field accesses on each object, and they are field-insensitive for access paths with length greater than k.

This thesis introduces BLUJ, a tool that improves the precision of library data-flow summaries by tracking the flow of *all* fields of objects passed into the library, including access paths of arbitrary depth, using a combination of precise field abstractions: PDSs [11] and heap graphs [14]. BLUJ instantiates a synchronized pushdown system (SPDS) [55] for every base object that enters the library and additionally maintains a heap graph that tracks any field reads that the library performs on that object. We target flow-sensitive points-to analyses as the primary consumer of our library summaries, due to the large class of potential client analyses that require flow-sensitive points-to information and the difficulty of obtaining whole-program flow-sensitive points-to information for large Java programs.

We aim to determine whether data-flow summaries based on synchronized pushdown systems and heap graphs are feasible in practice. Specifically, we (1) assess whether they can be computed efficiently and (2) whether they can be efficiently and precisely integrated into an existing whole-program points-to analysis. For (1), we evaluate whether BLUJ can summarize enough library classes to capture at least 50% of application-to-library data-flows within the span of 12 hours. For (2), we integrate BLUJ's summaries into the BOOMERANG points-to analysis framework [56] and compare BOOMERANG to BOOMERANG+BLUJ on the basis of analysis time, memory consumption, soundness, and precision. We determine that BLUJ's integration with BOOMERANG is efficient and precise if it satisfies the following criteria:

- On average, BOOMERANG+BLUJ takes less time to analyze the same target program set as BOOMERANG.
- On average, BOOMERANG+BLUJ uses less maximum memory to analyze

the same target program set as BOOMERANG.

- BOOMERANG+BLUJ is sound with respect to BOOMERANG, i.e., it reaches the same set of application states as BOOMERANG when analyzing the same target program set.

- BOOMERANG+BLUJ does not introduce more than 25% additional reached application states when compared to BOOMERANG on the same target program set.

We show that BLUJ can compute its summaries efficiently, but integration of BLUJ's summaries into an existing analysis is not efficient.

This thesis includes the following contributions:

- We provide a novel technique for generating precisely field-sensitive library data-flow summaries based on synchronized pushdown systems and heap graphs.

- We present a process for integrating our data-flow summaries into an existing data-flow analysis framework.

- We present BLUJ, a tool that generates our data-flow summaries for Java library classes and integrates them into the BOOMERANG points-to analysis framework.

- We provide an empirical evaluation of precision and performance effects that result from using our data-flow summaries in a whole-program points-to analysis.

This thesis is organized as follows. Chapter 2 provides relevant background on data-flow analysis, call graphs, and points-to analysis, as well as pushdown systems and their relation to data-flow analysis. Chapter 3 discusses related work in the area of partial program analysis and library data-flow summaries. Chapter 4 describes how we use synchronized pushdown systems and heap graphs to summarize library data-flow in the absence of application code. Chapter 5 outlines how BLUJ writes persistent summaries and integrates them

into a whole-program points-to analysis. Chapter 6 provides an evaluation on the basis of soundness, precision, and performance for a points-to analysis that integrates BLUJ's summaries. Chapter 7 reviews the contributions and conclusions of this thesis and discusses potential directions for future work.

Chapter 2

Background

Data-flow analysis first became an area of research interest in the late 1960s as a technique to perform aggressive optimizations in compilers. Early research on data-flow analysis was often tailored to specific compiler optimizations, such as common subexpression elimination [16] and reaching definitions [3]. These analysis problems both fall into the category of classical bit-vector problems, which take their name from the fact that they have solutions that are efficiently computable using bit-vectors. In 1973, Kildall [28] introduced the first generic approach to data-flow analysis, formalizing the bit-vector problems in a single framework.

2.1 Intraprocedural data-flow analysis

Kildall's framework applies to intraprocedural data-flow, meaning that the analysis restricts the tracking of data-flow to the confines of procedure boundaries. The framework operates over a control-flow graph (CFG) $G_c = (S, E, s_0)$, where S is the set of statements in the program, $E \subseteq S \times S$ is the set of edges of the CFG, and s_0 is the program entry statement. A CFG represents the execution order of statements in a program. For a CFG edge $(s, s') \in E$, s' is called the control-flow *successor* of s, and s is called the control-flow *predecessor* of s'. The framework additionally requires a domain of data-flow facts $\mathbb{D}$ that form a semilattice with a meet operator $\sqcap : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, and a *flow function* $f : S \times \mathbb{D} \to \mathbb{D}$ that maps a pair (s, D) to D' for $s \in S$ and $D, D' \in \mathbb{D}$. The flow function f can be viewed as an input/output relation on each state-

ment $s \in S$—given a set of input data-flow facts D before statement s, the flow function returns a new output set of data-flow facts D' after statement s. In other words, each $s \in S$ naturally yields a function $f(s) : \mathbb{D} \to \mathbb{D}$.

Kildall's framework computes the meet-over-all-path (MOP) solution to a data-flow problem, i.e., the set of data-flow facts $D \in \mathbb{D}$ that holds at some statement s. Let $PATH(s)$ be the set of all paths from the initial statement s_0 to s in the CFG, and let $\perp$ be the least element of $\mathbb{D}$. Then, the MOP solution for a statement s is:

$$MOP_s = \bigsqcap_{(s_0,\ldots,s) \in PATH(s)} (f(s) \circ \ldots \circ f(s_0))(\perp)$$

The MOP solution is computable for data-flow problems that are *distributive*. A problem is distributive if it defines a flow function f such that for all $s \in S$ and $D, D' \in \mathbb{D}$, $f(s, (D \sqcap D')) = f(s, D) \sqcap f(s, D')$. Some practical data-flow problems, such as constant propagation, are non-distributive. In 1977, Kam and Ullman [25] generalized Kildall's work to the class of *monotone* data-flow problems, which includes the distributive data-flow problems as well as some non-distributive problems like the aforementioned constant propagation. A problem is monotone if it defines a flow function f such that for all $s \in S$ and $D, D' \in \mathbb{D}$, $D \leq D' \implies f(s, D) \leq f(s, D')$, where $\leq$ is the ordering operator for the semilattice $\mathbb{D}$. For the class of monotone data-flow problems, the MOP solution is not computable. Instead, the monotone data-flow framework computes the maximal fixed point (MFP) solution, which is an over-approximation of the MOP solution. For problems where the flow functions are distributive, the monotone framework's MFP solution is equivalent to the MOP solution.

2.2 Interprocedural data-flow analysis

Interprocedural analysis tracks data-flows through procedure calls and returns in addition to tracking data-flow within individual procedures. Early research on interprocedural analysis emerged from attempts to perform whole-program

optimizations in the IBM PL/I compiler [57]. Generally, interprocedural analysis requires a call graph [4] to keep track of referencing relationships between procedures. A call graph is a mapping from each program statement (or "call site") that invokes a procedure to each potential target of the procedure call.

For object-oriented languages such as Java, statically resolving the exact target of a procedure call is, in general, undecidable. However, there are a large number of effective approximate algorithms for computing static call graphs, e.g., [8], [17], [35], [59]. The choice of call graph construction algorithm significantly affects the efficiency and precision of an interprocedural data-flow analysis. An imprecise call graph can lead to a large number of spurious data flows (negatively impacting precision) and large sets of data-flow facts (negatively impacting performance). Conversely, computing a very precise call graph may not be feasible for large programs due to constraints on memory usage or running time.

2.3 Data-flow dimensions

Data-flow analyses are characterized by the ways in which they propagate data-flow facts with respect to control flow, procedure calls/returns, and heap accesses [46]. Each data-flow analysis has a level of sensitivity in each of the aforementioned dimensions, which we discuss in the following subsections.

2.3.1 Flow-sensitivity

A flow-sensitive data-flow analysis is an analysis that takes into account the control flow of a program when computing data-flow facts. A flow-sensitive analysis maintains data-flow facts at each location along a program's control flow graph. A flow-insensitive analysis instead computes data-flow facts that must hold at **all** program locations. Flow-sensitive analyses are inherently more precise than their flow-insensitive counterparts because they are able to perform strong updates to remove data-flow facts that no longer hold at a particular statement. Most distributive data-flow problems have an efficient flow-sensitive formulation, including the classical bit-vector problems.

Flow-insensitive analysis is most applicable to problems where the flow functions are non-distributive and computing per-statement information is computationally expensive, such as points-to analysis. Widely-used instantiations of flow-insensitive points-to analyses include Andersen's set-based formulation [5] and Steensgaard's equality-based formulation [58]. Recently, some advancement has been made on efficient flow-sensitive points-to analysis, e.g. [21], [33], [56], and we incorporate many of the ideas of Späth et al. [56] into BLUJ.

2.3.2 Context-sensitivity

A context-sensitive data-flow analysis is an interprocedural analysis where the information computed for a procedure depends on its calling context. Conversely, a context-insensitive analysis makes no distinction between possible calling contexts of individual procedures. A context-sensitive analysis uses a call graph to determine the possible calling contexts for each procedure that it analyzes. Two classical and influential approaches to context-sensitive data-flow analysis are the call-strings approach and the functional approach [52].

The call-strings approach attempts to approximate the runtime stack by maintaining a "string" of calling contexts for each procedure call. When the analysis enters a procedure call, it appends the calling context to the call-string. When it exits that procedure, it removes the context, ensuring calls and returns are properly matched. To guarantee that the analysis terminates, call-strings are bound to a finite length chosen by the analysis designer. The largest downside to the call-string approach is that it cannot effectively model recursive procedure calls, making it infeasible for languages where call graphs are likely to contain many cycles [34].

The functional approach attempts to summarize the input-output relations for each procedure. Once the analysis finishes computing the summary for a given procedure, it does not need to analyze the procedure again. Instead, when the analysis encounters an invocation of the procedure elsewhere in the program, it applies the summary to the current calling context. Usually, the analysis begins by analyzing the set L of procedures that do not invoke any

other procedures, then continues recursively analyzing callers of L until the analysis reaches a fixed-point. This is commonly referred to as "reverse invocation order" or "bottom-up" analysis. An advantage of the functional approach is that it effectively handles recursive procedure calls. However, it is difficult to create procedure summaries that are sufficiently general to be applied at any possible calling context, yet lightweight enough to be applied efficiently. Many modern context-sensitive analysis techniques are variations of the functional approach, e.g., [13], [19], [40], [47].

2.3.3 Field-sensitivity

In object-oriented languages, programs read and write to heap memory by executing loads and stores on fields of objects. A precise data-flow analysis must therefore be field-sensitive and implement an abstraction that tracks object fields.

2.3.3.1 Field-based abstraction

A field-based analysis does not distinguish between field accesses on different objects of the same type. Instead, for an object o of type T and a field access $o.f$, the analysis abstracts the object by its type and treats the field access as $T.f$. If there is a field access $o'.f$ on a different object o' that also has type T, the analysis again treats the field access as $T.f$ and does not distinguish between the objects o and o'. The advantage of the field-based approach is that it has a relatively small data-flow domain $\mathbb{D} = V \cup F$, where facts are either variable names or field names. The small data-flow domain restricts the size of sets of data-flow facts that the analysis needs to propagate, and in many cases translates to an efficient analysis. The analysis also grows less precise as the number of field accesses on different objects of the same type increases. Despite its limitations, the field-based abstraction is significantly more precise and generally more efficient than field-insensitive abstractions [22].

2.3.3.2 k-limited access paths

Access paths describe a base variable, v, followed by a finite sequence of field references f^* of up to length k. The data-flow domain of an analysis that uses k-limited access paths is then $\mathbb{D} = V \times F^*$. Access paths offer more precision than the field-based abstraction, at the cost of a larger data-flow domain. Additionally, for data-flow facts with field access chains longer than k, the analysis becomes field-insensitive. In practice, an analysis developer may need to select an appropriate value of k that balances precision and scalability. A high value for k yields a more precise analysis, but it also leads to an explosion in the size of the data-flow domain [31]. Access paths have proven to be a useful abstraction for alias analysis, as aliasing relationships can be represented as equivalence classes of access paths [26].

2.3.3.3 Access graphs and heap graphs

Access graphs remove the need for k-limiting by representing field accesses as a (possibly infinite) regular set of paths [27]. Analyses that use this abstraction typically capture the set of access paths in a graph $G = (v, f_0, F, E)$, where $v \in V$ is the root variable of each access path, F is the set of fields in the program, $f_0 \in F$ is the entry point of the graph that contains no incoming edges, and $E \subseteq F \times F$. The data-flow domain is then $\mathbb{D} = \mathbb{G}_P$, the set of all possible access graphs. Access graphs are more precise than k-limited access paths because they capture recursive field accesses. Despite a large data-flow domain, access graphs are often more efficient than k-limited access paths. Since each access graph concisely represents a set of access paths, the quantity of data-flow facts that the analysis must propagate is often considerably smaller, especially when the access path abstraction uses a high value of k.

Heap graphs (also referred to as shape graphs) are closely related to access graphs but differ in their representation of field accesses. Rather than maintaining a separate graph for each root variable, a heap graph captures the field access relations between heap-allocated objects in the form of a labelled directed graph $G_H = (A, F, \delta)$, where A is the set of heap allocated objects, F

is the set of fields in the program, and $\delta \subseteq A \times F \times A$ is the set of field-labelled edges between heap objects. Similar to access graphs, the data-flow domain for a heap graph-based analysis is $\mathbb{D} = \mathbb{G}_P$. Kanvar and Khedker [26] discuss this class of heap abstractions at depth.

2.4 Pushdown systems

Pushdown systems were originally developed as a tool for model-checking programs with recursive procedure calls [11] and are currently widely-used for both verification and program analysis [30]. Definitions 1-5 in this section are adapted from Bouajjani et al. [11], and Definitions 6 and 7 are adapted from Späth et al. [55].

Definition 1. A pushdown system (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control locations (i.e., states), Γ is a finite stack alphabet, and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of PDS rules. We use the syntax $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ for a rule $((p, \gamma), (p', w)) \in \Delta$, and we refer to $c = \langle p, w \rangle$ as a *configuration* of $\mathcal{P}$, where $p \in P$ and $w \in \Gamma^*$. We denote the set of all configurations by $\mathcal{C}$.

Throughout the remainder of this thesis, we assume that $\forall w.(\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta \implies |w| \leq 2)$. This assumption does not restrict the expressiveness of a PDS, because any PDS that admits rules with $|w| > 2$ can be transformed to an equivalent PDS with the restriction $|w| \leq 2$ [20].

Similar to pushdown automata, pushdown systems describe a set of operations performed on a stack. In contrast to pushdown automata, pushdown systems do not accept any input because they are not intended to solve language-recognition problems. Rather, a pushdown system is generative—when provided with a set of initial configurations C, a PDS generates a transition system that captures the set of configurations that are reachable from C. We define PDS reachability in terms of the predecessor relation *pre* and the successor relation *post*.

Definition 2. $pre : 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ is defined as

$pre(C) = \{\langle p, \gamma w\rangle \in \mathcal{C} \mid \exists \langle p', w'\rangle \in C \;.\; \langle p, \gamma\rangle \hookrightarrow \langle p', w'\rangle \in \Delta\}$

Definition 3. $post : 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ is defined as
$post(C) = \{\langle p, w\rangle \in \mathcal{C} \mid \exists \langle p', \gamma w'\rangle \in C \;.\; \langle p', \gamma\rangle \hookrightarrow \langle p, w\rangle \in \Delta\}$

We can compute all configurations reachable from a set of initial configurations C by applying the transitive closure of the pre and $post$ relations, denoted by pre^* and $post^*$, respectively. The sets $pre^*(C)$ and $post^*(C)$ are possibly infinite, but they are regular and can therefore be represented by finite automata.

Definition 4. Given a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$, a $\mathcal{P}$-automaton is a finite state automaton $\mathcal{A} = (\Gamma, Q, \delta, P, F)$, where $Q \supseteq P$ is the set of states, $\delta \subseteq Q \times \Gamma \times Q$ is the set of transitions, P is the set of initial states, and $F \subseteq Q$ is the set of final states.

Definition 5. The transition relation $\rightarrow \subseteq Q \times \Gamma^* \times Q$ for a $\mathcal{P}$-automaton is the smallest relation satisfying the following:

- if $q \in Q$, then $q \xrightarrow{\epsilon} q$
- if $(q, \gamma, q') \in \delta$, then $q \xrightarrow{\gamma} q'$
- if $q \xrightarrow{\gamma} q'$ and $q' \xrightarrow{w} q''$, then $q \xrightarrow{\gamma w} q''$

Informally, a transition $q \xrightarrow{w} q'$ is included in the transition relation for $\mathcal{A}$ if q' is reachable from q along a path labelled by w. A $\mathcal{P}$-automaton recognizes a set of configurations C if, for every $\langle q, w\rangle \in C$, there is a corresponding $q_f \in F$ such that $q \xrightarrow{w} q_f$. The sets $pre^*(C)$ (resp. $post^*(C)$) are computable by first constructing a $\mathcal{P}$-automaton $\mathcal{A}_0$ that recognizes C, and then repeatedly adding transitions to $\mathcal{A}_0$ such that every time a new transition (q, γ, q') is added to δ, a new configuration $\langle q, \gamma w\rangle \in pre^*(C)$ (resp. $post^*(C)$) is recognized by $\mathcal{A}_0$. Efficient algorithms [20] can construct $pre^*(C)$ in $O(|Q|^2|\Delta|)$ time and $O(|Q||\Delta| + |\delta|)$ space, and $post^*(C)$ in $O(|P||\Delta|(|Q| + |\Delta|) + |P||\delta|)$ for both time and space.

```
1   main() {
2     A a = new A();
3     A b = new A();
4     A c = m(a, b);
5   }
6
7   m(A x, A y) {
8     if (...) {
9       return x;
10    } else {
11      return y;
12    }
13  }
```

(a) A small example program.

Normal Rules
$\langle \texttt{a@2}, * \rangle \hookrightarrow \langle \texttt{a@3}, * \rangle$
$\langle \texttt{x@7}, * \rangle \hookrightarrow \langle \texttt{x@8}, * \rangle$
$\langle \texttt{x@7}, * \rangle \hookrightarrow \langle \texttt{x@10}, * \rangle$
$\langle \texttt{x@8}, * \rangle \hookrightarrow \langle \texttt{x@9}, * \rangle$
$\langle \texttt{x@10}, * \rangle \hookrightarrow \langle \texttt{x@11}, * \rangle$
$\langle \texttt{y@7}, * \rangle \hookrightarrow \langle \texttt{y@8}, * \rangle$
$\langle \texttt{y@7}, * \rangle \hookrightarrow \langle \texttt{y@10}, * \rangle$
$\langle \texttt{y@8}, * \rangle \hookrightarrow \langle \texttt{y@9}, * \rangle$
$\langle \texttt{y@10}, * \rangle \hookrightarrow \langle \texttt{y@11}, * \rangle$

Push Rules
$\langle \texttt{a@3}, * \rangle \hookrightarrow \langle \texttt{x@7}, 4 \cdot * \rangle$
$\langle \texttt{b@3}, * \rangle \hookrightarrow \langle \texttt{y@7}, 4 \cdot * \rangle$

Pop Rules
$\langle \texttt{x@9}, 4 \rangle \hookrightarrow \langle \texttt{a@4}, \epsilon \rangle$
$\langle \texttt{y@9}, 4 \rangle \hookrightarrow \langle \texttt{b@4}, \epsilon \rangle$
$\langle \texttt{x@11}, 4 \rangle \hookrightarrow \langle \texttt{a@4}, \epsilon \rangle$
$\langle \texttt{y@11}, 4 \rangle \hookrightarrow \langle \texttt{b@4}, \epsilon \rangle$
$\langle \texttt{x@9}, 4 \rangle \hookrightarrow \langle \texttt{c@4}, \epsilon \rangle$
$\langle \texttt{y@11}, 4 \rangle \hookrightarrow \langle \texttt{c@4}, \epsilon \rangle$

(b) PDS rules for the program in Figure 2.1a. The symbol $*$ is the wildcard symbol, denoting that the rule accepts any statement $s \in S$.

Figure 2.1

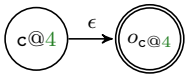
2.4.1 Context-sensitive analysis with pushdown systems

Pushdown systems can encode distributive data-flow problems. As an example, we construct a context-sensitive PDS to track interprocedural data-flow for the example in Figure 2.1a.

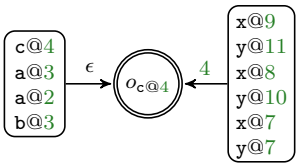
Definition 6. Given a set of program variables V and a set of program statements S, a context-sensitive PDS is $\mathcal{P}_S = (V \times S, S \cup \{\epsilon\}, \Delta_S)$. We use the notation $v@s$ for a control location $(v, s) \in V \times S$. The symbol ϵ denotes an empty stack.

Figure 2.1b contains the PDS rules for the example in Figure 2.1a. For a PDS rule of the form $\langle v@s, \gamma \rangle \hookrightarrow \langle v'@s', w \rangle$, we refer to rules with $|w| = 1$ as *normal rules*, rules with $|w| = 2$ as *push rules*, and rules with $|w| = 0$ as *pop rules*.

Normal rules encode intraprocedural data-flow, while push and pop rules encode procedure calls and returns, respectively. For example, the push rule $\langle \texttt{a@3}, * \rangle \hookrightarrow \langle \texttt{x@7}, 4 \cdot * \rangle$ reflects that the value `a` at Line 3 flows to value `x` at Line 7. The portion of the rule $w = 4 \cdot *$ pushes the return site at Line 4 onto the stack. The pop rule $\langle \texttt{x@9}, 4 \rangle \hookrightarrow \langle \texttt{c@4}, \epsilon \rangle$ handles the return flow from `x`



(a) Initial automaton $\mathcal{A}_0$ that accepts the configuration $\langle \texttt{c@4}, \epsilon \rangle$



(b) Saturated automaton $\mathcal{A}$ that accepts the set of configurations $pre^*(\langle \texttt{c@4}, \epsilon \rangle)$

Figure 2.2: Initial and saturated $\mathcal{P}$-automata for the example in Figure 2.1a.

at Line 9 to `c` at Line 4. The 4 in the start configuration $\langle \texttt{x@9}, 4 \rangle$ removes the return site 4 from the stack, and the ϵ in the end configuration indicates that no other statement is pushed onto the stack.

To determine which values flow to variable `c` at Line 4, we compute $pre^*(C)$ for $C = \{\langle \texttt{c@4}, \epsilon \rangle\}$. Figure 2.2a shows the initial $\mathcal{P}$-automaton that accepts C, and Figure 2.2b shows the saturated $\mathcal{P}$-automaton that accepts $pre^*(C)$. From the saturated $\mathcal{P}$-automaton, we conclude that both $\langle \texttt{a@2}, \epsilon \rangle$ and $\langle \texttt{b@3}, \epsilon \rangle$ flow to $\langle \texttt{c@4}, \epsilon \rangle$.

2.4.2 Synchronized pushdown systems

The example in Section 2.4.1 uses a pushdown system to perform a flow- and context-sensitive analysis. However, this formulation is field-insensitive, and the problem of simultaneously fully-field-sensitive and fully-context-sensitive analysis is undecidable [39]. Synchronized pushdown systems provide a close approximation of a fully-field-sensitive and fully-context-sensitive analysis by solving the field-sensitive and context-sensitive problems separately and taking the intersection of their results. The SPDS approach constructs two PDS instances—a fully-field-sensitive but context-insensitive instance and a fully-context-sensitive but field-insensitive instance. We define the constext-sensitive PDS (from here onwards we refer to the context-sensitive PDS as the "call-PDS") in Section 2.4.1, and we define the field-sensitive PDS ("field-PDS") below.

Definition 7. Given a set of program variables V, a set of program statements S, and a set of program fields F, a field-PDS is $\mathcal{P}_F = (V \times S, F \cup \{\epsilon\}, \Delta_F)$.

```
14      main() {
15        A a = new A();
16        A b = new A();
17        a.f = b;
18        A c = foo(a);
19      }
20
21      foo(A x) {
22        if (...) {
23          return x;
24        }
25        A y = x.f;
26        A z = foo(y);
27        return z;
28      }
```

Figure 2.3: A small recursive program with field accesses.

Because the field-PDS is context-insensitive, it encodes procedure calls and returns as normal rules rather than push and pop rules. Instead, the field-PDS stack abstraction tracks field write (push) operations and field read (pop) operations. Conversely, the call-PDS encodes field read and write statements as normal rules.

Figure 2.3 provides an example program with both interprocedural flow and field accesses. Figure 2.4a and Figure 2.4b show the rules for the call-PDS and field-PDS, respectively. By applying either the pre^* or $post^*$ saturation procedure to both pushdown systems for some initial configuration c_0, we can determine data-flow reachability for c_0.

For this example, suppose we wish to compute the set of value-statement pairs in the program that can point to the allocation site at line 16. We first choose our initial configuration $c_0 = \langle \mathtt{b}@16, \epsilon \rangle$ and construct two initial automata $\mathcal{A}_0^S$ and $\mathcal{A}_0^F$ that recognize c_0. The superscripts S and F refer to the respective stack symbol domains of the call-PDS and field-PDS. Because our analysis starts at an allocation site, we wish compute forward reachability information, so we compute the $post^*(\{c\})$ sets for both pushdown systems.

Figure 2.5a and Figure 2.5b show the saturated automata $\mathcal{A}^S$ and $\mathcal{A}^F$ that

encode the sets of forward-reachable configurations $post^*(\{c\})$ for the call-PDS and field-PDS, respectively. Taking the intersection of the reachable states of $\mathcal{A}^S$ and $\mathcal{A}^F$ yields Table 2.1, which describes the per-statement points-to information for the allocation site at Line 3.

Variable	Statement	Access Path
`b`	16, 17, 18	ϵ
`a`	17, 18	`f`
`c`	18	`f` $\cup \epsilon$
`x`	21, 22, 24	`f`
`y`	25, 26	ϵ
`z`	26	ϵ

Table 2.1: Points-to information for the object allocated at Line 16

BluJ uses synchronized pushdown systems and heap graphs to construct summaries of library data-flow. The summaries retain the majority of the desirable properties of SPDS as an analysis framework: they are fully context- and field-sensitive, even in the presence of recursive method calls and unbounded field accesses. The main difference between BluJ's summaries and the pure SPDS framework is in the dimension of flow-sensitivity. SPDS is fully flow-sensitive, but BluJ uses a flow-insensitive heap graph. Despite this, BluJ is still flow-sensitive in most cases. We discuss BluJ's flow sensitivity further in Section 4.6. The following chapter discusses other approaches to library summarization and how they differ from BluJ.

Normal Rules
$\langle \mathtt{a}@15, * \rangle \hookrightarrow \langle \mathtt{a}@16, * \rangle$
$\langle \mathtt{a}@16, * \rangle \hookrightarrow \langle \mathtt{a}@17, * \rangle$
$\langle \mathtt{b}@16, * \rangle \hookrightarrow \langle \mathtt{b}@17, * \rangle$
$\langle \mathtt{b}@16, * \rangle \hookrightarrow \langle \mathtt{a}@17, * \rangle$
$\langle \mathtt{b}@17, * \rangle \hookrightarrow \langle \mathtt{b}@18, * \rangle$
$\langle \mathtt{x}@21, * \rangle \hookrightarrow \langle \mathtt{x}@22, * \rangle$
$\langle \mathtt{x}@21, * \rangle \hookrightarrow \langle \mathtt{x}@24, * \rangle$
$\langle \mathtt{x}@24, * \rangle \hookrightarrow \langle \mathtt{x}@25, * \rangle$
$\langle \mathtt{x}@24, * \rangle \hookrightarrow \langle \mathtt{y}@25, * \rangle$
$\langle \mathtt{x}@25, * \rangle \hookrightarrow \langle \mathtt{x}@26, * \rangle$

Pop Rules
$\langle \mathtt{x}@23, 18 \rangle \hookrightarrow \langle \mathtt{c}@18, \epsilon \rangle$
$\langle \mathtt{x}@23, 18 \rangle \hookrightarrow \langle \mathtt{a}@18, \epsilon \rangle$
$\langle \mathtt{x}@23, 26 \rangle \hookrightarrow \langle \mathtt{z}@26, \epsilon \rangle$
$\langle \mathtt{x}@23, 26 \rangle \hookrightarrow \langle \mathtt{y}@26, \epsilon \rangle$
$\langle \mathtt{z}@27, 18 \rangle \hookrightarrow \langle \mathtt{c}@18, \epsilon \rangle$
$\langle \mathtt{x}@27, 18 \rangle \hookrightarrow \langle \mathtt{a}@18, \epsilon \rangle$
$\langle \mathtt{z}@27, 26 \rangle \hookrightarrow \langle \mathtt{z}@26, \epsilon \rangle$
$\langle \mathtt{x}@27, 26 \rangle \hookrightarrow \langle \mathtt{y}@26, \epsilon \rangle$

Push Rules
$\langle \mathtt{a}@17, * \rangle \hookrightarrow \langle \mathtt{x}@21, 18 \cdot * \rangle$
$\langle \mathtt{y}@25, * \rangle \hookrightarrow \langle \mathtt{x}@21, 26 \cdot * \rangle$

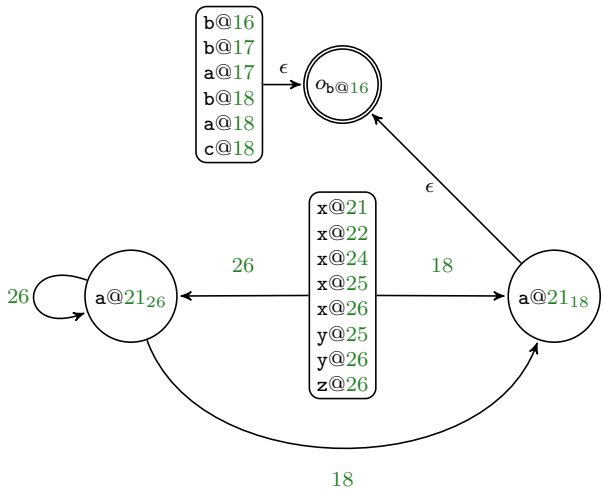
(a) Call-PDS rules for the program in Figure 2.3.

Normal Rules
$\langle \mathtt{a}@15, * \rangle \hookrightarrow \langle \mathtt{a}@16, * \rangle$
$\langle \mathtt{a}@16, * \rangle \hookrightarrow \langle \mathtt{a}@17, * \rangle$
$\langle \mathtt{b}@16, * \rangle \hookrightarrow \langle \mathtt{b}@17, * \rangle$
$\langle \mathtt{b}@17, * \rangle \hookrightarrow \langle \mathtt{b}@18, * \rangle$
$\langle \mathtt{x}@21, * \rangle \hookrightarrow \langle \mathtt{x}@22, * \rangle$
$\langle \mathtt{x}@21, * \rangle \hookrightarrow \langle \mathtt{x}@24, * \rangle$
$\langle \mathtt{x}@24, * \rangle \hookrightarrow \langle \mathtt{x}@25, * \rangle$
$\langle \mathtt{x}@25, * \rangle \hookrightarrow \langle \mathtt{x}@26, * \rangle$
$\langle \mathtt{a}@17, * \rangle \hookrightarrow \langle \mathtt{x}@21, * \rangle$
$\langle \mathtt{x}@23, * \rangle \hookrightarrow \langle \mathtt{a}@18, * \rangle$
$\langle \mathtt{x}@23, * \rangle \hookrightarrow \langle \mathtt{c}@18, * \rangle$
$\langle \mathtt{x}@27, * \rangle \hookrightarrow \langle \mathtt{a}@18, * \rangle$
$\langle \mathtt{z}@27, * \rangle \hookrightarrow \langle \mathtt{c}@18, * \rangle$
$\langle \mathtt{y}@25, * \rangle \hookrightarrow \langle \mathtt{x}@21, * \rangle$
$\langle \mathtt{x}@23, * \rangle \hookrightarrow \langle \mathtt{y}@26, * \rangle$
$\langle \mathtt{x}@23, * \rangle \hookrightarrow \langle \mathtt{z}@26, * \rangle$
$\langle \mathtt{x}@27, * \rangle \hookrightarrow \langle \mathtt{y}@26, * \rangle$
$\langle \mathtt{z}@27, * \rangle \hookrightarrow \langle \mathtt{z}@26, * \rangle$

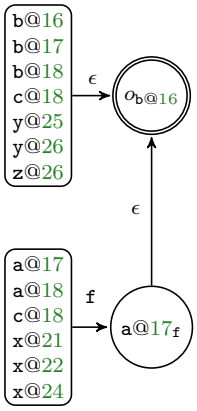
Pop Rules
$\langle \mathtt{x}@24, \mathtt{f} \rangle \hookrightarrow \langle \mathtt{y}@25, \epsilon \rangle$

Push Rules
$\langle \mathtt{b}@16, * \rangle \hookrightarrow \langle \mathtt{a}@17, \mathtt{f} \cdot * \rangle$

(b) Field-PDS rules for the program in Figure 2.3.



(a) Saturated call-automaton for the example in Figure 2.3



(b) Saturated field-automaton for the example in Figure 2.3

Figure 2.5: Saturated $\mathcal{P}$-automata initialized with the configuration $\langle \mathtt{b}@16, \epsilon \rangle$

Chapter 3

Related Work

In this chapter, we discuss related approaches to library summarization for static analysis, as well as the more general case of partial program analysis.

3.1 Analysis of Partial Programs

Rountev et al. [44] formalize the notion of a flow-sensitive data-flow analysis for partial programs as an extension of the classical monotone data-flow framework (Section 2.1). They extend the framework with a program fragment F, which is a collection of procedures contained within some program P, and an abstraction of available whole-program information I. This approach works under the assumption that I may contain some easily-computable information to bootstrap the analysis, such as flow-insensitive whole-program pointer information and a context-insensitive call graph. Because BLUJ computes persistent summaries that are applicable across multiple programs, it cannot rely on any whole-program information.

Chatterjee et al. [15] present a bottom-up approach to modular points-to analysis that can be used to analyze partial programs. Their analysis initially constructs a call graph and traverses the methods of the call graph in reverse topological order, computing a pointer summary transfer function for each method. The pointer summary transfer function captures the effects that a method's internal pointer assignments may have on its unknown initial inputs. The analysis then traverses the call graph in topological (top-down) order and propagates each method's concrete pointer relations to its callees. When Chat-

terjee, et al.'s approach analyzes partial programs, it requires k-limiting on the access paths of unknown initial values at the boundaries of the partial program. BLUJ avoids k-limiting at library boundaries by using a pushdown system as its field abstraction. BLUJ's internal implementation also differs from that of Chatterjee, et al.—they use a summary-based compositional approach, while BLUJ computes demand-driven points-to information.

Utture and Palsberg [63] improve static analysis performance by providing the client analysis with a partial library specifically tailored to the application code rather than the whole library. The downside of this approach is that it is inherently unsound—it ignores references to code outside the scope of the partial library. The target clients for this approach are IDE integrations aimed at developer productivity. In these scenarios, soundness has less importance than precision and scalability. BLUJ instead targets client analyses that benefit from soundness guarantees (up to some measurable level of soundness [36]), such as security analyses.

3.2 Call Graph Summaries

Ali and Lhoták [1] formalize the distinction between Java *application classes* and *library classes* as a set of constraints on library behaviour that follow from the assumption that library classes can be compiled in the absence of application classes. In subsequent work [2], Ali and Lhoták leverage the constraints from [1] to construct persistent summaries of Java library behaviour in a tool called AVERROES. The summaries that AVERROES constructs use classes and methods referenced in the application code to reason about objects that are possibly available to the library and are therefore tailored to individual applications. AVERROES can construct its summaries quickly and with low memory footprint because it does not actually inspect the library source code. Instead, it obtains information about class, method, and field references from the constant pool. AVERROES' generated summary library is flow-insensitive and is most useful for constructing application call graphs, whereas BLUJ aims to

compute more precise and mostly-flow-sensitive[1] data-flow information.

3.3 Data-flow summaries

Rountev et al. [45] construct library data-flow summaries for the interprocedural distributive environment (IDE) [47] framework. The core insight of this approach is that the IDE framework can be extended with awareness of callbacks to application methods from the library, allowing the summaries that IDE computes to be composed once the application code is available. Curiously, Rountev, et al. do not provide details on how the summaries can be persisted for later re-use, and they instead discard the summaries at the end of each analysis run.

Arzt and Bodden [6] build upon the work of Rountev, et al. [45] to create flow-sensitive taint analysis summaries for Java and the Android framework. They implement their approach in a tool called StubDroid, which takes a Java library class and a list of taint sources/sinks as input and produces a taint summary for that class. At library boundaries, StubDroid reports tainted fields in the form of k-limited access paths. While StubDroid is specifically tailored to taint analysis, BluJ is more general and captures additional data flows that may nor be of specific interest to a taint analysis. For example a points-to analysis that integrates BluJ's summaries can answer aliasing queries for arbitrary application variables.

Schubert et al. [49] apply a compositional data-flow summary approach to C/C++ that simultaneously computes a call graph, pointer information, and distributive data-flow information for the interprocedural finite distributive subset (IFDS) [40] and IDE frameworks. The data-flow portion of the approach is similar to that of Rountev, et al. [45] in that it is bottom-up and composes function information at each calling context, and the points-to information that it computes is flow-insensitive.

Tang et al. [61] present a language-theoretic approach to library data-flow

[1] see Section 4.6 for a discussion of the situations in which BluJ's computed summary is not flow-sensitive

summarization via tree-adjoining language (TAL) reachability. TALs are a family of mildly context-sensitive languages that allow the definition of strings that contain *gaps*. The ultimate decision of whether a tree-adjoining language L with alphabet Σ contains a particular string $s \in \Sigma^*$ can be expressed as a conditional expression over the gaps present in s. The nature of TALs makes them a convenient formalism for representing context-sensitive data-flow in library code that contains many callbacks. However, for a program with a control-flow graph $G_c = (S, E, s_0)$, the time complexity for solving TAL-reachability is $O(|S|^6)$, making this approach infeasibly slow for many real-world programs. By contrast, BLUJ's time complexity is identical to that of $post^*$, i.e., $O(|P||\Delta|(|Q| + |\Delta|) + |P||\delta|)$, which is equivalent to a worst case of $O(|S|^5)$ (Section 2.4).

Chapter 4

Heap Graph Augmented Synchronized Summaries

Chapter 2 and Chapter 3 provide background and an overview of recent research results on data-flow analysis and persistent library summaries. This chapter introduces a novel method of generating a precise library data-flow summary, which we refer to as a heap graph augmented synchronized summary (HGASS). Similar to prior work [6], an HGASS is field- and context-sensitive, and it captures data flows affected by aliasing relations within the library. An HGASS is also flow-sensitive for most summarized data-flows (Section 4.6 discusses situations where an HGASS is not flow-sensitive). An HGASS additionally captures data flows involving arbitrarily complex field accesses by combining *Synchronized Pushdown Systems* [55] with *heap graphs* [26]. We additionally implement a data-flow solver called BLUJ to generate HGASSs.

This chapter first provides an overview of the challenge involved in generating precise library summaries and the motivation for *Heap Graph Augmented Synchronized Summaries*. This chapter then details a naive approach to library summarization that uses synchronized pushdown systems, highlighting the data-flows that cannot be captured by synchronized pushdown systems alone. Finally, this chapter introduces heap graphs and describes how an HGASS uses them to account for the flows that synchronized pushdown systems miss.

```
29  recursive(A x){
30    while (...) {
31      x = x.f;
32      ...
33    }
34    return x;
35  }
```

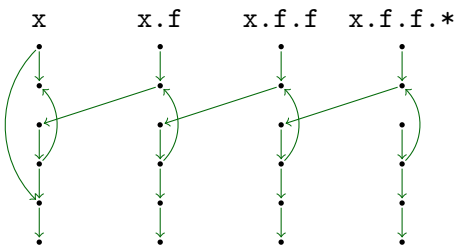


Figure 4.1: Data-flow graph for library method `recursive()` using the access path field abstraction with limit $k = 2$.

4.1 Overview

When analyzing library code in the absence of application code, the analysis possesses little information about the objects passed into the library from an application. A library analysis is able to reason about the declared type of the object at the library entry point, but not about the object's runtime type or what might be stored in the object's fields.

One technique to deal with the lack of information about fields of objects passed to the library is to abstract the objects' fields with k-limited access paths at the library entry point [6]. A k-limited data-flow analysis tracks data flows for not only each object in a given program, but for each of the object's fields, recursively, up to a depth of k chained field accesses. As k increases, there is a combinatorial explosion in the number of flows that must be tracked through the library, and the analysis designer must determine a value for k that allows the analysis to terminate in a reasonable amount of time.

For chains of field accesses longer than k (including recursive field accesses), the analysis does not distinguish between fields, resulting in a drastic drop in precision. Figure 4.1 shows a method `recursive()` that contains recursive field accesses and its exploded supergraph (ESG) [40]. Each vertex of the ESG represents a value-statement pair (v, s) in the method, and the set of edges is a graphical representation of the flow function f assuming a data-flow domain $\mathbb{D} = 2^V$ (i.e., the powerset of program values). Within the `while` loop, field `x.f` is read and assigned to `x` an arbitrary number of times. Depending

Table 4.1: Definitions for entry and exit points of a library method m. Square brackets denote that a portion of the statement is optional.

Variable	Statement	Condition	Type
		Library Entry Points	
$this$	$firstStmt(m)$	$isPublic(m) \wedge \neg isStatic(m)$	instance method receiver
v	$firstStmt(m)$	$isPublic(m) \wedge v \in params(m)$	method parameter
v	$v = new\ A()$		allocation site
v	$v = A.f$	$f \in staticFields(A)$	static field read
r	$r = v.p(...)$	$isOverridable(p)$	callback return value
v	$[r =]\ v.p(...)$	$isOverridable(p)$	callback receiver return
x	$[r =]\ v.p(x, ...)$	$isOverridable(p)$	callback argument return
		Library Exit Points	
$r.*$	$return\ r$	$isPublic(m)$	return value for m
$this.*$	$return\ [r]$	$isPublic(m) \wedge \neg isStatic(m)$	receiver return
$v.*$	$return\ [r]$	$isPublic(m) \wedge v \in params(m)$	parameter return
$v.*$	$A.f = v$	$f \in staticFields(A)$	static field write
$v.*$	$[r =]\ v.p(...)$	$isOverridable(p)$	callback receiver
$x.*$	$[r =]\ v.p(x, ...)$	$isOverridable(p)$	callback parameter

on the application calling context, the access path abstraction may or may not be able to precisely reason about all objects accessible to the library through repeated field accesses on `x.f`.

Synchronized pushdown systems provide an alternative to k-limited access paths by abstracting fields with *pushdown systems* that precisely encode all field reads and writes. An SPDS instance is a pair of pushdown systems $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})$ where $\mathcal{P}_{\mathbb{F}}$ encodes a field-sensitive (and context-insensitive) analysis and $\mathcal{P}_{\mathbb{S}}$ encodes a context-sensitive (and field-insensitive) analysis (Section **??**). The field-PDS avoids the precision limitations of k-limited access paths by abstracting all reachable field accesses on an object as a set of regular configurations that can be efficiently represented by a finite state automaton.

In a typical whole-program analysis, the analysis instantiates an SPDS instance for every allocation site in the program, and each SPDS instance executes its saturation procedures until all reachability information is computed for every allocation site. In the absence of application code, BluJ instead instantiates an SPDS instance at every **library entry point** and tracks the flow of objects passed into the library to every reachable **library exit point**. A variable-statement pair is considered a library entry point (or library exit

Table 4.2: Helper functions for determining library entry and exit points

Function	Input	Output
isPublic	Method m	`true` if m is declared `public` `false` otherwise
isProtected	Method m	`true` if m is declared `protected` `false` otherwise
isStatic	Method m	`true` if m is declared `static` `false` otherwise
params	Method m	P, the set of parameters declared by m
staticFields	Class A	F, the set of static fields declared by A
isOverridable	Method p	`true` if $(isPublic(p) \vee isProtected(p)) \wedge \neg isStatic(p)$ $\wedge$ the class that declares p is not declared `final` `false` otherwise

point) if it conforms to the conditions in one or more of the rows of Table 4.1. The helper functions used in Table 4.1 are defined in Table 4.2. Note that BluJ's definition for *isOverridable*() assumes that applications do not declare classes in the same packages as library classes and consequently cannot override library methods declared as `package-private`. For now, we restrict our discussion of HGASS entry points to the first two entries in Table 4.1. Allocation sites and static fields are discussed in Section 4.2, and callbacks are discussed in Section 4.5.

To summarize method `foo()` from the example in Figure 4.2, BluJ instantiates an SPDS solver at the library entry point for each respective parameter: $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{u@36}$ and $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{v@36}$. The saturation procedure for $\mathcal{P}_{\mathbb{S}u@36}$ is straightforward: variable `u` is not used in any call statements and is only involved in the field write statements at Lines 39 and 40. Since the field write statements overwrite *fields* of `u` rather than `u` itself, the field write statements do not affect the saturated call-automaton of $\mathcal{A}_{\mathbb{S}u@36}$. The saturation procedure for $\mathcal{P}_{\mathbb{F}u@36}$ is similar. Once again, `u` is not used in any call statements and the field writes on Lines 39 and 40 do not affect the field-automaton $\mathcal{A}_{\mathbb{F}u@36}$. $\mathcal{A}_{\mathbb{S}u@36}$ and $\mathcal{A}_{\mathbb{F}u@36}$ are shown in Figure 4.3a and Figure 4.3b, respectively. Since the library exit point `u`@40 is reachable in both automata, we conclude that `u`@36 flows to `u`@40.

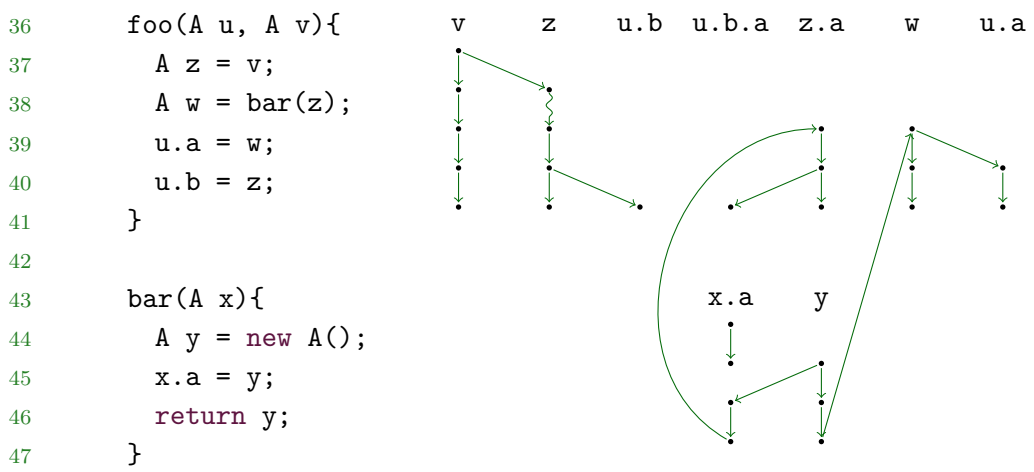
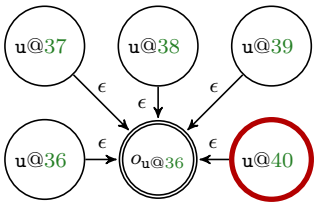


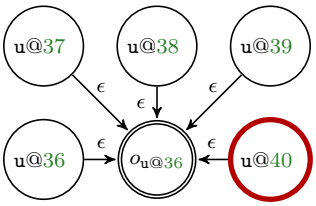
Figure 4.2: Data-flow graph for library methods `foo` and `bar`, ignoring aliasing relations.

The saturation procedure for $\mathcal{A}_{\mathbb{S}v@36}$ yields a call automaton that tracks the flow of `z` through method `bar` and the assignment of `z` to `u.*` at Line 40. $\mathcal{A}_{\mathbb{F}v@36}$ also tracks the flow of `z` through method `bar()` and processes the field write statement that assigns `z` to `u.b`. $\mathcal{A}_{\mathbb{S}v@36}$ and $\mathcal{A}_{\mathbb{F}v@36}$ are shown in Figure 4.3c and Figure 4.3d, respectively. There are two library exit points reachable in both automata: `v`@40 and `u`@40. Since `v`@40 is reachable in both automata along ϵ-transitions, we conclude that `v`@36 flows to `v`@40. Since `u`@40 is only reachable in $\mathcal{A}_{\mathbb{F}v@36}$ along a path labelled with `b`, we conclude that `v`@36 flows to `u.b`@40.

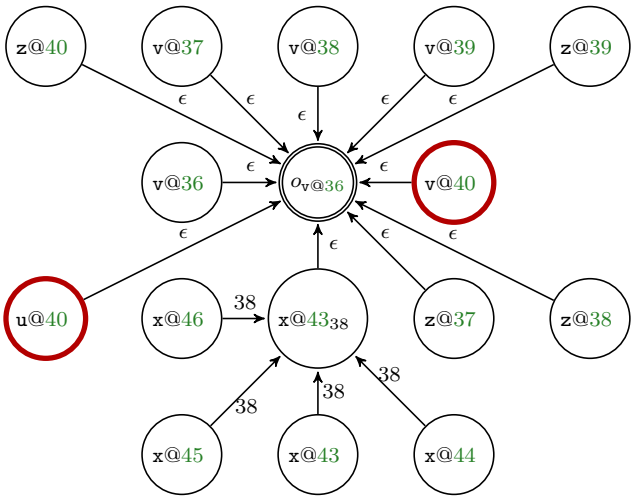
Putting all of the aforementioned data-flows together yields the summary in Figure 4.4. However, this summary only partially captures the data-flows that escape the library. Notably, the flows from the allocation site at Line 44 to `u.a` and `u.b.a` are not realized. Neither is the flow from the allocation site to `v.a` that arises from the aliasing relationship between `v` and `z`. The following two sections discuss these missing flows and how an HGASS instance can capture them.



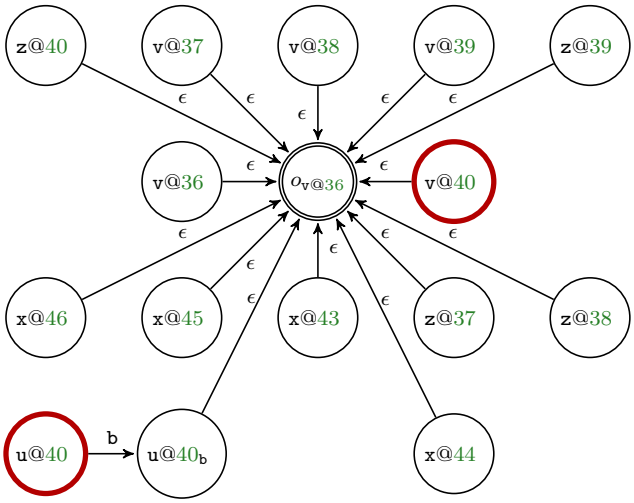
(a) Saturated call $\mathcal{P}$-automaton for parameter `u`



(b) Saturated field $\mathcal{P}$-automaton for parameter `u`



(c) Saturated call $\mathcal{P}$-automaton for parameter `v`



(d) Saturated field $\mathcal{P}$-automaton for parameter `v`

Figure 4.3: Saturated automata for SPDS instances $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{u@36}$ and $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{v@36}$. Library exit points are highlighted in dark red.

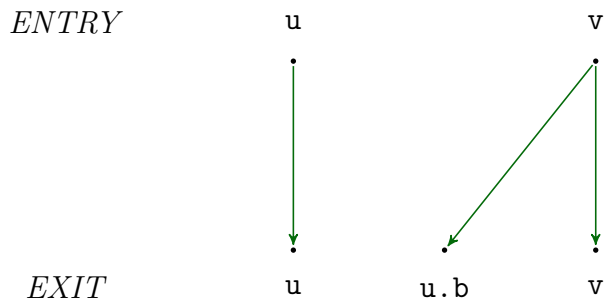


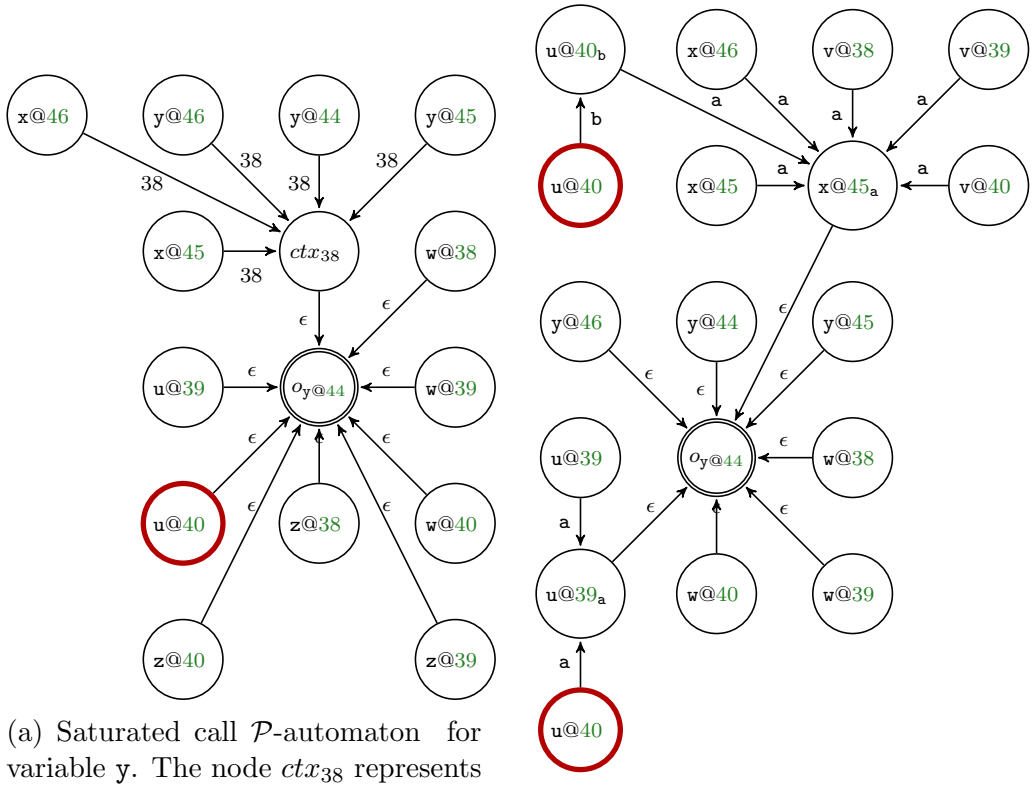
Figure 4.4: Alias-unaware data-flow summary for paramters `u` and `v` of method `foo`.

4.2 Allocation sites and static fields

BluJ handles allocation site and static field read entry points as they are discovered during the saturation procedures of other SPDS instances. In most cases, the process is straightforward: an existing SPDS instance reaches an allocation site or static field read statement, and BluJ creates a new SPDS instance originating at that statement.

Returning to the example in Figure 4.2, the SPDS solver $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{v@36}$ reaches Line 44 where a new object is allocated. From here, BluJ creates a new SPDS instance $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{y@44}$ that tracks the flow of `y` through the library. To maintain context sensitivity with respect to the data-flow path along which the allocation site was discovered, $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{v@36}$ shares its context with $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{y@44}$. Each time BluJ discovers an allocation site under a different calling context, it creates a new SPDS instance for the allocation site, taking its context from the discovering solver. Figure 4.5 shows the saturated automata $\mathcal{A}_{\mathbb{S}y@44}$ and $\mathcal{A}_{\mathbb{F}y@44}$. Since $\mathcal{A}_{\mathbb{S}y@44}$ receives context information from $\mathcal{A}_{\mathbb{S}v@36}$, the solver identifies the return flow from `y`@46 to `w`@38. Both automata reach the exit state `u`@40, and the state is reachable in $\mathcal{A}_{\mathbb{F}y@44}$ along two paths: a path labelled with `a` and a path labelled with `b` $*$ `a`. Therefore, we conclude that `y`@44 flows to `u.a` and `u.b.a`. The summary is still not complete, however, because it is missing a flow that results from aliasing. Section 4.3 addresses how BluJ handles aliasing.

BluJ handles static field reads in a similar manner to allocation sites.



(a) Saturated call $\mathcal{P}$-automaton for variable `y`. The node ctx_{38} represents the calling context shared with $\mathcal{P}_{\mathbb{S}y@44}$ by $\mathcal{P}_{\mathbb{S}v@36}$.

(b) Saturated field $\mathcal{P}$-automaton for variable `y`

Figure 4.5: Saturated automata for SPDS instance $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{y@44}$. Library exit points are highlighted in dark red.

```
48    B readWriteStatic(B t) {
49      B q = Util.f;
50      Util.f = t;
51      return q;
52    }
```

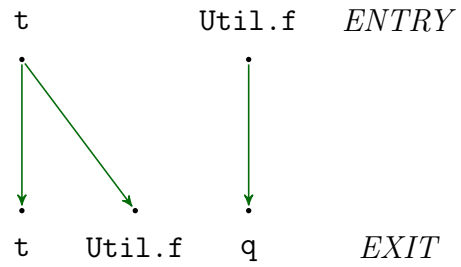


Figure 4.6: Library method `readWriteStatic()` and its data-flow summary.

When BluJ encounters a static field read, it instantiates a new SPDS instance to track the flow of the value assigned at the field read. BluJ also treats static field writes as library exit points, because the value written to a static field possibly becomes accessible to application code. For the example in Figure 4.6, BluJ tracks the flow from parameter `t` to the static field write at Line 50 as well as from the static field read at Line 49 to the return value `q`.

In general, there is no guarantee that all allocation sites and static field reads along a valid execution path will occur along BluJ's tracked data-flow paths. It is possible that a data-flow query encounters a static method call site `r = m()`. In this situation, the solver will not enter method `m()`, but flows could possibly originate in method `m()` that may escape the library through the return value `r`. Because we assume that all library classes can be compiled in the absence of application classes (the separate compilation assumption [1]), this situation may only arise at parameterless static method calls. It follows from the separate compilation assumption that any object available to the library is either created in the library or passed into the library by the application at a library entry point. In either case, BluJ tracks the object. Any method call in the library that is non-static must therefore have a receiver o that BluJ tracks into the body of the method, and for any static method call in the library that has some parameter p, BluJ tracks p into the body of the method. Because paramaterless static method calls do not propagate the data-flow of other objects into the method body, BluJ must treat them as a special case.

It is important to note that the separate compilation assumption is a simplifying assumption that may not always hold in practice. For example, some Java libraries such as JUnit [24] make heavy use of reflection to obtain information about the application that is not available to the library at compile time. For some languages, libraries frequently violate the separate compilation assumption, such as C++ (via template metaprogramming) and Common Lisp (via macros).

To avoid missing flows that originate in static method calls, BluJ abstracts the declaring classes of static members as if they are heap objects and tracks

```
53    public class L {
54      public A libraryMethod() {
55        A x = L.getNewA();
56        return x;
57      }
58
59      private static A getNewA() {
60        A y = new A();
61        return y;
62      }
63    }
```

Figure 4.7: Library class with API method `libraryMethod()` and non-API static method `getNewA()` that creates a new instance of `A`. When `libraryMethod()` is called, the new instance of `A` is returned from the library.

their flow through the library as well. For the example in Figure 4.7, BluJ creates an SPDS solver instance for `L`@55 as if `L` is the receiver of the method call. While tracking the flow of `L`, the SPDS solver $(\mathcal{P}_{\mathbb{F}}, \mathcal{P}_{\mathbb{S}})_{\texttt{L@55}}$ discovers the allocation site at Line 60, and BluJ begins tracking the flow of `y`@60 as well. If BluJ did not track the flow of `L`, no SPDS instance would enter the body of method `getNewA()`, and the value returned from the library at Line 56 would be missed.

4.3 Aliasing

Points-to analysis is non-distributive [40]. Since SPDS flow functions **are** distributive, a single SPDS instance is incapable of accurately capturing points-to information for flows that involve aliasing [54]. To account for this, BluJ handles non-distributivity in the same manner as Boomerang [56]: SPDS instances share information with one another at *points of aliasing*, which occur at field writes and call statements along a solver's data-flow path. Definitions 8 and 9 in this section are adapted from Späth [54].

Definition 8. Given a field $\mathcal{P}$-automaton $\mathcal{A}_{\mathbb{F}} = (Q, \Gamma, \delta, P, acc)$, a *field write point of aliasing* is a state $c = (y@x.f = y) \in Q$.

Any object tracked by an SPDS instance that was accessible through y

(or any of its fields) at statement $x.f = y$ is then also accessible through $x.f$ (or any of its fields) after the field write. Additionally, the object is now accessible through field f of any **alias** of x. To account for aliases of x, BluJ uses a system of listeners to receive updates from any field automata $\mathcal{A}'_{\mathbb{F}} = (Q', \Gamma', \delta', P', acc')$ such that $(x@x.f = y) \in Q'$ and copies relevant aliasing information from $\mathcal{A}'_{\mathbb{F}}$ to $\mathcal{A}_{\mathbb{F}}$. BluJ then re-starts the saturation procedure for $\mathcal{A}_{\mathbb{F}}$ with the new information present in the automaton. However, this procedure is only sufficient for capturing aliasing that occurs within the scope of the current method. If x is aliased to an object outside the method scope, BluJ must also process a *call site point of aliasing*.

Definition 9. Given a field $\mathcal{P}$-automaton $\mathcal{A}_{\mathbb{F}} = (Q, \Gamma, \delta, P, acc)$, a *call site point of aliasing* is a pair of states (c_f, c_s) where $c_f = (y@z.f = y) \in Q$ is a field write point of aliasing, $c_s = (x_i@x_0.m(..., x_n)) \in Q$ for some $0 \leq i \leq n$, and z is the ith parameter of method m.

At the call return site $x_0.m(..., x_n)$, z and x_i point to the same object. BluJ locates the field automata for additional aliases of x_i and copies relevant information to $\mathcal{A}_{\mathbb{F}}$. As with field write points of aliasing, BluJ then re-starts the saturation procedure for $\mathcal{A}_{\mathbb{F}}$ with the new information. For a more in-depth discussion of points-to analysis using SPDS, we refer the reader to Chapter 5 of Johannes Späth's doctoral thesis [54].

Figure 4.8 reproduces the example from Figure 4.2 while highlighting the alias relationships that were omitted in Section 4.1. The field write point of aliasing at Line 73 triggers a call site point of aliasing when method `bar()` returns at Line 66. BluJ searches for aliases of `z` at Line 66 and copies relevant information from any aliases that it finds. Since `z` and `v` alias at Line 66, BluJ copies the state v@66 into $\mathcal{A}_{\mathbb{F}_{y@72}}$ and connects it to the accepting state of $\mathcal{A}_{\mathbb{F}_{y@72}}$ via an edge labelled with `a`. This new edge reflects that at Line 66, `v.a` points to $o_{\mathtt{y@72}}$. After re-saturating $\mathcal{A}_{\mathbb{F}_{y@72}}$, the flow from y@72 now reaches the library exit point v@68 along access path `a`.

4.4 Tracking complex field flows

```
64      foo(A u, A v){
65        A z = v;
66        A w = bar(z);
67        u.a = w;
68        u.b = z;
69      }
70
71      bar(A x){
72        A y = new A();
73        x.a = y;
74        return y;
75      }
```

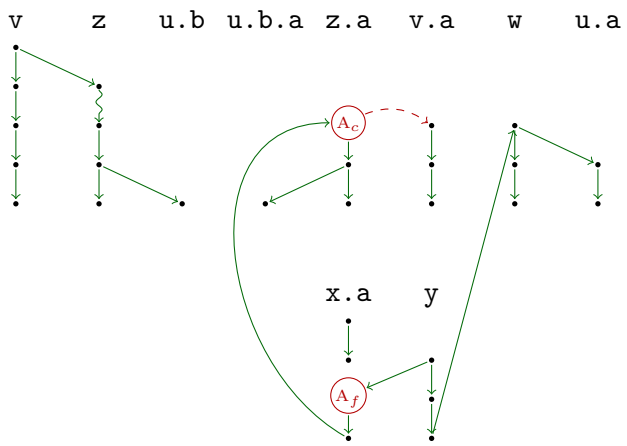


Figure 4.8: Data-flow graph for library methods `foo` and `bar`, with aliasing information.

```
76      baz(A x, A y){
77        B p = x.b;
78        A q = y.a;
79        q.b = p;
80      }
```

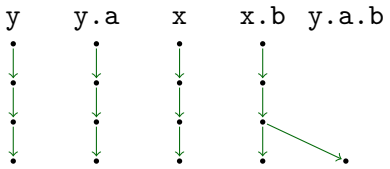


Figure 4.9: Data-flow graph for library method `baz()`.

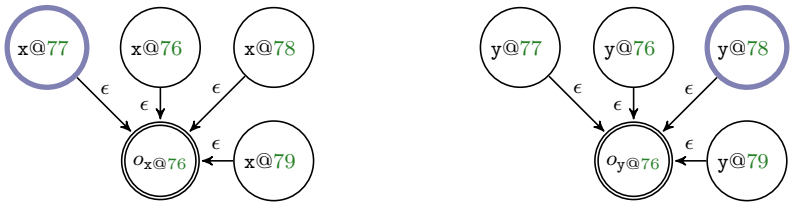


Figure 4.10: Saturated field $\mathcal{P}$-automata for parameters `x` (left) and `y` (right) in Figure 4.9. Unbalanced field reads are highlighted in dark blue.

An SPDS instance is sufficient for capturing field-sensitive flows for objects that are passed directly into the library via method parameters, static field reads, or new object allocations, but it misses flows originating at objects that are passed into the library as fields of other objects. Consider the example in Figure 4.9. Line 77 reads field `b` from parameter `x` and assigns it to `p`. Similarly, Line 78, reads field `a` from parameter `y` and assigns it to `q`. Then, Line 79 assigns variable `p` to `q.b`. Since `q` aliases with `y.a`, this results in a flow from `x.b` to `y.a.b`.

Figure 4.10 shows the saturated field automata for `x` and `y`. The SPDS instances for the parameters miss the flow from `x.b` to `y.a.b` because the flow originates at fields of the parameters rather than at the parameters themselves. This issue is solved with a combination of **delegate SPDS solvers** and **heap graphs**.

4.4.1 Delegate solvers

To ensure that the analysis is aware of values passed into the library as fields, we introduce a delegate solver at each *unbalanced field read*.

Definition 10. Given a field $\mathcal{P}$-automaton $\mathcal{A}_{\mathbb{F}} = (Q, \Gamma, \delta, P, acc)$, an *unbalanced field read* is a state $c = (v@x = v.f) \in Q$ for some arbitrary v, x such that $\forall q \in Q, (c, f, q) \notin \delta$.

Informally, an *unbalanced field read* is any field read where, in the absence of application code, the library has no way of determining what is stored in the field.

A delegate solver is an additional SPDS instance that tracks the assigned variable of a field read. A delegate solver is denoted by $\mathcal{D}_{v@s}$, where v is the

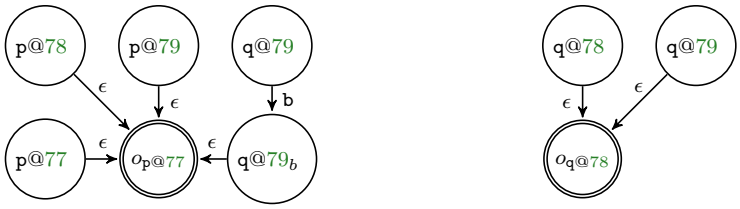


Figure 4.11: Saturated field $\mathcal{P}$-automata for delegate solvers $\mathcal{D}_{\texttt{p@77}}$ (left) and $\mathcal{D}_{\texttt{q@78}}$ (right) in Figure 4.9.

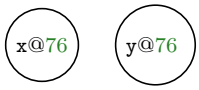
value tracked by $\mathcal{D}$ and s is the statement at which the $\mathcal{D}$ is generated. BluJ additionally maintains the set of delegate solvers $\mathbb{D} \subset \mathbb{A}$ and a mapping $\mathbb{D} \xrightarrow{f} \mathbb{A}$ between delegate solvers and their parent solvers. Delegate solver creation is restricted to unbalanced field reads to avoid unnecessarily repeating analysis steps. In the case of a *balanced* field read of the form $c@d = c.f$, a solver that encodes the reachable state $c@d = c.f$ is able to reason about the flow from $c.f$ to d because the corresponding write into field f occurs along a data-flow path that is known to the solver. For the example in Figure 4.9, all field reads are unbalanced and BluJ generates two delegate solvers: $\mathcal{D}_{\texttt{p@77}}$, which tracks the value `p` loaded from `x.b`, and $\mathcal{D}_{\texttt{q@78}}$, which tracks the value `q` loaded from `y.a`. Figure 4.11 shows the saturated field automaton for each of the delegate solvers.

There are now four total solvers for method `baz()`, and the delegate solver $\mathcal{D}_{\texttt{p@77}}$ informs the analysis that `p` and `q.b` point to the same object when the method returns at Line 79. However, the analysis is still unable to detect the flow from `x.b` to `y.a.b`. To capture this flow, it is necessary to keep track of the field read and write relationships between each of the solver instances.

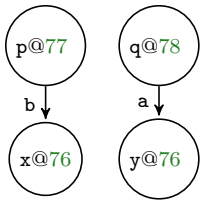
4.4.2 Heap graphs

Heap graphs [14] are abstract models of heap memory that are commonly used in shape analysis to reason about the structure and relations between heap allocated objects [41]. In this thesis, we define a heap graph as follows (adapted from Kanvar and Khedker [26]).

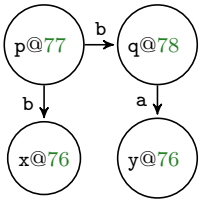
Definition 11. A heap graph is a 5-tuple $G = (V, S, F, Q, E)$ where V is a set of program values, S is a set of program statements, F is a set of field names,



(a) Initial heap graph containing only the respective accepting states of $\mathcal{A}^x{}_{\mathbb{F}}$ and $\mathcal{A}^y{}_{\mathbb{F}}$



(b) Heap graph after the creation of delegate solvers $\mathcal{D}_{\texttt{p@77}}$ and $\mathcal{D}_{\texttt{q@78}}$



(c) Heap graph after the addition of the field write alias edge $\texttt{p@77} \xrightarrow{\texttt{b}} \texttt{q@78}$

Figure 4.12: Heap graph construction for the summary of library method `baz()` in Figure 4.9.

$Q \subseteq V \times S$ is the set of *locations* or *nodes* of G, and $E \subseteq Q \times F \times Q$ is the set of *labelled edges* of G.

The following operations are defined on G:

- $AddNode(q)$: Given $q = (v, s)$, if $v \in V$ and $s \in S$, then $Q \leftarrow Q \cup \{q\}$
- $AddEdge(q, f, q_0)$: if $q, q_0 \in Q$ and $f \in F$, then $E \leftarrow E \cup \{(q, f, q_0)\}$

BluJ uses a heap graph to keep track of field read and write relationships between solvers, and the heap graph is constructed simultaneously with the iterative solving approach described in Section 4.3. As described in subsection 4.4.1, delegate solvers are created at unbalanced field reads. When BluJ creates a new delegate solver $\mathcal{D}_{v@s}$, BluJ updates the heap graph with a new node (v, s). Additionally, BluJ adds an edge from (v, s) to the accepting state (v_0, s_0) of the solver containing the unbalanced field read. Returning to the example in Figure 4.9, Figure 4.12b shows the state of the heap graph after the creation of delegate solvers $\mathcal{D}_{\texttt{p@77}}$ and $\mathcal{D}_{\texttt{q@78}}$.

When a delegate solver tracks a value v that is written to a field f of variable x, the heap graph must also be updated to take into account any values which

can now access the object pointed to by v through $x.f$. To accomplish this, BluJ finds any other solvers in $\mathbb{A}_\mathbb{F}$ for which the variable x is reachable at the field write statement. BluJ then updates the heap graph with new f-labelled edges from the delegate solver's accepting state to the respective accepting states of the other solvers, reflecting the aliasing relationship between v and $x.f$ at the field write statement. For the delegate solvers in Figure 4.11, $\mathcal{D}_{\texttt{p@77}}$ contains a state `q@79` that is reachable by a write to field `b`. Since $\mathcal{D}_{\texttt{q@78}}$ also contains the state `q@79`, BluJ adds a new heap graph edge from `p@77` to `q@78`, labelled with `b`, as shown in Figure 4.12c.

We give the precise semantics of heap graph construction in the form of inference rules below.

$$\frac{\begin{array}{c}(1)\ \mathcal{A}_\mathbb{F} = (Q, \Gamma, \delta, P, acc) \\ (2)\ \mathcal{A}_\mathbb{F} \in \mathbb{A}_\mathbb{F} \qquad (3)\ q = (v, x = v.f) \in Q \\ (4)\ IsUnbalancedRead(\mathcal{A}_\mathbb{F}, q) \qquad (5)\ q' = (x, x = v.f)\end{array}}{\begin{array}{c}G.AddNode(q') \qquad G.AddEdge(q', f, acc)\end{array}}\ \textsc{NewDelegate}$$

$$\frac{\begin{array}{c}(1)\ \mathcal{A}_\mathbb{F} = (Q, \Gamma, \delta, P, acc) \\ (2)\ \mathcal{A}_\mathbb{F} \in \mathbb{D}_\mathbb{F} \qquad (3)\ \mathcal{A'}_\mathbb{F} = (Q', \Gamma', \delta', P', acc') \\ (4)\ \mathcal{A'}_\mathbb{F} \in \mathbb{A}_\mathbb{F} \qquad (5)\ \mathcal{A'}_\mathbb{F} \neq \mathcal{A}_\mathbb{F} \\ (6)\ q = (v, x.f = v) \in Q \qquad (7)\ q' = (x, x.f = v) \in Q'\end{array}}{G.AddEdge(acc, f, acc')}\ \textsc{AliasEdge}$$

The inference rule NewDelegate defines the logic for creating a new heap graph node and edge at the point that a new delegate solver is introduced. Premises (1) and (2) require that there exists some solver $\mathcal{A}_\mathbb{F}$ in the method summary's set of field solvers $\mathbb{A}_\mathbb{F}$. Premise (3) requires that the set of states Q of $\mathcal{A}_\mathbb{F}$ contains some state q where the associated statement is a field read and the associated value is the base of the field read. In premise (4), the helper function $IsUnbalancedRead(\mathcal{A}_\mathbb{F}, q)$ returns *true* if q is an unbalanced field read of the automaton $\mathcal{A}_\mathbb{F}$. Premise (5) introduces a state q' that has the same associated statement as q, but the associated value is the value assigned to at the field read statement. The conclusions add q' as a new node in G and

add a new edge from q' to the accepting state of $\mathcal{A}_{\mathbb{F}}$, labelled with field f.

The rule AliasEdge defines the addition of new graph edges at field write points of aliasing. Premises (1) and (2) introduce a delegate solver $\mathcal{A}_{\mathbb{F}}$. Premises (3) and (4) introduce a second solver, $\mathcal{A'}_{\mathbb{F}}$, which can be any solver that is a member of the method summary's set of field solvers. Premise (5) ensures that the two solvers are not equivalent. $\mathcal{A}_{\mathbb{F}}$ already encodes the field relation between x and v, so the heap graph does not need to be updated in this case. Premise (6) states that $\mathcal{A}_{\mathbb{F}}$ must have some state q where the associate statement is a field write and the associated value is the value being written. Premise (7) states that $\mathcal{A'}_{\mathbb{F}}$ must have a state q' where the associated statement is the same field write statement as in premise (6) and the associated value is the base of the field being written to. The conclusion adds a new edge to G that starts at the accepting state of $\mathcal{A}_{\mathbb{F}}$, ends at the accepting state of $\mathcal{A'}_{\mathbb{F}}$, and is labelled with field f.

4.4.3 Generating summaries with complex field flows

In previous sections, we have listed access paths at *library exit points* by reading the labelled edges of the saturated field automaton. When a non-delegate solver has incoming edges from another solver in the heap graph, we read the labelled edges of the heap graph and append the resulting *reversed* field expression $(f)^*$ to the field expression $(g)^*$ generated by the solver's field automaton. Conversely, we obtain the access path at a *library entry point* in the following manner: for each delegate solver $\mathcal{D}_{v'@s'}$ in the heap graph with a non-delegate ancestor $\mathcal{A}_{v@s}$ we take the field expression $(h)^*$ generated by the relation $\mathcal{D}_{v'@s'}(\xrightarrow{f})^*\mathcal{A}_{v@s}$. If $\mathcal{A}_{v@s}$ contains a *library exit point* $v_e@s_e$, we conclude that there is a data flow from $v.(h)^*@s$ to $v_e.(g)^*.(f)^*@s_e$. In practice, $(h)^*$, $(f)^*$ and $(g)^*$ can be arbitrarily complex regular expressions. To generate these regular expressions, we use an implementation[1] of Robert Tarjan's path expression algorithm [62].

Returning to the example in Figure 4.9, once all saturation procedures are finished, BluJ looks up the delegate solver $\mathcal{D}_{\texttt{p@77}}$ in the mapping $\mathbb{D} \xrightarrow{f} \mathbb{A}$

[1] `https://github.com/johspaeth/PathExpression`

and finds that $\mathcal{D}_{\texttt{p@77}}$ maps to its parent solver $\mathcal{A}_{\texttt{x@76}}$ along $(h)^* = \texttt{b}$. Then, BluJ consults the heap graph and finds that $\mathcal{D}_{\texttt{p@77}}$ reaches $\mathcal{A}_{\texttt{x@76}}$ along a path labelled with $(f)^* = \texttt{b}$ and $\mathcal{A}_{\texttt{y@76}}$ along a path labelled with $(f)^* = \texttt{b.a}$. Finally, BluJ looks for library exit points in $\mathcal{A}_{\texttt{x@76}}$ and $\mathcal{A}_{\texttt{y@76}}$. In, $\mathcal{A}_{\texttt{x@76}}$, BluJ finds the exit point `x`@79, which is accessible along the path $(g)^* = \epsilon$ and adds the summary data-flow `x.b`@76 $\rightarrow$ `x.b`@79 to the HGASS. In, $\mathcal{A}_{\texttt{y@76}}$, BluJ finds the exit point `y`@79, which is accessible along the path $(g)^* = \epsilon$ and adds the summary data-flow `x.b`@76 $\rightarrow$ `y.a.b`@79 to the HGASS.

4.5 Callbacks

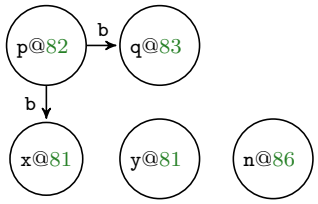
When a library method is overridable by application code, calls by the library to the overridable method may resolve to application methods at runtime. Consequently, BluJ treats calls into overridable methods as library exit points and returns from overridable methods as library entry points. The helper function *isOverridable* in Table 4.2 defines the criteria for determining whether or not a library method is overridable.

In the running example in Figure 4.8, if the method `bar()` is overridable, BluJ considers the flow into and out of the call site at Line 66. This means that BluJ treats `z`@65 as a library exit point. BluJ additionally treats `z`@66 and `w`@66 as library entry points and instantiates SPDS instances at both locations. When generating HGASSs at the end of the analysis, flows involving entry/exit points at overridable callsites are marked as *conditional* to inform the client analysis consuming the HGASS that the flows depend on whether or not the application overrides `bar()`. Once the client analysis has access to application code, it can determine whether or not to use the conditional flows. Section 5.2.2 discusses in detail how our example client analysis handles summaries with callbacks.

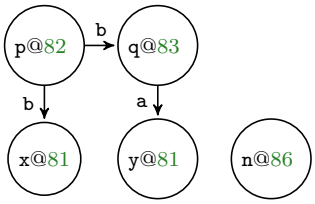
4.6 Discussion

One of the drawbacks of heap graphs as a primary mechanism for heap abstraction is their inefficiency [26]. Traditionally, heap graphs model a whole

```
81      baz(A x, A y){
82        B p = x.b;
83        A q = y.a;
84        q.b = p;
85        C.f = y;
86        A n = new A();
87        y.a = n;
88      }
```



(a) Unsound heap graph implemented with strong updates



(b) Heap graph implemented without strong updates

Figure 4.13: Example library method where BLUJ cannot perform strong updates on the heap graph.

program's heap state at every program statement (and under every calling context if the analysis is context-sensitive), meaning that the analysis must maintain at least one graph node for every variable that is in scope at a given statement. The upper bound on the total number of heap graph nodes is $2^{|Var|}$, where $|Var|$ is the number of program variables [48]. For large programs, the memory required to maintain a precise and complete heap graph can be prohibitive. However, BLUJ only maintains heap graph nodes at locations that generate SPDS queries (i.e., library entry points and unbalanced field read statements) rather than at every statement-variable pair.

An additional consequence of the reduced scope of nodes stored in BLUJ's heap graph is that BLUJ cannot soundly perform strong updates on the heap graph. The example in Figure 4.13 extends the code snippet in Figure 4.9 to illustrate the issue. The field `y.a` is overwritten with a new value at Line 87, so `x.b` no longer escapes the library through `y.a.b`. Because BLUJ's heap graph does not store heap information at every statement, it cannot reason about the relative execution order of the static field write statement at Line 85 and the write to `y.a` at Line 87. Consequently, BLUJ cannot determine whether

`y.a` and `q` still point to the same object when static field `C.f` is written to. BluJ performs a safe over-approximation by keeping the edge from `q`@83 to `y`@81 in the graph, leading to a spurious data-flow from `x.b` to `y.a.b` in the final HGASS.

Chapter 5

Adapting Summaries for Client Analyses

While Chapter 4 detailed the process of HGASS generation, this chapter describes how a client analysis can consume an HGASS. We integrate our summaries into Boomerang, a state-of-the-art SPDS-based pointer analysis framework, via the BluJ summary adapter (BSA), which parses and applies summaries at calls to library methods. Figure 5.1 provides an overview of BluJ's architecture.

In the following section, we discuss the structure of HGASSs and the format in which BluJ stores them on disk. We then discuss how our Boomerang-based summary adapter consumes HGASSs and conclude by describing sources of imprecision that HGASSs may introduce to a pointer analysis.

5.1 Storing Persistent Summaries

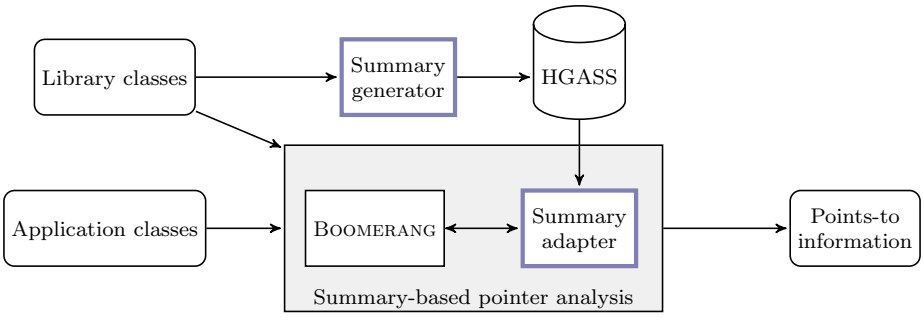


Figure 5.1: BluJ's architecture and typical workflow. Components outlined in blue are part of BluJ.

```
...
{
  "declaringClass": "Example",
  "subSignature": "void foo(A, A)"
},
...
  {
    "start": {
      "location": {
        "value": {
          "valueType": "ParameterRef",
          "n": 0,
          "paramType": {
            "sootClass": "A"
          }
        },
        "stmt": {
          "stmtType": "JNopStmt"
        }
      },
      "expression": "Wildcard"
    },
    "end": {
      "location": {
        "value": {
          "valueType": "JimpleLocal",
          "name": "r4",
          "type": {
            "sootClass": "A"
          }
        },
        "stmt": {
          "stmtType": "JReturnVoidStmt"
        }
      },
      "expression": "Wildcard"
    },
    "callbackGaps": []
  }
...
```

Figure 5.2: Excerpt from an HGASS for the example in Figure 4.2

Storing summaries on disk allows a client analysis to amortize the cost of summary generation over each run of the client analysis. Assuming that each execution of the client analysis saves computation time by using the summary, the up-front cost of summary generation becomes negligible with respect to time saved in the client analysis. While we tailor our summaries to SPDS-based solvers such as BOOMERANG, an HGASS is generic and consumable by other data-flow frameworks as long as they provide an appropriate summary adapter. To facilitate persistence and genericity, we serialize HGASSs to easily-parsable JavaScript object notation (JSON). BLUJ generates an HGASS for each summarized library class. The top-level JSON element in each file is an array of API methods declared by the class. For each method, the summary contains an array of data-flows, and each data-flow provides a start point, an end point, and a set of possible callbacks that may affect the flow. The start and end points of each data-flow additionally define the field regular expression through which the value of the data-flow is reachable. Figure 5.2 shows an excerpt from the JSON representation of an HGASS detailing a single data flow from the first parameter of `foo()` to its corresponding local value at a return statement.

An HGASS is not fully generic across all static analysis frameworks. It relies on abstractions provided by the Soot static analysis framework, specifically the program statement and value abstractions of Soot's Jimple intermediate representation (IR). While it would be possible to generate summaries that are analysis-framework agnostic, doing so would add an additional computational cost to summary serialization and deserialization. Determining whether an additional layer of abstraction may provide any practical benefit is beyond the scope of this thesis.

5.2 Using Summaries

BLUJ's BSA hooks into BOOMERANG by registering a listener on each BOOMERANG solver instance. A BOOMERANG solver notifies the BSA listener when its *post** algorithm encounters a *library entry call rule*.

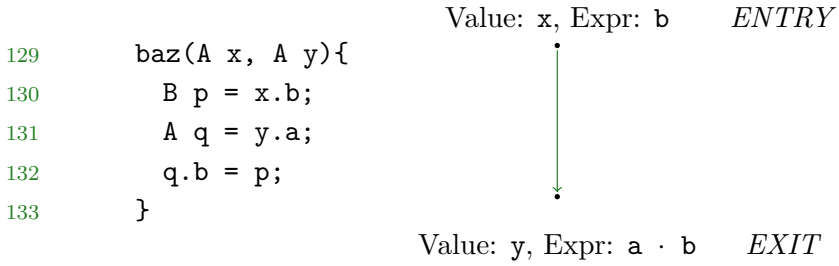


Figure 5.3: Summary of a single data-flow for library method `baz()`, from `x`@129 to `y`@132

Definition 12. Given $v_a, v_l \in V$, the set of program variables, $s_c \in S_{app}$, the set of statements contained within application classes, and $s_{entry} \in S_{lib}$, the set of statements contained within library classes, a *library entry call rule* is a Call-PDS rule of the form $\langle v_a @ s_c, * \rangle \hookrightarrow \langle v_l @ s_{entry}, s_c \cdot * \rangle$

A library entry call rule corresponds to the interprocedural data-flow generated at an application call site that targets a library method. When BOOMERANG encounters a library entry call rule $\langle v_a @ s_c, * \rangle \hookrightarrow \langle v_l @ s_{entry}, s_c \cdot * \rangle$, the BSA checks whether there is a data-flow summary present that begins at the library entry point $v_l @ s_{entry}$. If no summary is present, the BSA returns control to BOOMERANG, and BOOMERANG continues its analysis of the library method. If a summary is present, the BSA caches the summary and begins the process of creating PDS rules from the field regular expressions at the library entry and exit points.

5.2.1 Generating PDS rules from regular expressions

The primary purpose of the BSA is to transform the field access path regular expressions at the entry and exit point of each summary into the PDS rule format recognized by BOOMERANG. We use the example in Figure 5.3 to explain the PDS rule generation process. At the library entry point, the expression is the single field `b`. To access field `b` from the base variable `x`, the program would need to perform a field read, so the BSA generates a field pop PDS rule $\langle \texttt{x} @ 129, \texttt{b} \rangle \hookrightarrow \langle \texttt{x} @ s_{fresh}, \epsilon \rangle$. Here, s_{fresh} denotes a unique placeholder statement that does not exist in the concrete program. The BSA then processes the expression `a · b` at the library exit point. The BSA generates one push

$$\textsc{Epsilon}\ \frac{}{(\epsilon, v@s, \emptyset)}$$

$$\textsc{EntryField}\ \frac{}{(f, v@s, \{\langle v@s, f\rangle \hookrightarrow \langle v'@s_{fresh}, \epsilon\rangle\}) \Downarrow (\epsilon, v'@s_{fresh}, \emptyset)}$$

$$\textsc{ExitField}\ \frac{}{(f, v@s, \{\langle v@s, *\rangle \hookrightarrow \langle v'@s_{fresh}, f \cdot *\rangle\}) \Downarrow (\epsilon, v'@s_{fresh}, \emptyset)}$$

$$\textsc{Concat}\ \frac{(e_1, v@s, \Delta_{\mathbb{F}}) \Downarrow (e_1', v'@s', \Delta_{\mathbb{F}}') \qquad (e_2, v'@s', \Delta_{\mathbb{F}}) \Downarrow (e_2', v''@s'', \Delta_{\mathbb{F}}'')}{(e_1 e_2, v@s, \Delta_{\mathbb{F}}) \Downarrow (e_1' e_2', v''@s'', \Delta_{\mathbb{F}}' \cup \Delta_{\mathbb{F}}'')}$$

$$\textsc{Union}\ \frac{(e_1, v@s, \Delta_{\mathbb{F}}) \Downarrow (e_1', v'@s', \Delta_{\mathbb{F}}') \qquad (e_2, v@s, \Delta_{\mathbb{F}}) \Downarrow (e_2', v''@s'', \Delta_{\mathbb{F}}'')}{\begin{array}{c}(e_1 \cup e_2, v@s, \Delta_{\mathbb{F}} \cup \{\langle v'@s', *\rangle \hookrightarrow \langle v_f@s_{fresh}, *\rangle, \langle v''@s'', *\rangle \hookrightarrow \langle v_f@s_{fresh}, *\rangle\}) \Downarrow \\ (e_1' \cup e_2', v_f@s_{fresh}, \Delta_{\mathbb{F}}' \cup \Delta_{\mathbb{F}}'')\end{array}}$$

$$\textsc{Star}\ \frac{(e, v@s, \Delta_{\mathbb{F}}) \Downarrow (e', v'@s', \Delta_{\mathbb{F}}')}{(e*, v@s, \Delta_{\mathbb{F}} \cup \{\langle v'@s', *\rangle \hookrightarrow \langle v@s, *\rangle\}) \Downarrow (e', v'@s', \Delta_{\mathbb{F}}')}$$

Figure 5.4: Big-step operational semantics for generating field-PDS rules from HGASS regular expressions

rule for each field symbol at the exit, yielding $\langle \mathtt{x}@s_{fresh}, *\rangle \hookrightarrow \langle \mathtt{y}@s'_{fresh}, a \cdot *\rangle$ and $\langle \mathtt{y}@s'_{fresh}, *\rangle \hookrightarrow \langle \mathtt{y}@s''_{fresh}, b \cdot *\rangle$.

Figure 5.4 provides a more precise notion of PDS rule generation in the form of big-step operational semantics. For each regular expression that the BSA must process, we define a *meta-configuration* that captures the relation between the regular expression and its corresponding set of PDS rules.

Definition 13. Given a set of program fields F, a set of program control locations $P \subseteq V \times S$, and the set $\Delta_{\mathbb{F}} = (P \times F) \times (P \times F)$ of all PDS rules over P and F, the set of *meta-configurations* is $C = E \times P \times 2^{\Delta_{\mathbb{F}}}$ where E is the set of regular expressions over the alphabet of field symbols F.

We additionally define the big-step operator $\Downarrow$ as the relation $\Downarrow \subseteq C \times C$. Our discussion of the example in Figure 5.3 gave an informal description of the rules EntryField, ExitField, and Concat. We use the example in Figure 5.5 to illustrate the rules Union and Star. As with the previous example, we generate a pop rule for each terminal symbol in the expression. The terminal symbols are `f` and `g`, so the BSA generates the rules

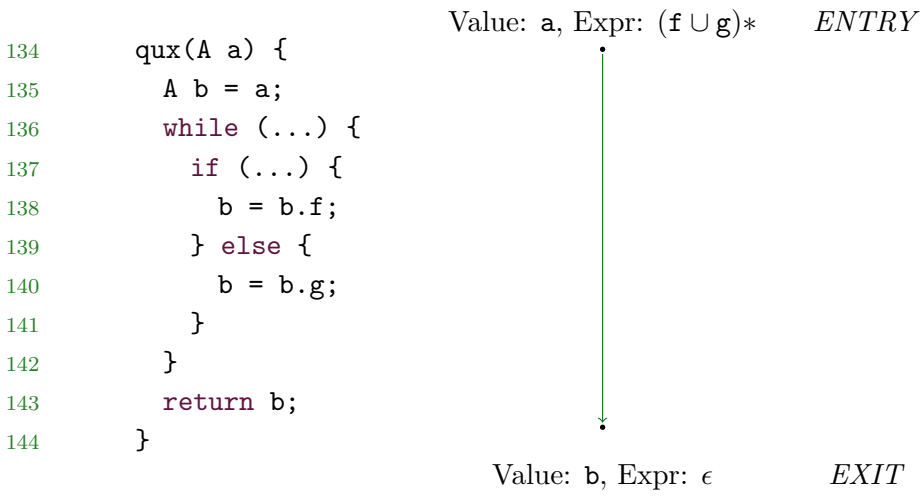


Figure 5.5: Summary of a single data-flow for library method `qux()`, from a@134 to b@143.

$\langle \mathtt{a}@134, \mathtt{f} \rangle \hookrightarrow \langle \mathtt{a}@s'_{fresh}, \epsilon \rangle$ and $\langle \mathtt{a}@134, \mathtt{g} \rangle \hookrightarrow \langle \mathtt{a}@s''_{fresh}, \epsilon \rangle$. The two terminals are combined via the $\cup$ operator, so the BSA additionally generates the normal rules $\langle a@s'_{fresh}, * \rangle \hookrightarrow \langle a@s_{fresh}, * \rangle$ and $\langle a@s''_{fresh}, * \rangle \hookrightarrow \langle a@s_{fresh}, * \rangle$. The two normal rules have the same target configuration, reflecting the merger of data-flow facts upon exiting a branching control structure. Finally, the BSA processes the Kleene star applied to $\mathtt{f} \cup \mathtt{g}$, generating one additional normal rule $\langle a@s_{fresh}, * \rangle \hookrightarrow \langle a@134, * \rangle$. This normal rule reflects a control-flow back edge. Together, we refer to the set of five previously described rules as $\Delta_{\mathbb{F}}$. Using the operational semantics defined in Figure 5.4, we prove the correctness of the set of rules $\Delta_{\mathbb{F}}$ for the regular expression $(\mathtt{f} \cup \mathtt{g})*$ via derivation tree.

$$\dfrac{\dfrac{\dfrac{}{(f, \mathtt{a}@134, \alpha) \Downarrow (\epsilon, \mathtt{a}@s'_f, \emptyset)}\,\textsc{EntryField} \quad \dfrac{}{(g, \mathtt{a}@134, \beta) \Downarrow (\epsilon, \mathtt{a}@s''_f, \emptyset)}\,\textsc{EntryField}}{(\mathtt{f} \cup \mathtt{g}, \mathtt{a}@134, \{\alpha, \beta, \gamma, \delta\}) \Downarrow (\epsilon, \mathtt{a}@s_f, \emptyset)}\,\textsc{Union}}{((\mathtt{f} \cup \mathtt{g})*, \mathtt{a}@134, \{\alpha, \beta, \gamma, \delta, \zeta\}) \Downarrow (\epsilon, \mathtt{a}@s_f, \emptyset)}\,\textsc{Star}$$

After the BSA finishes processing the regular expressions, it provides Boomerang with a *library exit call rule* to process, passing control back to the analysis of the application code.

Definition 14. Given $v_l, v_a \in V$, $s_c, s_{succ} \in S_{app}$, and $s_{exit} \in S_{lib}$, a *library exit*

Symbol	Expanded Rule
α	$\langle a@134, f\rangle \hookrightarrow \langle a@s'_f, \epsilon\rangle$
β	$\langle a@134, g\rangle \hookrightarrow \langle a@s''_f, \epsilon\rangle$
γ	$\langle a@s'_f, *\rangle \hookrightarrow \langle a@s_f, *\rangle$
δ	$\langle a@s''_f, *\rangle \hookrightarrow \langle a@s_f, *\rangle$
ζ	$\langle a@s_f, *\rangle \hookrightarrow \langle a@134, *\rangle$

call rule is a Call-PDS rule of the form $\langle v_l@s_{exit}, s_c\rangle \hookrightarrow \langle v_a@s_{succ}, \epsilon\rangle$, where s_{succ} is the control flow successor of the call site s_c.

The library exit call rule mirrors the library entry call rule, ensuring properly matched calls and returns when applying the data-flow summary.

5.2.2 Callbacks

When the BSA encounters a callback gap while processing a summary, it consults BOOMERANG's call graph for the program and determines the (possibly empty) set of *realizable callbacks* for the callback gap.

Definition 15. Given a program P with a set of statements S, a set of methods M and a call graph $G \subseteq S \times M$, a *realizable callback* is an edge $(s_l, m_a) \in G$ such that s_l is a statement in the library code of P and m_a is a method in the application code of P

If the set of realizable callbacks is empty, the BSA continues processing the data-flow summary. If the set of realizable callbacks is non-empty, the BSA generates a call-PDS push rule $\langle v_l@s_l, *\rangle \hookrightarrow \langle v_a@s_a, s_l \cdot *\rangle$ for each realizable callback and registers a callback return listener with BOOMERANG. The BSA then provides the rules to BOOMERANG, and BOOMERANG analyzes the target methods. When BOOMERANG's analysis reaches the end of a target method, BOOMERANG notifies the callback return listener, which produces a call-PDS pop rule $\langle v'_a@s'_a, s_l\rangle \hookrightarrow \langle v'_l@s'_l, \epsilon\rangle$ for the callback return site $v'_l@s'_l$ and continues applying the library data-flow summary.

Figure 5.6 provides an example program where the data-flow summary includes a library callback. At Line 148, the library calls the abstract method `callback()`. The data-flow summary reflects this by registering a callback gap

```
145     // Library Code
146     public abstract class LibraryClass {
147       public A foo(A x) {
148         A y = callback(x);
149         return y;
150       }
151       public abstract A callback(A z);
152     }
153
154     // Application Code
155     public class ApplicationClass
156       extends LibraryClass {
157
158       public static void main() {
159         A a = new A();
160         A b = foo(a);
161       }
162
163       @Override
164       public A callback(A z) {
165         A w = new A();
166         w.f = z;
167         return w;
168       }
169     }
```

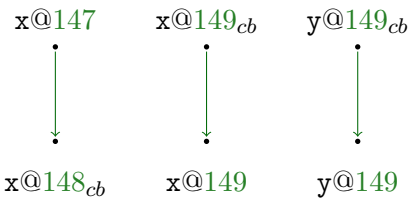


Figure 5.6: Sample program that includes a callback method `callback()` (left), and the data-flow summary for library method `foo()` (right).

for the data-flow of parameter `x` at Line 148. When the BSA processes the data-flow summary for parameter `x` during the client analysis, it encounters the callback gap and consults the program's call graph to find realizable callbacks in the application. In this case, there is only one possible callback target, the overridden method at Line 164. From here, BOOMERANG analyzes the body of the concrete `callback()` method defined on Line 164 and reaches the configurations `z`@167 and `w`@167 (reachable through field `f`). The BSA then resumes applying the summary at the configurations `x`@149 and `y`@149, respectively.

5.3 Additional Imprecision

For each object that an application passes into a library method, BluJ has access to type information for that object only as declared at the library entry point. Consequently, a whole program analysis tracking the flow of the same object from its allocation site may be able to set more precise bounds on the type of the object. The code listing in Figure 5.7 provides an example where a whole-program points-to analysis provides more precise results than a points-to analysis that uses HGASSs. The application code makes a call to `libraryMethod()` on Line 204, and `libraryMethod` (Line 194) declares the type of its first parameter as `LeftOrRight`. In the absence of application code, a sound summary must conclude that the method call `getObject()` on Line 195 may resolve to any of `Left.getObject()` (Line 181), `Right.getObject()` (Line 189), or an unknown override in an application class.

Figure 5.8 shows the data-flow graph for a whole-program analysis of the example in Figure 5.7 with BLUJ's summary flow edges overlaid on the graph. A whole-program analysis has access to the allocation site at Line 201 and can determine that the object passed into the library at Line 204 is of type `Left`. Since the analysis knows the concrete type of `o` at Line 195, the only possible call target is `Left.getObject()`. The whole-program analysis then precisely determines that the points-to set of `y` at Line 204 is $\{o_{\texttt{new@202}}\}$. When the analysis uses BLUJ's summaries instead of analyzing the library, it computes

```
170   // Library code
171   public class LeftOrRight {
172     LeftOrRight left;
173     LeftOrRight right;
174
175     public abstract LeftOrRight getObject();
176   }
177
178   public class Left extends LeftOrRight {
179
180     @Override
181     public LeftOrRight getObject() {
182       return left;
183     }
184   }
185
186   public class Right extends LeftOrRight {
187
188     @Override
189     public LeftOrRight getObject() {
190       return right;
191     }
192   }
193
194   public LeftOrRight libraryMethod(LeftOrRight o){
195     LeftOrRight r = o.getObject();
196     return r;
197   }
198
199   // Application code
200   public void main() {
201     Left x = new Left();
202     x.left = new Left();
203     x.right = new Right();
204     LeftOrRight y = libraryMethod(x);
205   }
```

Figure 5.7: A small program for which BLUJ's library summary is less precise than a whole-program analysis.

```
public void main() {
  Left x = new Left();
  x.left = new Left();
  x.right = new Right();
  LeftOrRight y = libraryMethod(x);
}
```

```
public LeftOrRight libraryMethod(LeftOrRight o){
  LeftOrRight r = o.getObject();
  return r;
}
```

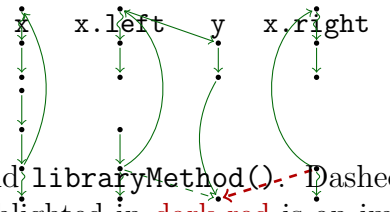


Figure 5.8: Flow graph for methods `main()` and `libraryMethod()`. Dashed edges are BLUJ summary edges. The edge highlighted in dark red is an imprecise flow edge.

the points-to set of `y` at Line 204 as $\{o_{\texttt{new@202}}, o_{\texttt{new@203}}\}$

In some cases, this additional imprecision may have a profound effect on client analysis performance. For example, the commonly-used library methods `StringBuilder.append(Object)` and `HashSet.contains(Object)` have no type bounds on their respective parameters and make use of commonly-overridden `java.lang.Object` methods. The number of spurious data flows resulting from the respective calls to `Object.toString()` and `Object.hashCode()` is so large that summaries of these methods significantly slow down the client analysis. In cases like these, where an object `o` has no type bounds and is the receiver of a method call, BLUJ does not produce a data-flow summary for `o`. BLUJ instead defers the tracking of object `o` to the client analysis.

This chapter gave a theoretical view of how BLUJ integrates its summaries into an existing analysis. In the following chapter, we assess the feasibility of generating HGASSs and their utility as a tool for improving the performance of an existing data-flow analysis.

Chapter 6

Evalutation

The usefulness of an HGASS depends on the balance it can provide between increased performance and decreased precision to a client analysis. Additionally, a useful HGASS must be sound with respect to its client analysis, i.e., it must not miss data-flows that its client analysis detects, and it must be computable in a reasonable amount of time and with a reasonable amount of resources (for example, a summary that takes years to compute is not likely to be useful). To evaluate the performance, precision, and soundness of HGASSs, we formulate the following research questions:

- **RQ1** How much memory does BluJ need to generate a method data-flow summary?
- **RQ2** How many library classes can BluJ summarize in the span of 12 hours?
- **RQ3** How does the integration of HGASSs into an existing data-flow analysis affect the running time, memory usage, soundnes, and precision of that analysis?

To address **RQ3**, we integrated HGASSs into the Boomerang [56] points-to analysis framework and compared analysis results for Boomerang and Boomerang+HGASS. We chose to evaluate our work for **RQ3** with a whole-program points-to analysis because points-to information is a prerequisite for a large class of other analyses. We chose Boomerang as the framework for

integrating HGASSs because it uses SPDSs internally and is therefore able to take advantage of the additional field-sensitivity that BluJ provides over other data-flow summaries.

6.1 Experimental Setup

We carried out all experiments on a server with two AMD EPYC 7351 16-Core Processors. For all experiments we set the maximum Java virtual machine (JVM) heap size to 256GB and the maximum JVM stack size to 1GB.

We used the DaCapo [9] benchmark suite (Version 9.12-MR1-Bach) as our evaluation dataset. Specifically, we use the subset of DaCapo benchmarks available from the Doop [12] benchmark repository [53].[1] Each benchmark consists of two java archive (JAR) files: an application JAR file, which contains all application classes for the benchmark in addition to the classes used by the DaCapo test harness, and a library JAR file, which includes all non-JDK library dependencies for the benchmark. The DaCapo test harness makes heavy use of the Java reflection API, which is notoriously difficult for static analyses to reason about [32]. To account for this, we supplement each analysis run with reflection traces that we obtained by running TamiFlex [10] on each benchmark.

The first step of our evaluation methodology was mining DaCapo to identify library classes as candidates for summarization. We determined candidate classes for summarization by determining which library classes had the most *direct usages* by application classes.

Definition 16. Given a program and its static call graph G, a *direct usage* is a call graph edge $(m, n) \in G$ such that method $m()$ is a member of an application class and method $n()$ is a member of a library class. If $n()$ is a member of library class C, we say that (m, n) is a *direct usage* of C.

To determine direct usages, we used a precise static call graph algorithm

[1] This subset excludes the benchmarks `fop`, `tomcat`, and `daytrader`. We additionally exclude `jython`—we were unable to obtain reflection traces for `jython` due to an issue with TamiFlex.

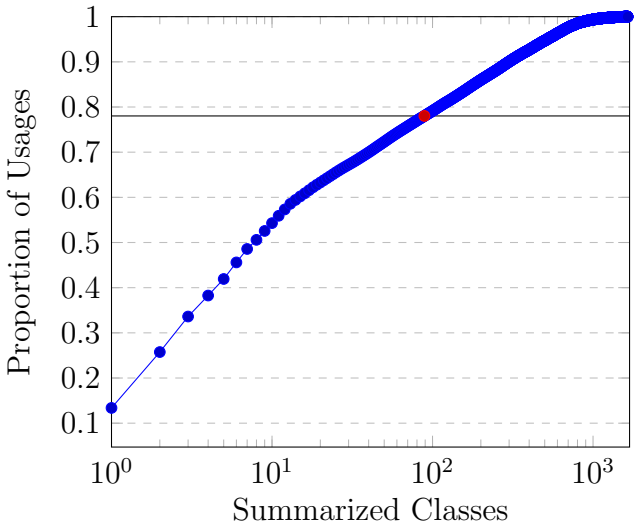


Figure 6.1: The number of summarized library classes and the proportion of total library usages captured by summarizing that number of classes.

[35] to construct a call graph for each application and totalled the direct usages for all library classes across the 9 DaCapo applications in our evaluation dataset. For our evaluation, we focus on summarizing as many of the most-used library classes as possible within a 12 hour time window. We chose a 12 hour time window because we envision BLUJ's summary generation as a component of an overnight build. Appendix A provides a table of the 100 library classes with the most direct usages, and Figure 6.1 shows the cumulative proportion of library usages that BLUJ is able to capture with respect to number of classes summarized.

6.2 Summary Generation

We performed BLUJ's summarization of each class in parallel across 32 physical threads, with a configured timeout of 20 minutes for each summarized API method. Table 6.1 shows the peformance results for summary generation by package. Appendix B shows the full results by class.

BLUJ generated summaries for 89 library classes within the allotted 12 hours, and the 89 summaries captured 78% of library usages (**RQ2**). The total peak heap memory usage across the full summary generation run was

Table 6.1: Summary generation statistics for library classes by package.

Package	Classes	CPU Time Used	Flows	Summary Size (MB)
java.util	33	62hr 6min	3309	66.03
java.lang	23	39hr 50min	2604	118.07
java.io	11	26hr 15min	1210	43.32
org.antlr.runtime	5	3hr 32min	743	13.99
java.awt	2	9hr 22min	360	5.41
java.awt.image	2	10hr 50min	274	3.49
java.net	2	13hr 0min	236	12.18
other	11	10hr 27min	2001	92.37

14.2 GB, or 444 MB on average per thread, suggesting that summary generation is not particularly memory intensive (**RQ1**). This indicates that on very high-powered systems, the generation phase could be further parallelized, perhaps even summarizing individual methods in parallel, to decrease summary construction time. The on-disk size of the 89 generated class summaries is 355MB.

6.3 Whole-program points-to analysis

To evaluate **RQ3**, we integrated HGASSs into the Boomerang flow-sensitive points-to analysis framework. The whole-program analysis initializes an SPDSs instance at every allocation site reachable from the program entry points and solves each SPDSs query independently. We measured performance on the basis of total analysis time and peak memory usage. We measured soundness and precision by comparing the size of the set of *reachable application states* for Boomerang and BluJ.

Definition 17. A *reachable application state* is an SPDSs control location $v@s_a \in V \times S$ such that s_a is a statement in the application code and there exists some configuration $\langle v@s_a, w_f\rangle \in post^*(\langle o@s_{alloc}, \epsilon\rangle)$ for arbitrary $w_f \in F^*$ and arbitrary object allocation site $o@s_{alloc}$.

Table 6.2 provides a summary of the results for each benchmark. While evaluating Boomerang and BluJ, we encountered a limitation of the Boomerang framework: Boomerang does not use a worklist algorithm to compute PDS

Table 6.2: Memory usage, analysis time, library analysis time, reached application states, and failed queries for each benchmark. Reported mean values are the arithmetic mean.

Benchmark		Analysis Time (S)		Mem. Usage (GB)		Lib. Analysis Time (S)		kStates Reached		Failed Queries	
		mean	st. dev.	mean	st. dev.	mean	st. dev.	mean	st. dev.	mean	st. dev.
avrora	Boomerang	461.4	32.7	19.7	1.7	0.84	0.06	1 278	36	1	0
	BluJ	1 220.2	196.1	18.1	1.4	0.34	0.02	855	44	74	2
batik	Boomerang	13.5	0.3	2.3	0.3	0.93	0.07	223	0	0	0
	BluJ	14.9	0.7	2.4	0.3	0.21	0.02	191	0	53	0
eclipse	Boomerang	287.5	32.8	19.2	3.7	8.61	2.12	2064	271	48	15
	BluJ	123.2	6.3	7.2	1.9	24.48	1.34	420	3	884	1
h2	Boomerang	9.5	0.1	1.7	0.3	0.76	0.06	161	0	0	0
	BluJ	16.9	0.3	2.3	0.2	0.29	0.01	125	0	47	0
luindex	Boomerang	129.1	13.8	10	1.8	5.83	0.47	1743	392	20	15
	BluJ	135.3	8.9	7.1	1.6	1.47	0.06	606	13	137	1
lusearch	Boomerang	30.3	0.6	3	0.2	0.74	0.04	514	0	0	0
	BluJ	60	1.3	3.8	0.6	0.41	0.02	545	3	78	0
pmd	Boomerang	16.7	0.7	2.5	0.1	0.69	0.03	311	0	0	0
	BluJ	17.7	0.4	2.5	0.1	0.3	0.01	238	0	89	0
sunflow	Boomerang	71.1	1	5.5	0.5	1.43	0.04	1148	9	1	1
	BluJ	63.3	0.9	4.1	0.3	0.37	0.03	623	0	125	0
xalan	Boomerang	9.9	0.3	1.9	0.1	0.62	0.08	160	0	0	0
	BluJ	12.4	0.2	2	0.3	0.19	0.01	119	0	46	0

reachability information but instead makes heavy use of the observer design pattern. For large programs like those in our evaluation dataset, the heavy use of this design pattern leads to deeply nested sequences of method calls and occasional stack overflow errors, even when the Java Virtual Machine is configured with the maximum stack size. Additionally, analysis execution is non-deterministic, and there is no way to predict which queries will fail with a stack overflow error on a particular analysis run. To mitigate this limitation, we executed BluJ and Boomerang five times for each benchmark and report the arithmetic mean for each metric.

For each benchmark, we show the analysis time, peak memory usage, reachable application states, and failed queries in Figure 6.2. The results indicate that there is generally not much advantage to using BluJ rather than Boomerang. This result initially surprised us, but a closer examination of the time that Boomerang actually spends analyzing library code for this set of benchmarks (Figure 6.3) reveals that there is not much performance to be gained by speeding up library analysis. For 8 of the 9 benchmarks, BluJ reduced library analysis time, but library analysis time was nearly inconsequential when compared to application analysis time. Additionally, BluJ

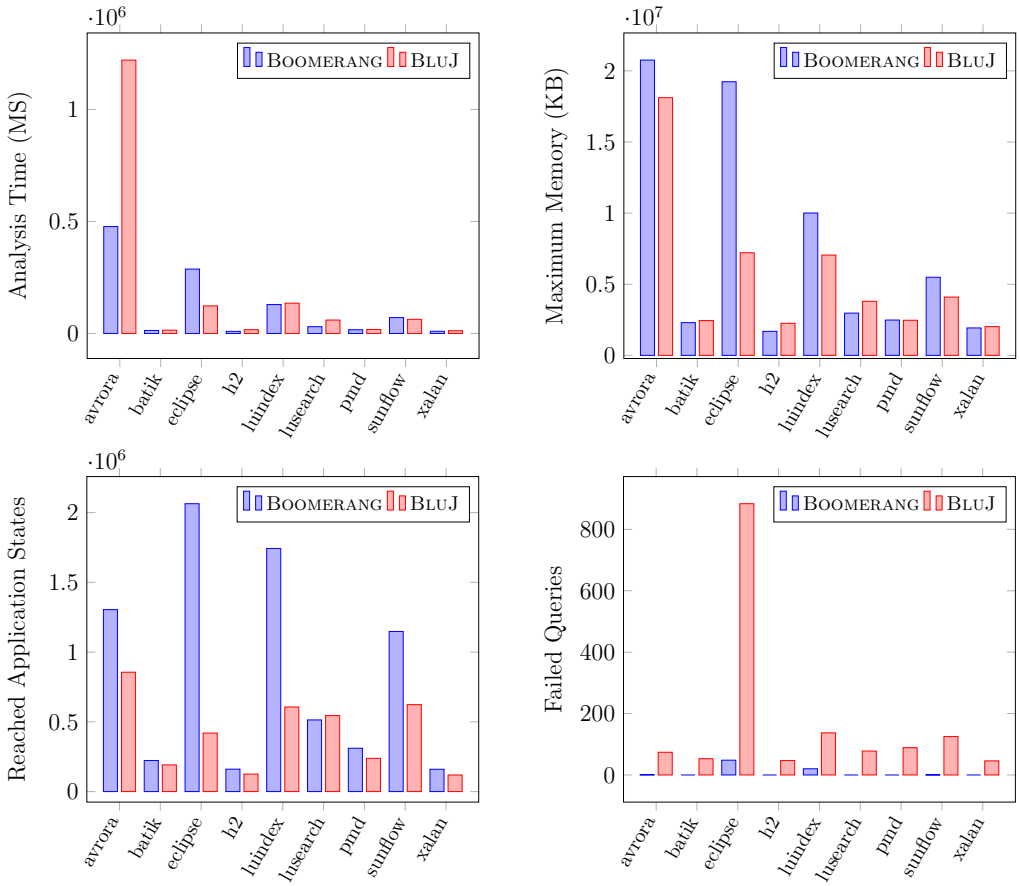


Figure 6.2: Analysis time, maximum heap memory usage, reached application states, and failed queries for BOOMERANG and BLUJ. Plotted values are the arithmetic means for the five analysis runs.

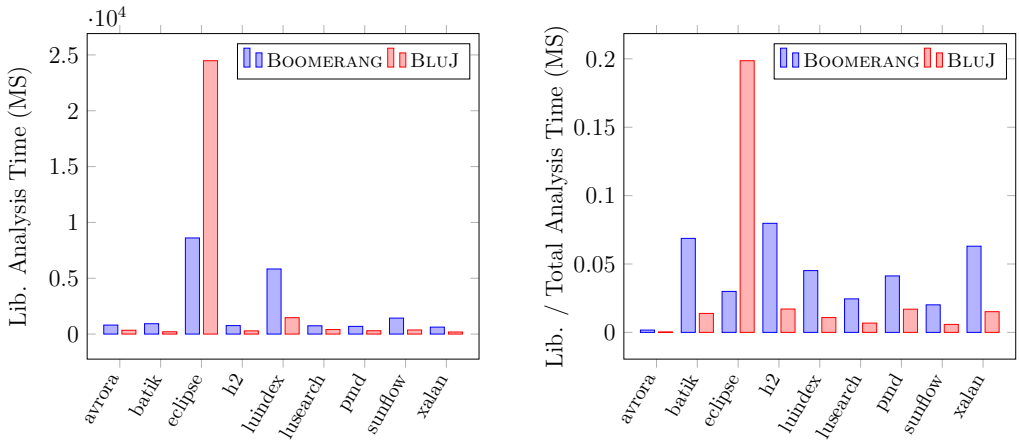


Figure 6.3: Library analysis time and library analysis time as a proportion of total analysis time for BOOMERANG and BLUJ. Plotted values are the arithmetic means for the five analysis runs.

creates performance overhead by checking to see if it can apply a summary at every application-to-library call edge, and this overhead more than offsets the reduction in library analysis time. The unusually low proportion of time spent analyzing library code runs contrary to results reported in existing studies, e.g., [2], [6], [45]. The following section discusses some possible reasons for this surprising result.

6.4 Discussion

While the benchmark source code contains a high number of library calls, both BOOMERANG and BLUJ determine the majority of them to be unreachable from the application entry point. This suggests that TAMIFLEX is unable to capture adequate reflection traces for the DaCapo benchmarks, the DaCapo harness does not provide high code coverage, or some combination of the two. The Jacoco code coverage tool [23] reports that of the 9 benchmarks in our corpus, only sunflow surpasses 20% coverage, with batik and xalan below 10% coverage [18]. Additionally, the TAMIFLEX booster agent (the portion of TAMIFLEX responsible for integrating reflection traces into a static analysis) only supports basic Java reflection mechanisms and ignores several types of reflection that occur in both the benchmark applications and the DaCapo harness.

We attempted to mitigate this problem by ignoring program entry points and setting all application classes as reachable. However, all analysis runs of both BOOMERANG and BLUJ failed with out-of-memory errors, and we were unable to obtain useful data. We expect that if we had been able to successfully perform the analysis with all application methods set as reachable, we would have found the proportion of analysis time spent analyzing library code to be more in line with existing research.

BLUJ's current implementation computes an under-approximation of BOOMERANG's reached application states, meaning it is not sound with respect to BOOMERANG. The majority of the states that BLUJ did not determine to be reachable were missed as a result of failed queries. Overall, BLUJ had 21 times as many queries fail due to stack overflow errors compared to BOOMERANG. Manual inspection of BLUJ and BOOMERANG's respective sets of reached states for the batik benchmark also revealed 4 successful queries for which BLUJ under-approximated the set of reachable states. While we are currently working to determine the cause of the increased stack overflows and additional unsoundness, we do not believe that addressing these issues will significantly change the answer to **RQ3**.

Chapter 7

Conclusion

Constructing data-flow summaries that are simultaneously precise, easily generated, and efficiently consumed by a client analysis is a difficult task. In this thesis, we attempted to address the problem of precise and efficient data-flow summaries through BluJ, a tool that generates context- and field-sensitive data-flow summaries using synchronized pushdown systems and heap graphs. We provided formal descriptions of how BluJ generates its HGASS summaries and how BluJ integrates them into an existing whole-program points-to analysis.

We successfully generated HGASSs for 89 library classes within 12 hours, indicating that HGASS generation is feasible in a production environment. For example, new summaries could be generated as part of a nightly build, if necessary.

We integrated our summaries into the Boomerang points-to analysis framework and evaluated the performance of BluJ against Boomerang. The results for analysis time and memory usage indicate that HGASS summaries, in their current formulation and implementation, cannot be efficiently consumed by an existing whole-program points-to analysis. The empirical evaluation also uncovered issues with soundness in BluJ's implementation that we are working to address.

While BluJ's summaries do not offer performance gains for generic Java applications, they may offer more benefit to analyses that target framework-heavy applications, e.g., Android applications. Future work could evaluate

BluJ on a suite of Android benchmarks to assess their feasibility.

BluJ could possibly be extended to *weighted* pushdown system. Weighted pushdown systems extend pushdown systems with an additional finite weight domain represented by an algebraic structure called an *idempotent semiring*. The weight domain can encode a variety of information for specific data-flow analyses, such as access control [50], constant propagation [42], and typestate analysis [54]. An extension of BluJ's pushdown systems to weighted pushdown systems would facilitate analysis-specific library summaries. The primary challenge of this work would lie in extending the heap graph portion of BluJ's summary generation framework with the appropriate weight domain and weight functions such that BluJ can compute weighted reachability information.

BluJ may also be applicable to dynamic languages such as JavaScript, provided that we modify BluJ to efficiently handle its language features. BluJ's notion of context-sensitivity would need to be extended to handle closures. Additionally, BluJ would need to soundly model the variety of mechanisms that JavaScript provides for asynchronous computation. BluJ may also be able to build on existing work that performs lightweight partial program analysis on individual JavaScript compilation units [29], [38].

References

[1] K. Ali and O. Lhoták, "Application-only call graph construction," in *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, J. Noble, Ed., ser. Lecture Notes in Computer Science, vol. 7313, Springer, 2012, pp. 688–712. DOI: 10.1007/978-3-642-31057-7_30. [Online]. Available: https://doi.org/10.1007/978-3-642-31057-7%5C_30.

[2] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program," in *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, G. Castagna, Ed., ser. Lecture Notes in Computer Science, vol. 7920, Springer, 2013, pp. 378–400. DOI: 10.1007/978-3-642-39038-8_16. [Online]. Available: https://doi.org/10.1007/978-3-642-39038-8%5C_16.

[3] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, R. S. Northcote, Ed., ACM, 1970, pp. 1–19. DOI: 10.1145/800028.808479. [Online]. Available: https://doi.org/10.1145/800028.808479.

[4] F. E. Allen, "Interprocedural analysis and the information derived by it," in *Programming Methodology, 4th Informatik Symposium, IBM Germany, Wildbad, September 25-27, 1974*, C. Hackl, Ed., ser. Lecture Notes in Computer Science, vol. 23, Springer, 1974, pp. 291–322. DOI: 10.1007/3-540-07131-8_31. [Online]. Available: https://doi.org/10.1007/3-540-07131-8%5C_31.

[5] L. O. Andersen, "Program analysis and specialization for the c programming language," DIKU, University of Copenhagen, Tech. Rep., 1994.

[6] S. Arzt and E. Bodden, "Stubdroid: Automatic inference of precise data-flow summaries for the android framework," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds., ACM, 2016, pp. 725–735. DOI: 10.1145/2884781.2884816. [Online]. Available: https://doi.org/10.1145/2884781.2884816.

[7] S. Arzt, S. Rasthofer, C. Fritz, *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds., ACM, 2014, pp. 259–269. DOI: 10.1145/2594291.2594299. [Online]. Available: https://doi.org/10.1145/2594291.2594299.

[8] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1996, San Jose, California, USA, October 6-10, 1996*, L. Anderson and J. Coplien, Eds., ACM, 1996, pp. 324–341. DOI: 10.1145/236337.236371. [Online]. Available: https://doi.org/10.1145/236337.236371.

[9] S. M. Blackburn, R. Garner, C. Hoffman, *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[10] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., ACM, 2011, pp. 241–250. DOI: 10.1145/1985793.1985827. [Online]. Available: https://doi.org/10.1145/1985793.1985827.

[11] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, A. W. Mazurkiewicz and J. Winkowski, Eds., ser. Lecture Notes in Computer Science, vol. 1243, Springer, 1997, pp. 135–150. DOI: 10.1007/3-540-63141-0_10. [Online]. Available: https://doi.org/10.1007/3-540-63141-0%5C_10.

[12] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, S. Arora and G. T. Leavens, Eds., ACM, 2009, pp. 243–262. DOI: 10.1145/1640089.1640108. [Online]. Available: https://doi.org/10.1145/1640089.1640108.

[13] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Z. Shao and B. C. Pierce, Eds., ACM, 2009, pp. 289–300. DOI: 10.1145/1480881.1480917. [Online]. Available: https://doi.org/10.1145/1480881.1480917.

[14] D. R. Chase, M. N. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, B. N. Fischer, Ed., ACM, 1990, pp. 296–310. DOI: 10.1145/93542.93585. [Online]. Available: https://doi.org/10.1145/93542.93585.

[15] R. Chatterjee, B. G. Ryder, and W. Landi, "Relevant context inference," in *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, A. W. Appel and A. Aiken, Eds., ACM, 1999, pp. 133–146. DOI: 10.1145/292540.292554. [Online]. Available: https://doi.org/10.1145/292540.292554.

[16] J. Cocke, "Global common subexpression elimination," in *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, R. S. Northcote, Ed., ACM, 1970, pp. 20–24. DOI: 10.1145/800028.808480. [Online]. Available: https://doi.org/10.1145/800028.808480.

[17] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings*, W. G. Olthoff, Ed., ser. Lecture Notes in Computer Science, vol. 952, Springer, 1995, pp. 77–101. DOI: 10.1007/3-540-49538-X_5. [Online]. Available: https://doi.org/10.1007/3-540-49538-X%5C_5.

[18] J. Dietrich, H. Schole, L. Sui, and E. D. Tempero, "Xcorpus - an executable corpus of java programs," *J. Object Technol.*, vol. 16, no. 4, 1:1–24, 2017. DOI: 10.5381/jot.2017.16.4.a1. [Online]. Available: https://doi.org/10.5381/jot.2017.16.4.a1.

[19] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv, "Precise and compact modular procedure summaries for heap manipulating programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds., ACM, 2011, pp. 567–577. DOI: 10.1145/1993498.1993565. [Online]. Available: https://doi.org/10.1145/1993498.1993565.

[20] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient algorithms for model checking pushdown systems," in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, E. A. Emerson and A. P. Sistla, Eds., ser. Lecture Notes in Computer Science, vol. 1855, Springer, 2000, pp. 232–247. DOI: 10.1007/10722167_20. [Online]. Available: https://doi.org/10.1007/10722167%5C_20.

[21] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*, IEEE Computer Society, 2011, pp. 289–298. DOI: 10.1109/CGO.2011.5764696. [Online]. Available: https://doi.org/10.1109/CGO.2011.5764696.

[22] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using CLA: A million lines of C code in a second," in *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, M. Burke and M. L. Soffa, Eds., ACM, 2001, pp. 254–263. DOI: 10.1145/378795.378855. [Online]. Available: https://doi.org/10.1145/378795.378855.

[23] *Jacoco*, https://github.com/jacoco/jacoco, 2022.

[24] *JUnit 5*, https://junit.org/junit5/, 2022.

[25] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica*, vol. 7, pp. 305–317, 1977. DOI: 10.1007/BF00290339. [Online]. Available: https://doi.org/10.1007/BF00290339.

[26] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *ACM Comput. Surv.*, vol. 49, no. 2, 29:1–29:47, 2016. DOI: 10.1145/2931098. [Online]. Available: https://doi.org/10.1145/2931098.

[27] U. P. Khedker, A. Sanyal, and A. Karkare, "Heap reference analysis using access graphs," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, p. 1, 2007. DOI: 10.1145/1290520.1290521. [Online]. Available: https://doi.org/10.1145/1290520.1290521.

[28] G. A. Kildall, "A unified approach to global program optimization," in *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, P. C. Fischer and J. D. Ullman, Eds., ACM Press, 1973, pp. 194–206. DOI: 10.1145/512927.512945. [Online]. Available: https://doi.org/10.1145/512927.512945.

[29] E. K. Kristensen and A. Møller, "Reasonably-most-general clients for javascript library analysis," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds., IEEE, 2019, pp. 83–93. DOI: 10.1109/ICSE.2019.00026. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00026.

[30] A. Lal and T. W. Reps, "Improving pushdown system model checking," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, T. Ball and R. B. Jones, Eds., ser. Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 343–357. DOI: 10.1007/11817963_32. [Online]. Available: https://doi.org/10.1007/11817963%5C_32.

[31] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, S. I. Feldman and R. L. Wexelblat, Eds., ACM, 1992, pp. 235–248. DOI: 10.1145/143095.143137. [Online]. Available: https://doi.org/10.1145/143095.143137.

[32] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection: Literature review and empirical study," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds., IEEE, 2017, pp. 507–518. DOI: 10.1109/ICSE.2017.53. [Online]. Available: https://doi.org/10.1109/ICSE.2017.53.

[33] O. Lhoták and K. A. Chung, "Points-to analysis with efficient strong updates," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, T. Ball and M. Sagiv, Eds., ACM, 2011, pp. 3–16. DOI: 10.1145/1926385.1926389. [Online]. Available: https://doi.org/10.1145/1926385.1926389.

[34] O. Lhoták and L. J. Hendren, "Context-sensitive points-to analysis: Is it worth it?" In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*, A. Mycroft and A. Zeller, Eds., ser. Lecture Notes in Computer Science, vol. 3923, Springer, 2006, pp. 47–64. DOI: 10.1007/11688839_5. [Online]. Available: https://doi.org/10.1007/11688839%5C_5.

[35] O. Lhoták and L. J. Hendren, "Scaling java points-to analysis using SPARK," in *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and*

Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings, G. Hedin, Ed., ser. Lecture Notes in Computer Science, vol. 2622, Springer, 2003, pp. 153–169. DOI: 10.1007/3-540-36579-6_12. [Online]. Available: https://doi.org/10.1007/3-540-36579-6%5C_12.

[36] B. Livshits, M. Sridharan, Y. Smaragdakis, *et al.*, "In defense of soundiness: A manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015. DOI: 10.1145/2644805. [Online]. Available: https://doi.org/10.1145/2644805.

[37] L. Luo, J. Dolby, and E. Bodden, "MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper)," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, A. F. Donaldson, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 21:1–21:25, ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.21. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10813.

[38] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of node.js applications," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds., ACM, 2021, pp. 29–41. DOI: 10.1145/3460319.3464836. [Online]. Available: https://doi.org/10.1145/3460319.3464836.

[39] T. W. Reps, "Undecidability of context-sensitive data-independence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 162–186, 2000. DOI: 10.1145/345099.345137. [Online]. Available: https://doi.org/10.1145/345099.345137.

[40] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, R. K. Cytron and P. Lee, Eds., ACM Press, 1995, pp. 49–61. DOI: 10.1145/199448.199462. [Online]. Available: https://doi.org/10.1145/199448.199462.

[41] T. W. Reps, M. Sagiv, and R. Wilhelm, "Shape analysis and applications," in *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*, Y. N. Srikant and P. Shankar, Eds., CRC Press, 2007, p. 12.

[42] T. W. Reps, S. Schwoon, and S. Jha, "Weighted pushdown systems and their application to interprocedural dataflow analysis," in *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, R. Cousot, Ed., ser. Lecture Notes in

Computer Science, vol. 2694, Springer, 2003, pp. 189–213. DOI: 10.1007/3-540-44898-5_11. [Online]. Available: https://doi.org/10.1007/3-540-44898-5%5C_11.

[43] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953, ISSN: 00029947. [Online]. Available: http://www.jstor.org/stable/1990888.

[44] A. Rountev, B. G. Ryder, and W. Landi, "Data-flow analysis of program fragments," in *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, O. Nierstrasz and M. Lemoine, Eds., ser. Lecture Notes in Computer Science, vol. 1687, Springer, 1999, pp. 235–252. DOI: 10.1007/3-540-48166-4_15. [Online]. Available: https://doi.org/10.1007/3-540-48166-4%5C_15.

[45] A. Rountev, M. Sharp, and G. Xu, "IDE dataflow analysis in the presence of large object-oriented libraries," in *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, L. J. Hendren, Ed., ser. Lecture Notes in Computer Science, vol. 4959, Springer, 2008, pp. 53–68. DOI: 10.1007/978-3-540-78791-4_4. [Online]. Available: https://doi.org/10.1007/978-3-540-78791-4%5C_4.

[46] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programming languages," in *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, G. Hedin, Ed., ser. Lecture Notes in Computer Science, vol. 2622, Springer, 2003, pp. 126–137. DOI: 10.1007/3-540-36579-6_10. [Online]. Available: https://doi.org/10.1007/3-540-36579-6%5C_10.

[47] S. Sagiv, T. W. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theor. Comput. Sci.*, vol. 167, no. 1&2, pp. 131–170, 1996. DOI: 10.1016/0304-3975(96)00072-2. [Online]. Available: https://doi.org/10.1016/0304-3975(96)00072-2.

[48] S. Sagiv, T. W. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," in *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, H. Boehm and G. L. S. Jr., Eds., ACM Press, 1996, pp. 16–31. DOI: 10.1145/

237721.237725. [Online]. Available: https://doi.org/10.1145/237721.237725.

[49] P. D. Schubert, B. Hermann, and E. Bodden, "Lossless, persisted summarization of static callgraph, points-to and data-flow analysis," in *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, A. Møller and M. Sridharan, Eds., ser. LIPIcs, vol. 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 2:1–2:31. DOI: 10.4230/LIPIcs.ECOOP.2021.2. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2021.2.

[50] S. Schwoon, S. Jha, T. W. Reps, and S. G. Stubblebine, "On generalized authorization problems," in *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*, IEEE Computer Society, 2003, p. 202. DOI: 10.1109/CSFW.2003.1212714. [Online]. Available: https://doi.org/10.1109/CSFW.2003.1212714.

[51] H. Seidl, R. Wilhelm, and S. Hack, *Compiler Design - Analysis and Transformation*. Springer, 2012, ISBN: 978-3-642-17547-3. DOI: 10.1007/978-3-642-17548-0. [Online]. Available: https://doi.org/10.1007/978-3-642-17548-0.

[52] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," New York Univ. Comput. Sci. Dept., Tech. Rep., 1978.

[53] Y. Smaragdakis, *Doop-benchmarks*, https://bitbucket.org/yanniss/doop-benchmarks/, 2021.

[54] J. Späth, "Synchronized pushdown systems for pointer and data-flow analysis," Ph.D. dissertation, University of Paderborn, Germany, 2019. [Online]. Available: https://nbn-resolving.org/urn:nbn:de:hbz:466:2-33957.

[55] J. Späth, K. Ali, and E. Bodden, "Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 48:1–48:29, 2019. DOI: 10.1145/3290361. [Online]. Available: https://doi.org/10.1145/3290361.

[56] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java," in *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, S. Krishnamurthi and B. S. Lerner, Eds., ser. LIPIcs, vol. 56, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 22:1–22:26. DOI: 10.4230/LIPIcs.ECOOP.2016.22. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2016.22.

[57] T. C. Spillman, "Exposing side-effects in a PL/I optimizing compiler," in *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28, 1971*, C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, Eds., North-Holland, 1971, pp. 376–381.

[58] B. Steensgaard, "Points-to analysis in almost linear time," in *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, H. Boehm and G. L. S. Jr., Eds., ACM Press, 1996, pp. 32–41. DOI: 10.1145/237721.237727. [Online]. Available: https://doi.org/10.1145/237721.237727.

[59] V. Sundaresan, L. J. Hendren, C. Razafimahefa, *et al.*, "Practical virtual method call resolution for java," in *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 2000, Minneapolis, Minnesota, USA, October 15-19, 2000*, M. B. Rosson and D. Lea, Eds., ACM, 2000, pp. 264–280. DOI: 10.1145/353171.353189. [Online]. Available: https://doi.org/10.1145/353171.353189.

[60] H. Tang, D. Wang, Y. Xiong, L. Zhang, X. Wang, and L. Zhang, "Conditional dyck-cfl reachability analysis for complete and efficient library summarization," in *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, H. Yang, Ed., ser. Lecture Notes in Computer Science, vol. 10201, Springer, 2017, pp. 880–908. DOI: 10.1007/978-3-662-54434-1_33. [Online]. Available: https://doi.org/10.1007/978-3-662-54434-1%5C_33.

[61] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei, "Summary-based context-sensitive data-dependence analysis in presence of callbacks," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, S. K. Rajamani and D. Walker, Eds., ACM, 2015, pp. 83–95. DOI: 10.1145/2676726.2676997. [Online]. Available: https://doi.org/10.1145/2676726.2676997.

[62] R. E. Tarjan, "Fast algorithms for solving path problems," *J. ACM*, vol. 28, no. 3, pp. 594–614, 1981. DOI: 10.1145/322261.322273. [Online]. Available: https://doi.org/10.1145/322261.322273.

[63] A. Utture and J. Palsberg, "Fast and precise application code analysis using a partial library," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, ACM, 2022, pp. 934–945. DOI: 10.1145/3510003.3510046. [Online]. Available: https://doi.org/10.1145/3510003.3510046.

Appendix A

Most Used Dacapo Library Classes

Library Class	Direct Usages
java.lang.StringBuilder	44 573
java.lang.StringBuffer	41 222
java.lang.String	26 199
java.lang.Throwable	15 485
java.lang.AbstractStringBuilder	12 220
java.util.ArrayList	12 220
java.lang.System	9 918
java.lang.Object	6 726
java.util.HashMap	6 568
java.lang.Integer	5 773
java.io.File	5 411
java.lang.Class	4 676
java.io.PrintStream	4 144
java.util.HashSet	2 867
java.util.Hashtable	2 609
java.lang.Math	2 288
java.util.Vector	2 252
java.util.ArrayList$Itr	2 175
java.lang.Boolean	1 865
java.lang.Thread	1 683
java.util.Arrays	1 626
java.lang.IllegalArgumentException	1 546
java.util.Properties	1 488
java.lang.NoClassDefFoundError	1 420
java.lang.Character	1 398
java.lang.Float	1 254
org.antlr.runtime.BufferedTokenStream	1 223
java.lang.Long	1 084
java.util.LinkedList	1 041
java.util.AbstractList$Itr	1 014
java.awt.geom.AffineTransform	953
java.io.FileInputStream	947
java.lang.RuntimeException	944
java.awt.Container	938
org.antlr.runtime.BitSet	934
org.antlr.runtime.BaseRecognizer	916
java.util.Locale	900
java.util.StringTokenizer	875
java.io.DataInputStream	851
java.util.LinkedList$ListItr	845
java.util.AbstractList$SubList$1	840
java.util.Vector$Itr	837
java.net.URL	830
org.antlr.runtime.tree.BaseTreeAdaptor	822

Library Class	Direct Usages
java.lang.Double	812
org.xml.sax.SAXException	765
java.io.BufferedReader	742
java.awt.image.Raster	741
java.awt.Component	725
com.ibm.icu.text.MessageFormat	714
java.io.PrintWriter	706
java.io.FileOutputStream	696
java.io.BufferedInputStream	688
java.util.AbstractMap$1$1	660
java.util.Collections	647
java.lang.AssertionError	638
javax.swing.JComponent	604
java.security.AccessController	588
java.util.LinkedHashMap$LinkedHashIterator	568
java.util.HashMap$HashIterator	568
java.io.DataOutputStream	550
java.io.RandomAccessFile	542
java.math.BigInteger	526
java.lang.Enum	510
sun.awt.util.IdentityLinkedList$ListItr	495
javax.imageio.metadata.IIOMetadataNode	491
org.antlr.runtime.ANTLRStringStream	491
java.lang.NullPointerException	481
org.w3c.dom.DOMException	475
java.lang.UnsupportedOperationException	474
java.util.ResourceBundle	468
org.antlr.runtime.DFA	441
java.util.Date	440
java.util.HashMap$Node	439
java.util.Collections$EmptyIterator	430
java.net.URI	427
java.util.Calendar	421
java.util.Stack	420
java.awt.Rectangle	409
java.util.AbstractCollection	407
java.util.AbstractMap$2$1	405
java.awt.image.BufferedImage	399
java.io.IOException	381
java.awt.Window	373
java.util.LinkedHashMap$LinkedKeyIterator	364
java.util.HashMap$KeyIterator	364
java.io.InputStreamReader	362
com.sun.org.apache.xerces.internal.impl.xs.opti.ElementImpl	361
java.lang.IllegalStateException	360
java.lang.ClassLoader	355
java.util.AbstractList	352
java.util.Arrays$ArrayItr	351
java.util.Collections$SynchronizedCollection	345
java.lang.Byte	341
javax.xml.transform.TransformerException	329
java.util.Collections$UnmodifiableCollection	327
java.lang.Short	327
java.util.Hashtable$Enumerator	326
sun.java2d.SunGraphics2D	325
java.util.ImmutableCollections$ListItr	321

Appendix B

Summary Generation

Class	Gen. Time (s)	Flows	Size (MB)
java.math.BigInteger	19 423.43	781	74.26
java.lang.Integer	15 508.10	758	49.90
java.lang.Long	15 489.25	607	32.70
java.awt.geom.AffineTransform	1 301.37	557	7.92
java.util.Properties	26 287.69	502	16.24
java.util.Vector	18 742.63	481	10.71
java.util.Collections	12 485.24	419	7.13
java.util.LinkedList	11 457.12	400	6.81
java.lang.StringBuffer	24 002.93	312	7.57
org.antlr.runtime.BaseRecognizer	10 693.55	299	5.59
javax.imageio.metadata.IIOMetadataNode	17.21	290	4.38
java.io.PrintWriter	16 048.82	267	20.79
org.antlr.runtime.BufferedTokenStream	932.00	235	4.87
java.util.HashMap	10 954.88	229	4.12
java.io.PrintStream	22 472.58	220	7.85
java.io.RandomAccessFile	5 766.35	213	4.06
java.awt.Container	33 664.05	211	3.68
java.util.Hashtable	16 469.86	208	3.17
java.net.URI	29 989.33	202	12.02
java.util.Calendar	7 225.43	182	3.89
java.util.ArrayList	9 351.80	175	3.04
java.awt.image.BufferedImage	14 410.50	154	1.97
java.awt.Rectangle	35.58	149	1.73
java.lang.Boolean	2 225.65	147	8.93
java.lang.Byte	7 867.50	136	6.32
org.antlr.runtime.tree.BaseTreeAdaptor	1.88	133	1.16
java.lang.Character	6 021.93	130	3.80
com.sun.org.apache.xerces.internal.impl.xs.opti.ElementImpl	0.30	126	0.93
java.io.FileInputStream	4 369.14	121	3.11
java.awt.image.Raster	24 581.06	120	1.52
java.lang.Throwable	9 735.91	120	3.42
java.util.AbstractList	7 207.81	117	2.05
java.util.HashSet	4 422.36	110	2.28
java.io.FileOutputStream	6 543.17	105	2.84
org.antlr.runtime.BitSet	1 088.67	103	1.89
java.io.DataOutputStream	2 718.84	103	2.56
javax.xml.transform.TransformerException	3 640.10	94	3.57
java.io.InputStreamReader	7 246.33	86	1.31
java.lang.Thread	21 411.67	69	1.38
org.antlr.runtime.DFA	0.33	65	1.42
java.util.LinkedList$ListItr	1 230.18	56	0.51
java.util.Hashtable$Enumerator	7.33	54	0.56
java.util.StringTokenizer	7 312.10	54	1.43
java.lang.Short	8 400.55	49	1.14
org.antlr.runtime.ANTLRStringStream	0.69	41	0.22
java.lang.ClassLoader	8 406.48	38	0.25

Class	Gen. Time (s)	Flows	Size (MB)
java.util.HashMap$Node	1 802.02	38	0.75
java.net.URL	16 800.48	34	0.16
java.lang.Object	311.61	33	1.12
java.io.BufferedReader	6 005.12	32	0.32
java.lang.Double	3 602.28	31	0.13
java.lang.Float	3 601.47	31	0.13
java.lang.System	10 813.71	29	0.39
java.util.ResourceBundle	14 406.80	29	0.24
java.util.Collections$UnmodifiableCollection	13 225.48	29	0.28
java.util.Date	15 600.70	27	0.40
java.io.BufferedInputStream	4 089.40	24	0.19
java.io.DataInputStream	18 025.84	24	0.19
java.util.AbstractCollection	10 805.63	23	0.26
java.util.Collections$SynchronizedCollection	16 800.51	22	0.14
java.util.LinkedHashMap$LinkedHashIterator	65.12	21	0.34
java.util.HashMap$HashIterator	54.34	21	0.34
java.lang.Enum	1 204.69	21	0.11
java.util.Stack	3 600.63	21	0.56
java.util.AbstractList$SubList$1	3 601.93	20	0.14
java.lang.AssertionError	1 203.48	16	0.15
java.lang.RuntimeException	1 202.56	15	0.12
java.lang.IllegalArgumentException	1 203.12	15	0.12
java.lang.IllegalStateException	1 203.12	15	0.12
java.lang.UnsupportedOperationException	4.88	15	0.12
java.io.IOException	1 204.39	15	0.12
sun.awt.util.IdentityLinkedList$ListItr	8.01	14	0.12
java.util.ArrayList$Itr	1 203.93	12	0.12
java.util.Vector$Itr	1 386.67	12	0.11
java.lang.NullPointerException	4.41	12	0.08
java.util.Collections$EmptyIterator	4.96	11	0.10
java.util.AbstractList$Itr	1 747.07	10	0.10
java.util.ImmutableCollections$ListItr	1 200.34	7	0.03
java.util.HashMap$KeyIterator	2.06	5	0.05
java.util.LinkedHashMap$LinkedKeyIterator	1.59	5	0.05
java.util.Arrays$ArrayItr	2.18	5	0.04
java.lang.NoClassDefFoundError	1.59	5	0.04
org.w3c.dom.DOMException	1.79	5	0.04
java.util.AbstractMap$2$1	2 417.50	2	0.01
java.util.AbstractMap$1$1	2 495.94	2	0.01
org.xml.sax.SAXException	0.52	1	0.00
java.security.AccessController	13 200.20	0	0.00
java.lang.Math	1.23	0	0.00
com.ibm.icu.text.MessageFormat	0.00	0	0.00