Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java

Johannes Späth[1], Lisa Nguyen Quang Do[*2], Karim Ali[3], and Eric Bodden[†4]

1 **Fraunhofer SIT**
`johannes.spaeth@sit.fraunhofer.de`

2 **Universität Paderborn and Fraunhofer IEM**
`lisa.nguyen@iem.fraunhofer.de`

3 **Technische Universität Darmstadt**
`karim.ali@cased.de`

4 **Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM**
`eric.bodden@uni-padeborn.de`

Abstract

Many current program analyses require highly precise pointer information about small, targeted parts of a given program. This motivates the need for demand-driven pointer analyses that compute information only where required. Pointer analyses generally compute points-to sets of program variables or answer boolean alias queries. However, many client analyses require richer pointer information. For example, taint and typestate analyses often need to know the set of *all aliases* of a given variable *under a certain calling context*. With most current pointer analyses, clients must compute such information through repeated points-to or alias queries, increasing complexity and computation time for them.

This paper presents Boomerang, a demand-driven, flow-, field-, and context-sensitive pointer analysis for Java programs. Boomerang computes rich results that include both the possible allocation sites of a given pointer (points-to information) and all pointers that can point to those allocation sites (alias information). For increased precision and scalability, clients can query Boomerang with respect to particular calling contexts of interest.

Our experiments show that Boomerang is more precise than existing demand-driven pointer analyses. Additionally, using Boomerang, the taint analysis FlowDroid issues up to 29.4x fewer pointer queries compared to using other pointer analyses that return simpler pointer information. Furthermore, the search space of Boomerang can be significantly reduced by requesting calling contexts from the client analysis.

1998 ACM Subject Classification F.3.2 – Logics and Meanings of Programs – Semantics of Programming Languages—Program Analysis

Keywords and phrases Demand-Driven; Static Analysis; IFDS; Aliasing; Points-to Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.22

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.2.1.12`

[*] Research was conducted while being at Fraunhofer SIT
[†] Research was conducted while being at Technische Universität Darmstadt and Fraunhofer SIT

```
1  context1(){
2    A d = new A();
3    A e = new A();
4    String f = source();
5    foo(d,e,f);
6  }

7  context2(){
8    A x = new A();
9    A y = x;
10   String z = noSource();
11   foo(x,y,z);
12 }

13 foo(A a, A b, String s){
14   a.f = s;
15   sink(b.f);
16 }
```

Figure 1 An example program that illustrates the problem of finding all aliases under a specific calling context.

1 Introduction

Static analysis nowadays is often used for vulnerability detection, finding bugs, and program understanding tools. This setting typically requires the underlying analyses —including pointer analyses— to compute information on-demand (i.e., only for the portions of the program that are of specific interest to the client that is querying them). In the case of pointer analyses, the different clients are interested in different types of pointer information [1, 28]. For example, a call graph analysis [26] approximates the runtime types of call receivers using points-to sets that contain all possible allocation sites of an object, and their runtime types. On the other hand, race-detection algorithms [18], typically need to know whether two given program variables may alias (i.e., point to the same memory location). Traditionally, pointer analyses can be categorized into either *points-to analyses* that compute points-to sets or *alias analyses* that give a boolean answer to an alias query. As we show in this work, computing only one kind of pointer information is ill-suited when supporting some client analyses. In particular, taint analysis [1] and typestate analysis [3, 16] have to post-process both types of pointer information, as they are interested in *all aliases* of a given variable.

Figure 1 shows an example program that illustrates this problem. In the case of a taint analysis starting at line 4, the taint flows into method `foo` and reaches the field write statement at line 14. To process this statement, the taint analysis must taint `a.f` and all of its aliases, which requires finding all aliases of `a`. Traditional alias analyses do not return this information directly, which is why the taint analysis has to query the alias analysis repeatedly: it must iterate over all existing local variables and determine if they alias with `a`. Additionally, a precise taint analysis should be able to restrict its pointer queries to specific calling contexts. In the example, a false positive is reported at line 15, if the taint analysis merges `context1`, where a taint is flowing but no aliases are created, and `context2`, where `a` and `b` effectively alias (due to the assignment at line 9) but no taint is issued.

As shown in our experiments, repeated queries impede performance and context-insensitive queries impede precision. Ideally, a taint analysis should be able to issue a unique pointer query at line 14 for the variable `a` *under a specific, client-defined calling context* and directly obtain *all possible aliases* of `a`. The result of such a query would be the alias set `{a}` under `context1()` (or `{a,b}` under `context2()` if that query is issued instead). Due to the lack of such sophisticated pointer analyses, some client analyses are tightly integrated with custom, client-specific pointer analyses [1, 28]. This drastically complicates their coordination and prevent the reuse of these custom pointer analysis.

In this paper, we present Boomerang, the first reusable on-demand pointer analysis that, for a given variable, computes both *points-to* information and *all alias* information. Boomerang is flow-, field-, and context-sensitive. For each query, Boomerang performs a backward analysis to collect points-to information and then proceeds with a forward analysis to determine all access graphs [13] that point to the discovered allocation sites. For the program in Figure 1, if a client queries Boomerang at line 14 for variable `a` under `context1()`, Boomerang first searches backwards for all related allocation sites, discovering the allocation at line 2. The forward analysis then computes that, in `foo` under `context1()`, only variable `a` points to that allocation site.

To increase efficiency, Boomerang is built on top of the IFDS framework for Inter-procedural Finite Distributive Subset problems [19]. This design choice allows Boomerang to store and reuse the results of intra-procedural computations in the form of highly reusable, extremely fine-grained summaries that reason about individual access graphs. Since IFDS can only solve subset problems with distributive flow functions—and pointer analysis is known to be a non-distributive problem—Boomerang uses IFDS only for sub-tasks that can be expressed in a distributive way. Non-distributive fragments of the analysis are handled through additional iterations at *points of indirection*, typically accesses to the heap and at context switches.

We have evaluated the precision and recall of Boomerang using PointerBench, a novel micro-benchmark suite created to evaluate pointer analyses by indicating aliases/non-aliases pairs, as well as points-to sets. In comparison to the demand-driven points-to analysis by Sridharan and Bodík (SB) [23] and the demand-driven alias analysis by Yan et al. (DA) [30], Boomerang reports fewer false-positives with respect to alias pairs and allocation sites. We also compare the effect of using Boomerang, SB, and DA in the client taint analysis FlowDroid. On average, FlowDroid issues 19.9x and 29.4x more queries when using DA and SB, respectively, compared to using Boomerang. Moreover, since SB and DA are less precise than Boomerang, they cause FlowDroid to report more false positives.

Boomerang applies the notion of context-resolution by querying the client analysis for calling contexts as it computes pointer information. Unlike previous approaches that visit all contexts, Boomerang only considers contexts of interest to the client. This improves both the precision of the client (as shown in Figure 1 with the false positive at line 15) and the overall scalability (by propagating along less calling contexts). To evaluate the efficiency of context-resolution, we measure the percentage of contexts filtered out by FlowDroid, a taint analysis client, when using Boomerang. Our experiments show that FlowDroid filters out up to 96% of the calling contexts, which represents a significant computational saving for Boomerang.

To summarize, this paper makes the following contributions:

- We present Boomerang, a demand-driven flow- and context-sensitive pointer analysis that provides both points-to and all alias information. To our knowledge, this is the first analysis that directly provides both types of pointer information.
- We evaluate the precision and recall of Boomerang compared to the state-of-the-art demand-driven pointer analyses on PointerBench, a micro-benchmark suite for pointer analyses.
- We compare the effect of using Boomerang to using other demand-driven pointer analyses when integrated into a taint analysis client.

2 Background

IFDS is a framework for solving inter-procedural finite distributive subset problems [19]. If an analysis can be expressed as such, the framework reduces it to a reachability problem on an *exploded supergraph*, a directed graph representing the analyzed program. Figure 2 shows the exploded supergraph of an example Java program. Throughout this section, we

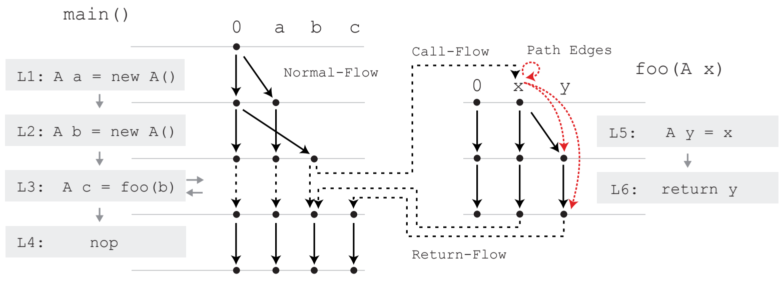


Figure 2 Creation of the exploded supergraph and summary edges in IFDS.

will use this example to explain how a whole-program points-to analysis can be computed using IFDS. For the sake of simplicity, we omit field accesses.

Data-Flow Domain

An IFDS-based analysis defines a *data-flow domain* D, composed of the *data-flow facts* whose validity is inferred by the analysis at each statement of the analyzed program. In the case of pointer analyses, facts typically represent references to objects in the analyzed program, but one can also encode richer information like typestate information [16]. In Figure 2, the fact `0` represents the tautological fact that always holds. It is the root of the exploded super graph. Before statement **L3**, facts `a` and `b` hold because they are reachable from `0`—both variables may point to objects created earlier in the program. Fact `c`, on the other hand, only holds after **L3** since it has not been defined before.

Flow Functions

In addition to the domain D, the analysis has to define *flow functions*. An IFDS flow function is a mapping from D to 2^D. It transfers a single fact to a (possibly empty) set of facts at the successors of a given statement. The composition of the flow functions incrementally creates the exploded supergraph. Flow functions can be of one of four types:

- **Normal flow functions** are applied to every non-call statement. For example, the flow function at statement **L2** transfers `a` to `a` because `a` continues to point to the same object. At the same statement, the flow function applied to `0` generates `b`, modeling a new points-to relationship for `b`.
- **Call flow functions** occur at invoke statements. They map facts from a caller's scope to the callee's scope such as replacing actual arguments by formal parameters.
- **Return flow functions** are the inverse functions of the call-flow functions, mapping the facts back to the scope of the caller.
- **Call-to-return flow functions** locally propagate the information that is not affected by a call (represented by the straight dashed edges in method `main` in Figure 2).

At control-flow merge points, IFDS merges the result of all incoming flow functions using the set union: a node is reachable if it is reachable along *any* path through the exploded supergraph. IFDS assumes that the flow functions are distributive with respect to set union (i.e., $f(A \cup B) = f(A) \cup f(B)$). Distributivity is a key property of the framework as it enables effective reuse of fine-grained per-fact procedure-summaries [19].

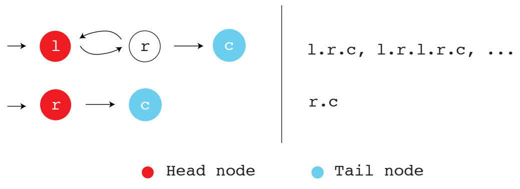


Figure 3 Access graph and access path correlation.

Path Edges

In the exploded supergraph, an existing fact holds (i.e., is reachable) if its predecessors are. This means that the solution to the IFDS problem is given by the facts reachable from the root fact `0`. To compute reachability, IFDS incrementally constructs *path edges* within the scope of methods. A path edge has the form $\langle s, d_1\rangle \to \langle t, d_2\rangle$, where s and t are statements and $d_1, d_2 \in D$. All path edges are intra-procedural and start from a method's entry point. The existence of a path edge summarizes a node's reachability in the exploded supergraph in relative terms: d_2 is reachable at t in every calling context in which d_1 is reachable at the method entry point s.

Figure 2 illustrates the construction of path edges in method `foo` (represented by red dotted arrows). First, IFDS creates a path edge from the starting point of the method, **SP**: $\langle \mathbf{SP}, x\rangle \to \langle \mathbf{SP}, x\rangle$. This is then extended to $\langle \mathbf{SP}, x\rangle \to \langle \mathbf{L5}, y\rangle$ and finally to $\langle \mathbf{SP}, x\rangle \to \langle \mathbf{L6}, y\rangle$. Unlike IFDS, path edges in Boomerang do not necessarily start from a method's entry point. We allow path edges to also start at allocation sites and call sites. This way, Boomerang can encode the reachability of allocation sites into the path edges. We discuss this approach in more detail in Section 3.4.

The path edges ending at a method's exit point are turned into *method summaries*, which avoids any re-evaluation of the same method for the same incoming facts. Contexts are thus quantified-over symbolically: a summary edge is applicable to every context in which the edge's source fact holds.

Access Graphs

To support field-sensitivity (i.e., distinguishing fields not only by their name, but also by their base object), Boomerang models data-flow facts using *access graphs* [13]. An access graph is defined as follows: let $\mathcal{L}$ be the set of all variables of the analyzed program and $\mathcal{F}$ the set of all existing fields of all classes within the program. An *access graph* has two main elements: (1) the *base*, a local variable $l \in \mathcal{L}$, and (2) the *field graph*, a directed graph of nodes from $\mathcal{F}$ that represent field accesses. If the field graph is empty, the data-flow fact represents a plain local variable. Otherwise, the field graph determines through which field accesses an object of interest is reachable. If the field graph is not empty, two nodes (possibly the same) are marked as the *head* and the *tail*. In addition, for each node n within the graph, there exists a path from the head to the tail passing through n. A field graph can then be uniquely identified by its edge set, its head, and its tail. We use $[l, s, E, t] \in \mathcal{L} \times \mathcal{F} \times 2^{(\mathcal{F}\times\mathcal{F})} \times \mathcal{F}$ to denote an access graph with local variable l, head node s, tail t, and edge set of the field graph E. We abbreviate the domain of all access graphs as $\mathcal{APG}$. Throughout the rest of the paper, we overset variables with $\widetilde{\ }$ as a shorthand notation for an access graph.

An access graph corresponds to (infinitely) many *access paths*: all paths obtained by traversing the field graph from the head to the tail. Figure 3 shows two access graphs and their corresponding sets of access paths.

We use operations on access graphs presented in earlier work [13]. These operations capture the transformations of the access graphs mostly at field read and write statements, where fields are consumed or added to the graph.

- **Remove Head:** for each successor field s' of the head s, we derive a new access graph with s' as the head. The appropriate edge is removed, if s is not part of a loop. This operation assumes a non-empty field graph.

$$\ominus[y,s,E,t] := \begin{cases} \{[y,s',E,t]\}, & \text{if } s \text{ is in a loop,} \\ \{[y,s',E \setminus \{(s,s')\},t] \mid (s,s') \in E\}. & \end{cases}$$

- **Remove Tail:** a complementary operation that moves the tail to its predecessor while removing the appropriate edge. We use $[y,s,E,t]\ominus$ to represent this operation.
- **Prepend Head:** connects the current head with a new head by adding the appropriate edge, and returns a single access graph:

$$f \oplus [y,s,E,t] := [y,f,E \cup \{(f,s)\},t]$$

  If the field graph was empty, the operation returns the access graph $[y,f,\varnothing,f]$.
- **Append Tail:** appends the field to the end of the access graph.

$$[y,s,E,t] \oplus f := [y,s,E \cup \{(t,f)\},f].$$

- **Concatenate:** appends the field graph from the second operand to the first one. Both field graphs must not be empty and the base is taken from the left operand.

$$[a,s_1,E_1,t_1] \uplus [b,s_2,E_2,t_2] = [a,s_1,E_1 \cup E_2,t_2]$$

All of those operations additionally perform a cleanup to remove nodes which no longer reside on a path between the head and tail nodes.

3 Boomerang

A Boomerang query is denoted by $(s,\widetilde{a})$ – compute pointer information for the access graph $\widetilde{a}$ at the program statement s. Boomerang performs a staged computation: a *backward* pass followed by a *forward* pass. For both passes, Boomerang uses the IFDS framework to ensure flow- and context-sensitivity and exploit the performance gains of using point-wise procedure summaries. The backward pass traverses the control-flow graph (CFG) backwards to discover the possible allocation sites that the queried access graph $\widetilde{a}$ may point to. Starting at those statements, the forward pass then collects the access graphs that, at the query statement s, may also point to the same allocation sites (i.e., may alias with $\widetilde{a}$). In this section, we assume a query to be *context-free* – propagation is only allowed within the transitively reachable callees of the method where the query is issued. In Section 4.2, we describe how allocation sites within callers are detected by the combination of multiple context-free queries. By that, Boomerang only generates the part of the supergraph required to compute results and, if available, reuses parts of the supergraph it computed from previous queries. We use $\overset{b}{\to}$ to denote the backward path edges and $\overset{f}{\to}$ to denote the edges of the forward pass.

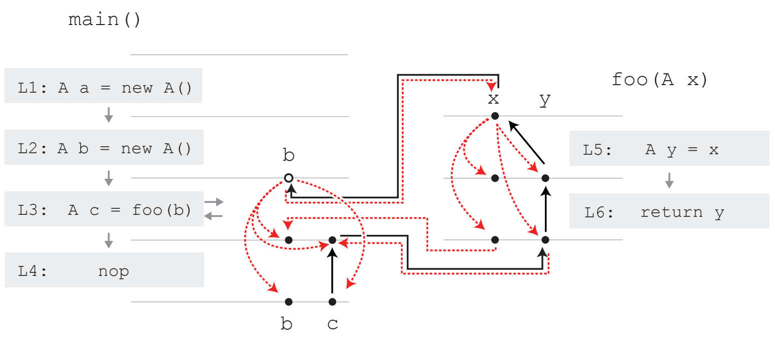


Figure 4 Constructing the exploded supergraph for the example in Figure 2.

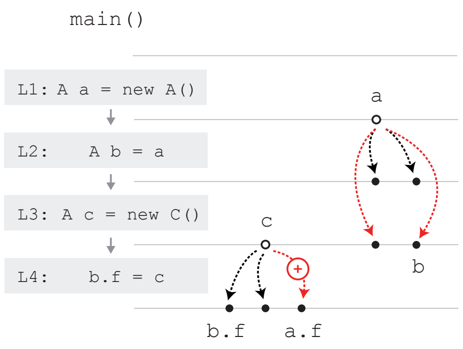


Figure 5 Handling field writes in Boomerang.

3.1 Basic Example

Figure 4 shows, for the example in Figure 2, the parts of the exploded super graph that have to be constructed to answer the query $(\mathbf{L4}, \texttt{c})$. At first, Boomerang starts a backward analysis (solid edges) that follows the definition chain of `c`, which results in analyzing the method `foo`. When the backward analysis reaches the allocation site at **L2**, Boomerang starts a forward analysis (dotted edges) that backtracks the path previously taken by the backward analysis. Boomerang discovers `b` and `c`, and reports that those variables may alias as they may both point to the object allocated at **L2**.

In Boomerang, the forward path edges are designed to hold the points-to and alias information. In the example, the forward path edges at statement **L4** are $\langle \mathbf{L2}, \texttt{b}\rangle \overset{f}{\to} \langle \mathbf{L4}, \texttt{b}\rangle$ and $\langle \mathbf{L2}, \texttt{b}\rangle \overset{f}{\to} \langle \mathbf{L4}, \texttt{c}\rangle$. Both edges share the same origin, $\langle \mathbf{L2}, \texttt{b}\rangle$ and by construction the data-flow facts of the targets may alias. Hence, at statement **L4** the variables `b` and `c` may alias and both are allocated at **L2**. To compute the alias information of `c`, Boomerang does not construct the part of the exploded super graph concerning variable `a`, avoiding the unnecessary effort that a whole-program analysis would make. As a side effect, this computation creates reusable backward and forward summaries for `foo`.

3.2 Handling Field Accesses

The flow functions of a precise pointer analysis are non-distributive [7]. They introduce a level of *indirection* that occurs, for instance, at *field write* statements. In Figure 5, to precisely reason about all aliases of `b.f` at the write at **L4**, the analysis must also take into

```
1: procedure BOOMERANG($s$, $\widetilde{v}$)
2:     $\overline{P}$ = PROPAGATE($\langle s,\widetilde{v}\rangle \xrightarrow{b} \langle s,\widetilde{v}\rangle$)          ▷ backward analysis
3:     do
4:         $p = \overline{P}.pop()$
5:         $\overline{N}$ = HANDLEPOI($p$)          ▷ triggers forward or backward analyses
6:         $\overline{P}.addAll(\overline{N})$
7:     while $\overline{P} \neq \varnothing$
8:     return $Res(s,\widetilde{v})$
9: end procedure
```

Figure 6 The main algorithm of BOOMERANG.

account the aliases of the the base variable `b`. In that case, alias information can only be retrieved indirectly. This is an example of a point of indirection, denoted by $\mathcal{POI}$. We discuss points of indirection in more detail in Section 4.1.

A sound treatment of those indirections raises the need for an *outer fixed-point iteration* around the forward and backward propagations. Such a handling of the $\mathcal{POI}s$ allows BOOMERANG to reuse summaries within the inner, distributive propagations, while at the same time computing a solution to a problem that is non-distributive as a whole. Figure 6 presents the pseudo-code for the outer fixed-point iteration. The process bootstraps the inner backward analysis with an initial self-loop (using the IFDS method PROPAGATE [17]). The propagated edge triggers the backward solver to find the object allocations that may flow into $\widetilde{v}$. The flow functions discover any $\mathcal{POI}s$ they find along the way. This yields the initial set $\overline{P}$. Once the backward propagation is finished, each of the newly discovered $\mathcal{POI}$ is handled by HANDLEPOI. Depending on the type of the $\mathcal{POI}$, the appropriate handler is executed. These handlers can trigger forward or backward analyses and can find new $\mathcal{POI}s$ (denoted $\overline{N}$) that are added to the set $\overline{P}$. This is repeated until a fixed point is reached.

In Figure 5, for the query of `c` at **L4**, let us assume that the backward analysis has already discovered the allocation site at line **L3**. In the forward pass, `c` flows to `b.f` at the field write statement in **L4** —a $\mathcal{POI}$. In HANDLEPOI, a sub-query to BOOMERANG then requests all aliases of `b` at **L3**, which returns the set $\{$`a`$,$`b`$\}$. Finally, the original query is continued by adding the dashed forward path edge marked with $\odot$ due to the discovered alias `a`.

When all $\mathcal{POI}s$ have been discovered and handled, the outer algorithm returns $Res(s,\widetilde{v})$, the solution to the query. In the example from Figure 5, the query $(\mathbf{L4},$ `c`$)$ results in the mapping $\langle \mathbf{L3},$ `c`$\rangle \mapsto \{$`c`$,$`a.f`$,$`b.f`$\}$. Thus, `a.f`, `b.f` and `c` all alias with each other, as their allocation site is statement **L3**. Section 3.4 describes the construction of $Res(s,\widetilde{v})$ in more detail.

3.3 Flow Functions

In BOOMERANG, flow functions are of type $\mathcal{S} \times \mathcal{APG} \to 2^{\mathcal{APG}}$. They take an access graph and a statement as input, and output a set of access graphs. We represent a flow function for the backward problem as $[\![s]\!]_b(\widetilde{\alpha})$ where s is a statement and α is an access graph. A flow function for the forward problem is similarly represented by $[\![s]\!]_f(\widetilde{\alpha})$.

The analysis is conducted on a three-address-code representation in which each statement contains at most one field dereference. We only define the flow functions for statements that affect the access graph being propagated as shown in Table 7a. To simplify the notations, we assume that invoke statements have at most one parameter.

Statement	Notation
Allocation site	$x \leftarrow \mathbf{new}$
Assign statement	$x \leftarrow y$
Field read	$x \leftarrow y.f$
Field write	$x.f \leftarrow y,$
Invoke statement	$r \leftarrow c.m(p)$

(a) Handled Statements.

$\mathcal{POI}$	Example	Symbol	Analysis
Allocation site	`a = new`	[**Alloc**]	Backward
Field read	`a = b.f`	[**Read**]	Backward
Alias on call	`c.m(args)`	[**Call**]	Backward
Alias on return	`c.m(args)`	[**Return**]	Forward
Field write	`a.f = b`	[**Write**]	Forward

(b) Points of indirection.

Figure 7 The statements handled by Boomerang and the points of indirection.

The definition of the flow functions is straightforward and simply follows the assignments of variables. Evaluating a flow function, causes new $\mathcal{POI}s$ to be discovered (shown next to the flow functions definitions). The $\mathcal{POI}s$ are handed over to the outer fixed-point iteration for later processing. Table 7b enumerates the types of $\mathcal{POI}s$ and which analysis discovers them. Section 4 describes how the $\mathcal{POI}s$ are processed in the outer fixed-point iteration.

3.3.1 Backward Analysis

For simplicity, we use $\widetilde{\alpha}$ as the argument of the flow function and express $\widetilde{\alpha}$ as $[v, s, E, t]$. We use $\widetilde{\alpha}_{>0}$ for the cases where the field graph cannot be empty, and $\widetilde{\alpha}_{=0}$ when the graph must be empty.

Normal-Flow Functions

$$[\![x \leftarrow y]\!]_b(\widetilde{\alpha}) = \begin{cases} \{[y, s, E, t]\} & \text{if } v = x \\ \{\widetilde{\alpha}\} \end{cases}$$

$$[\![x.f \leftarrow y]\!]_b(\widetilde{\alpha}) = \begin{cases} \ominus[y, s, E, t] & \text{if } v = x \wedge s = f \\ \{\widetilde{\alpha}\} \end{cases}$$

$$[\![x \leftarrow \mathbf{new}]\!]_b(\widetilde{\alpha}) = \begin{cases} \varnothing & \text{if } v = x \wedge \widetilde{\alpha}_{=0} \quad [\textbf{Alloc}] \\ \varnothing & \text{if } v = x \wedge \widetilde{\alpha}_{>0} \\ \{\widetilde{\alpha}\} \end{cases}$$

$$[\![x \leftarrow y.f]\!]_b(\widetilde{\alpha}) = \begin{cases} f \oplus [y, s, E, t] & \text{if } v = x \quad [\textbf{Read}] \\ \{\widetilde{\alpha}\} \end{cases}$$

At assignment statements of the form $x \leftarrow y$ where the access graph's local variable is x, the flow function replaces the base value of the access graph with y and keeps the whole field graph. For a field write statement $x.f \leftarrow y$, if the local variable and the first field of the access graph matches $x.f$, the base of the access graph is replaced by y and its first field is removed. At a field read statement, in contrast, the field f is prepended to the field graph. Additionally, if the flow functions discover $\mathcal{POI}s$ of type [**Read**] or [**Alloc**], they are added to the outer algorithm of Boomerang for later processing.

Call-Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_b(\widetilde{\alpha}) = \begin{cases} \{[\texttt{retVal}, s, E, t]\} & \text{if } v = r \quad [\textbf{Call}] \\ \{[\texttt{this}, s, E, t]\} & \text{if } v = c \wedge \widetilde{\alpha}_{>0} \quad [\textbf{Call}] \\ \{[\texttt{arg}, s, E, t]\} & \text{if } v = p \wedge \widetilde{\alpha}_{>0} \quad [\textbf{Call}] \\ \emptyset \end{cases}$$

In a backward analysis, the call-flow functions transfer facts from the call site to all possible return statements of the callees. In Boomerang, the backward call-flow functions map all access graphs $\widetilde{\alpha}$ for which the field graph is not empty. The base is then either the `this` variable or the argument to the callees. Java uses a call-by-value semantics, which is why a callee cannot redefine its parameters (including `this`). For this reason Boomerang propagates parameters only with non-empty field graphs. As a side effect, the backward call-flow function further registers a [**Call**].

Return-Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_b(\widetilde{\alpha}) = \begin{cases} \{[c, s, E, t]\} & \text{if } v = \texttt{this} \wedge \widetilde{\alpha}_{>0} \\ \{[p, s, E, t]\} & \text{if } v = \texttt{arg} \wedge \widetilde{\alpha}_{>0} \\ \emptyset \end{cases}$$

The return-flow functions map the access graphs back to the appropriate scope at the call sites. The access graphs representing the `this` value or a parameter that does not have any field accesses are not mapped back to the caller scope. They are handled by the call-to-return flow functions.

Call-to-Return Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_b(\widetilde{\alpha}) = \begin{cases} \emptyset & \text{if } v = r \\ \emptyset & \text{if } v \in \{c, p\} \wedge \widetilde{\alpha}_{>0} \\ \{\widetilde{\alpha}\} \end{cases}$$

Every access graph that is propagated by a call-flow function to hold within the callee is killed in the corresponding call-to-return flow function. These access graphs naturally flow out of the callees at the call site (via the return-flow function), if they are neither allocated nor overwritten within callees.

3.3.2 Forward Analysis

The forward flow functions are used to find all access graphs pointing to a given allocation site. The forward analysis is bootstrapped at an allocation site with an access graph having the allocated variable as base and an empty field graph.

Normal-Flow Functions

$$[\![x \leftarrow y]\!]_f(\widetilde{\alpha}) = \begin{cases} \{[x,s,E,t],\widetilde{\alpha}\} & \text{if } v = y \\ \varnothing & \text{if } v = x \\ \{\widetilde{\alpha}\} \end{cases}$$

$$[\![x.f \leftarrow y]\!]_f(\widetilde{\alpha}) = \begin{cases} \{\widetilde{\alpha}\} \cup f \oplus [x,s,E,t] & \text{if } v = y \quad [\textbf{Write}] \\ \varnothing & \text{if } v = x \wedge s = f \\ \{\widetilde{\alpha}\} \end{cases}$$

$$[\![x \leftarrow \textbf{new}]\!]_f(\widetilde{\alpha}) = \begin{cases} \varnothing & \text{if } v = x \\ \{\widetilde{\alpha}\} \end{cases}$$

$$[\![x \leftarrow y.f]\!]_f(\widetilde{\alpha}) = \begin{cases} \{\widetilde{\alpha}\} \cup \ominus[x,s,E,t] & \text{if } v = y \wedge s = f \\ \varnothing & \text{if } v = x \\ \{\widetilde{\alpha}\} \end{cases}$$

Whenever an assignment is encountered and its right-hand side matches the current fact, a new access graph with the prefix of the left-hand side is derived and propagated. In the forward flow functions, the access graph $\widetilde{\alpha}$ flowing into the function is maintained, as long as the left-hand side of an assign statement does not override it. $\mathcal{POI}s$ of type [**Write**] are detected at each field write statement.

Call-Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_f(\widetilde{\alpha}) = \begin{cases} \{[\texttt{this},s,E,t]\} & \text{if } v = c \\ \{[\texttt{arg},s,E,t]\} & \text{if } v = p \\ \varnothing \end{cases}$$

In the forward analysis, the call-flow functions map facts from call sites to the start points of the target methods. It is important to note that the access graphs with an empty field graph and having variable c or any parameter p as their base are also propagated into the callees. These access graphs can generate aliases within those callees. This is different from the backward analysis where facts that do not involve fields are not propagated into the callees.

Return-Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_f(\widetilde{\alpha}) = \begin{cases} \{[c,s,E,t]\} & \text{if } v = \texttt{this} \quad [\textbf{Return}] \\ \{[p,s,E,t]\} & \text{if } v = \texttt{arg} \quad [\textbf{Return}] \\ \{[r,s,E,t]\} & \text{if } v = \texttt{retVal} \quad [\textbf{Return}] \\ \varnothing \end{cases}$$

The return-flow functions map the base of the access graph back to the appropriate variable in the calling-context. As we will see later in Section 4, a $\mathcal{POI}$ of type [**Return**] needs to be handled here because of possible new aliases in the new scope.

Call-to-Return Flow Functions

$$[\![r \leftarrow c.m(p)]\!]_f(\widetilde{\alpha}) = \begin{cases} \varnothing & \text{if } v = r \\ \varnothing & \text{if } v \in \{c, p\} \wedge \widetilde{\alpha}_{>0} \\ \{\widetilde{\alpha}\} & \end{cases}$$

In the call-to-return flow function, an access graph is killed if it has a non-empty field graph and its base is the call receiver or a parameter. Such access graphs are handled by the call-flow functions and flows into the callees.

3.4 Path Edges

In IFDS, a path edge is a general summary between a fact at the method's first statement and an arbitrary node of the exploded supergraph belonging to the same method. In Boomerang, we define path edges as IFDS path edges between any two arbitrary nodes of the same method. The first statement restriction is loosened to be able to encode points-to information in the path edges. For example, when the backward analysis finds an allocation site ([**Alloc**] in the backward normal-flow functions), then the path edge $\langle \texttt{a = new}, \texttt{a}\rangle \xrightarrow{f} \langle \texttt{a = new}, \texttt{a}\rangle$ is added to the forward analysis. During the forward propagation where IFDS extends the added path edge, the origin of the path edge $\langle \texttt{a = new}, \texttt{a}\rangle$ is maintained for the derived path edges in order to keep track of the new allocation site.

For each derived forward path edge $\langle s, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\beta}\rangle$, Boomerang ensures that the object that is accessible through the access graph $\widetilde{\alpha}$ at statement s is also accessible through $\widetilde{\beta}$ at statement t. By design in IFDS, the path edge is restricted to the same method (i.e., the statements s and t belong to the same method). To track inter-procedural information, Boomerang defines three types of path edges depending on the access graph $\widetilde{\alpha}$ and the statement s:

- **Direct** ($s = d \leftarrow new$)
  The statement s is an allocation site of the access graph $\widetilde{\alpha}$. Therefore, $\widetilde{\alpha} = d$. If the forward solver generates such an edge, then the allocated object flows to the access graph $\widetilde{\beta}$ at statement t, where both s and t are in the same method.
- **Transitive** ($s = r \leftarrow c.m(p)$)
  The statement s is a call site. Therefore, the access graph referred to by $\widetilde{\alpha}$ is allocated in a method that is transitively reachable through this call. Upon exiting the callee, the forward path edge propagated in the caller has the call site s as its source statement. Boomerang can therefore distinguish between multiple objects instantiated at the same statement, but reachable via two distinct calls in the same method.
- **Parameter** ($s \in startPoint$)
  The statement s is the first statement of the method containing t. The base variable of the access graph $\widetilde{\alpha}$ represents a formal parameter of the method. This means that the allocation site of the access graph $\widetilde{\alpha}$ is not found in the current method nor any of its transitively reachable callees; the access graph has been allocated in a context that is not yet known to the analysis. This type of edge corresponds to the path edge in IFDS used to generate intra-procedural summaries [12].

Once the outer fixed-point algorithm of Boomerang stabilizes, the result $Res(t, \widetilde{\gamma})$ of the query $(t, \widetilde{\gamma})$ is gathered from all generated forward path edges. It is a map with domain $\{\langle s, \widetilde{\alpha}\rangle \mid \exists \langle s, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\gamma}\rangle\} \subseteq \mathcal{S} \times \mathcal{APG}$. To each element $\langle s, \widetilde{\alpha}\rangle$ of the domain, Boomerang associates the set $\{\widetilde{\beta} \mid \exists \langle s, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\beta}\rangle\} \subseteq \mathcal{APG}$. The statements of the elements of the domain hold the allocation sites, whereas the accompanying sets contain all access graphs accessible at statement t which point to the related allocation site. Hence, any access graph $\widetilde{\alpha}$ in one of the sets may-aliases with $\widetilde{\gamma}$ at statement t. We note that by $\widetilde{\alpha} \blacktriangleleft Res(t, \widetilde{\gamma})$.

The domain of the result can contain origins of parameter path edges. This means that some allocation sites were undetected, as they are contained in the callers. Further queries can be issued to find them. Therefore, the results are independent of any calling-context. This design decision enables BOOMERANG to answer a query under particular client-provided calling-context. We address this feature in Section 4.2.

4 Implementation Challenges

In this section, we discuss how BOOMERANG addresses the following challenges:

- How are the $\mathcal{POI}$s handled?
- How can the analysis answer queries under specific client-provided context?

4.1 Points of Indirection

As explained in Section 3.2, $\mathcal{POI}$s are resolved in the outer fixed-point iteration. Each $\mathcal{POI}$ is treated independently.

4.1.1 Allocation Site [Alloc]

Upon discovering an allocation site during the backward pass, BOOMERANG needs to determine which access graphs might point to it. From an allocation site $a \leftarrow new$, BOOMERANG creates an access graph $\widetilde{\alpha}$ with base a and an empty field graph. Then, the path edge $\langle a \leftarrow new, \widetilde{\alpha}\rangle \xrightarrow{f} \langle a \leftarrow new, \widetilde{\alpha}\rangle$ is added to the work list of the forward solver. To avoid unnecessary computations (e.g., in a branch of non-interest to the query), the forward propagations are directed along the path(s) taken by the backward analysis which discovered the $\mathcal{POI}$. This is done, by following the backward path edges. If an additional backward path is discovered later on, the appropriate forward part will then be computed.

4.1.2 Field Write Statements [Write]

Let $\langle r, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\beta}\rangle$ be the path edge currently processed by the forward solver when the [**Write**] is discovered. The statement t is a (forward) successor statement of a field write statement $(x.f \leftarrow y)$ where the base of the access graph $\widetilde{\beta}$ is x and its head is f. BOOMERANG searches for aliases of x, the base of the overwritten field. For all aliases $\widetilde{\delta} \blacktriangleleft Res(t, x)$, we construct $\widetilde{\gamma} = \widetilde{\delta} \uplus \widetilde{\beta}$. For each $\widetilde{\gamma}$, BOOMERANG adds the path edge $\langle r, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\gamma}\rangle$ to the IFDS work list of the forward solver. For example, in Figure 5, the path edge $\langle \mathbf{L3}, \texttt{c}\rangle \xrightarrow{f} \langle \mathbf{L4}, \texttt{a.f}\rangle$ (marked with $\odot$) is added to the forward solver because before **L4**, the base variable `b` and `a` are aliases.

4.1.3 Alias on Forward Return-Flow [Return]

The [**Return**] indirection occurs upon a change of scope. This change occurs when a path edge reaches a method's exit statement and the analysis propagation is continued in its caller or when an existing summary is applied at a call site [17]. Figure 8 illustrates the former case.

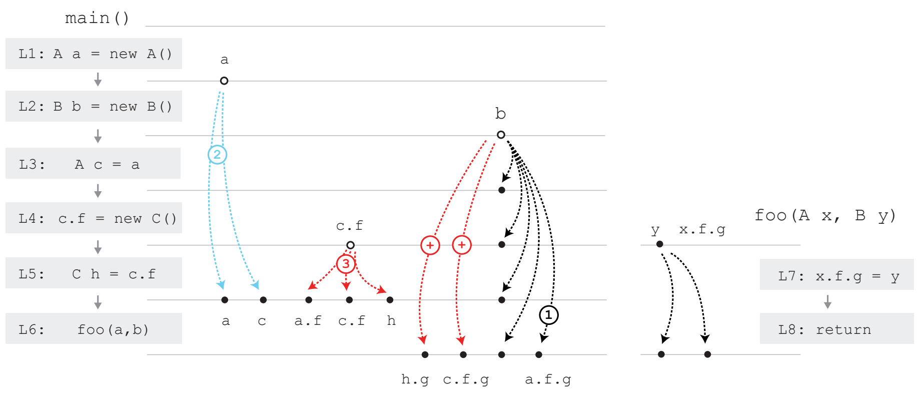


Figure 8 Handling alias on forward return-flow [**Return**].

Let us assume that we are interested in the query $(\mathbf{L6}, \texttt{b})$. The forward propagation of the allocation site at **L2** constructs the edge $\langle \mathbf{L2}, \texttt{b}\rangle \xrightarrow{f} \langle \mathbf{L6}, \texttt{a.f.g}\rangle$ (labeled with ①). This edge in the `main` method is a result of the propagations in `foo`. At this point, Boomerang needs to be aware of new aliases in the new scope. This requires a recursive query for all prefixes of `a.f.g`. First, the query $(\mathbf{L5}, \texttt{a})$ is issued, delivering the alias $c \blacktriangleleft Res(\mathbf{L5}, \texttt{a})$ (leftmost path edges marked with ②). The field `f` is then appended to all access graphs of the result set. This triggers the query $(\mathbf{L5}, \texttt{c.f})$, resulting in the path edges ③, that eventually finds the alias `h`. Finally, Boomerang appends the field `g` to the results and propagates the edges $\langle \mathbf{L2}, \texttt{b}\rangle \xrightarrow{f} \langle \mathbf{L6}, \texttt{c.f.g}\rangle$ and $\langle \mathbf{L2}, \texttt{b}\rangle \xrightarrow{f} \langle \mathbf{L6}, \texttt{h.g}\rangle$ forward. In Figure 8, those are the edges labeled with ⊕.

Formally, Boomerang iterates as follows. Let us assume the path edge $\langle u, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\beta}\rangle$, where $\widetilde{\beta} = [a, k, E, l]$, and t, one successor of the call site s, is propagated in the caller. For each node j of the access graph $\widetilde{\beta}$, the set F_j is computed:

$$F_j := \begin{cases} \{\widetilde{\gamma} \oplus k \mid \widetilde{\gamma} \blacktriangleleft Res(s, a)\} & \text{if } j = k, \\ \{\widetilde{\gamma} \oplus j \mid \widetilde{\gamma} \blacktriangleleft Res(s, \widetilde{\delta}),\ \widetilde{\delta} \in F_i, (i, j) \in E\}. & \end{cases}$$

By construction, the set F_j depends on the set F_i where i is a predecessor node of j in the access graph. The first set to be constructed is the set F_k, as it does not depend on any other sets. All other sets are computed successively[1]. For the tail node l of $\widetilde{\beta}$, the set F_l eventually represents all other aliases of $\widetilde{\beta}$ that hold just before the call site. For all of those aliases, the path edges $\langle u, \widetilde{\alpha}\rangle \xrightarrow{f} \langle t, \widetilde{\delta}\rangle$ with $\widetilde{\delta} \in F_l$ are propagated forward to connect the indirect aliases occurring in the new scope. In the example of Figure 8, the constructed sets are $F_{\texttt{f}} = \{\texttt{a.f}, \texttt{c.f}\}$ and $F_{\texttt{g}} = \{\texttt{a.f.g}, \texttt{c.f.g}, \texttt{h.g}\}$.

4.1.4 Field Read Statements [Read]

Handling field read statements is the inverse of forward handling field write statements. At a [**Read**] point, we can assume a path edge $\langle r, \widetilde{\alpha}\rangle \xrightarrow{b} \langle t, \widetilde{\beta}\rangle$ where t is a (backward) successor of a field read statement $(x \leftarrow y.f)$. The access graph $\widetilde{\beta}$ then has y as its base variable, and the field f as its head. Due to the field read, the allocations to any alias of $y.f$ become allocation

[1] The computation might lead to a fixed-point iteration over the sets F_j.

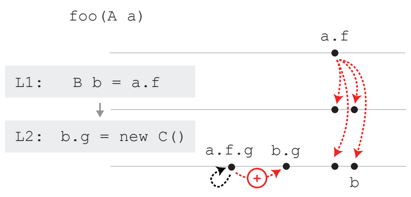


Figure 9 Example for [**Call**]

sites of interest for the backward analysis. To detect these allocation sites, Boomerang first searches aliases of y. The original access graph $\widetilde{\beta}$ is then concatenated to each alias $\widetilde{\delta} \blacktriangleleft \mathit{Res}(t,y)$ to form $\widetilde{\gamma} = \widetilde{\delta} \uplus \widetilde{\beta}$. The newly formed access graphs $\widetilde{\gamma}$ are propagated backward by adding the path edges $\langle r,\widetilde{\alpha}\rangle \overset{b}{\rightarrow} \langle t,\widetilde{\gamma}\rangle$ to the backward solver.

4.1.5 Alias on Backward Call-Flow [Call]

This $\mathcal{POI}$ is discovered in the call-flow functions of the backward analysis. Similar to [**Return**], [**Call**] leads to a recursive query of the prefixes of the access graph entering the method. Figure 9 shows an example where the backward analysis searches for the allocation site of `a.f.g` within `foo`. This access graph is allocated through an alias within `foo`. The local alias `b` of `a.f` has to be found first by executing the query (**L2**, `a.f`). This query results in the red dotted path edges shown on the right. From these path edges, Boomerang derives that it also needs to find allocations of `b.g` as well as `a.f.g`. Hence, the backward path edge labeled with $\odot$ is added to the backward solver. This edge leads to the detection of the allocation ([**Alloc**]) of the alias at **L2**.

In general, at a [**Call**], an access graph $\widetilde{\alpha} = [a,k,E,l]$ enters a method via the call-flow function. The last field of this access graph is then removed using the remove tail operation, resulting in the set $[a,k,E,l]\ominus$. For each element $\widetilde{\beta}$ of that set, all aliases are computed. The last step is to re-append the field l using the append tail operation ($\oplus$) and continue the backward propagation with the obtained aliases.

4.2 Client-Driven Context-Resolution

Up to this section, a query issued within method `m` was *context free*, causing it to propagate into `m` and its callees, but not into its callers. This design enables the creation of a client-driven context-resolution system where, after the context-free query is resolved, the client can define which particular calling contexts are relevant for Boomerang to further explore. This excludes contexts that are uninteresting to the client, helping increase precision and improve performance. If the client always chooses all available contexts, Boomerang computes pointer information over all contexts, like a standard pointer analysis would.

To use the client-driven context-resolution system, the client defines a *calling context*, which is a statement (optionally enriched by client-specific information) and a *calling-context function*, a function $\mathcal{C} \rightarrow 2^{\mathcal{C}}$, where $\mathcal{C}$ is the set of calling contexts. With the help of this function, Boomerang internally constructs a *context graph* $\mathcal{G}$, whose nodes are calling contexts. Initially, $\mathcal{G}$ is composed of one node, the *initial calling context*, given by the statement at which the query is issued. Boomerang then extends the graph successively. Except for the initial calling context, the other calling contexts' statements are call sites.

A context-free query is associated with each calling context of the context graph. Until a fixed point is reached, Boomerang computes the given queries for the calling contexts.

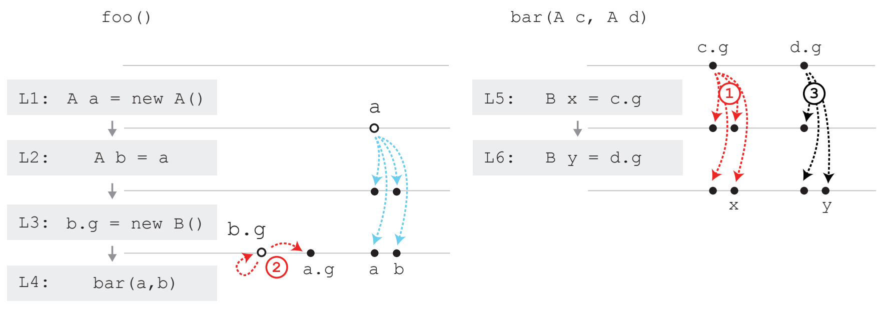


Figure 10 Client-Driven Context-Resolution.

This is done by using a worklist where each workslist item is a pair of a calling context and a query, $[C, (s, \widetilde{\alpha})]$.

Processing an item computes $Res(s, \widetilde{\alpha})$, the given context-free query. The result is then associated to the elements' calling context C. A check is performed to determine if the result set could potentially be missing allocation sites or aliases due to callers. This happens if the result contains an access graph $\widetilde{\beta}$ with a formal argument of the method as its base. If so, Boomerang evaluates the calling context function by supplying it the current calling context C to obtain more contexts. Each access graph $\widetilde{\beta}$ is then mapped to each new calling context D, and the element $[D, (t, \widetilde{\gamma})]$ is added to the worklist. Hereby, t is the (call site) statement of the calling context D and $\widetilde{\gamma}$ is the access graph $\widetilde{\beta}$ with replaced formals by actual arguments of the call site. In addition to extending the context graph from callee to callers, some results at the nodes must be *pushed back* along the caller to callee relationships within the graph, meaning that appropriate worklist elements are added. We clarify this in the following example. Finally, once the fixed point is reached, Boomerang extracts the complete pointer information from the context graph, by traversing the graph from each context-node containing an allocation site, to the original client context.

Figure 10 illustrates the context-resolution algorithm. The client analysis wishes to solve the query $(\mathbf{L6}, \mathtt{x})$ in method `bar` under *all calling contexts*. First, the context graph only contains the node **L6** [2] and the worklist is initialized with the element $[\mathbf{L6}, (\mathbf{L6}, \mathtt{x})]$. The result of the context-less query $(\mathbf{L6}, \mathtt{x})$ is $\langle \mathbf{SP}_{\mathtt{bar}}, \mathtt{c.g}\rangle \mapsto \{\mathtt{x}, \mathtt{c.g}\}$, which is encoded by the path edges ending at **L6** and labeled with ①. As the path edges are of parameter type, the alias information holds if and only if there exists an allocation site for `c.g` in one of `bar`'s callers. Boomerang then queries the client for the potential calling contexts of the calling context **L6**, and receives the set $\{\mathbf{L4}\}$. Therefore the calling context **L4** is created and linked to **L6** in the context graph, and $[\mathbf{L4}, (\mathbf{L4}, \mathtt{a.g})]$ is added to the worklist. For this query, Boomerang delivers $\langle \mathbf{L3}, \mathtt{b.g}\rangle \mapsto \{\mathtt{a.g}, \mathtt{b.g}\}$, —the path edges marked with label ② [3].

The results computed at the calling context **L4** in `foo` are pushed back to the callee `bar`, as the context graph contains an edge from **L6** to **L4**. This maps the access graphs `a.g` to `c.g` and `b.g` to `d.g`. At this point, Boomerang knows that `c.g`, `x`, and `d.g` are allocated at **L3**. However, one alias, `y`, is still missing. Therefore, an additional element is added to the worklist: $[\mathbf{L6}, (\mathbf{L6}, \mathtt{d.g})]$. It creates the path edges labeled with ③. Finally, the path

[2] We assume no client-specific information is bound to a context.
[3] The query internally creates the dotted unlabeled path edges within a sub-query to find the aliases `a` and `b`.

edges ①, ②, and ③ prove the reachability between the allocation site at **L3** and the access graph in $\{\texttt{c.g}, \texttt{x}, \texttt{d.g}, \texttt{y}\}$, which is the result of our initial query under all calling contexts.

In the example from Figure 10, a calling context is solely defined by a statement, and based on that the results are merged. Enriching the calling contexts with additional client-specific information enables the client to control the context-sensitivity of the results. For example, the client can add a call-stack of length k to make the analysis k-context-sensitive. As an alternative, the context can be modeled as a *value context*, holding more information about the (client's) data-flow value under which the method has been reached. We describe such value contexts in our experiments.

By filtering calling contexts, the client analysis can limit Boomerang's search space (e.g., by analyzing only `foo`, in Figure 10, and not all of the other callers of `bar`). This leads to significant savings in computational resources for Boomerang, as we show in **RQ3** in our experiments. The obtained results can, of course, only be considered sound with respect to the calling contexts provided by the client.

5 Evaluation

We implemented Boomerang on top of the Soot analysis framework and extended the existing IFDS solver Heros [4] to fit our needs (e.g., allowing path edges to start from a custom statement). The implementation was rigorously tested with more than 100 test cases. Boomerang is currently single-threaded, but a multi-threaded version is in the works.

5.1 Research Questions

Our empirical evaluation aims to answer the following research questions:

RQ1. How precise is Boomerang compared to other pointer analyses?
RQ2. How does the use of Boomerang affect the performance of a client analysis?
RQ3. How much influence does the client-driven context-resolution have on the search space for Boomerang?

5.2 Demand-driven Pointer Analyses

In **RQ1** and **RQ2**, we compare Boomerang to two flow-insensitive demand-driven pointer analyses: the refinement-based points-to analysis by Sridharan and Bodík [23] and the alias analysis by Yan et al. [30] (hereafter denoted SB and DA, respectively).

A query (m, v) to SB comprises a local variable v and the method m it belongs to. The query result is a points-to set of the given variable with each allocation site being enriched by a calling context. A query for DA consists of two local variables and the methods they belong to: (m_1, v_1, m_2, v_2). The result is a boolean value stating whether the two variables v_1 and v_2 in the given methods may alias or not. Although this interface is convenient for some client analyses (e.g., datarace analysis), points-to sets cannot be derived from alias information.

5.3 Results

RQ1. How precise is Boomerang compared to other pointer analyses?

Due to the difference in analysis types, returned information and query format, we compare the precision of SB, DA, and Boomerang on the common basis of alias information. We use DA's query format as this information can be derived by all three analyses. For SB, an alias query is mapped to two points-to queries, one for each variable of the alias query. If the intersection of the two points-to sets is non-empty, the two variables may alias. Boomerang, with its rich results, only needs to issue one query for one of the two variables. If the second variable is in the result set, both may alias. To evaluate the precision of Boomerang's points-to information, we additionally compare its results to SB's points-to sets.

The PointerBench Micro-Benchmark. Evaluating the precision of a pointer analysis requires a ground truth to compare against. To the best of our knowledge, there exists no benchmarks for pointer analyses. In this paper, we introduce PointerBench[4], a micro-benchmark for pointer analyses that contains 36 specially crafted small programs that depict common pointer analysis issues for Java (e.g., handling collections, field-sensitivity, access paths). Each program is provided with may (and, whenever possible, must) information about all aliases, non-aliases, and allocation sites of a particular variable. Using PointerBench, alias, points-to, or any kind of pointer analysis can be compared against one another based on the same ground truth.

Alias Pairs. In Table 1, we report the true positives, false positives, and false negatives for each pair of aliasing/non-aliasing variables. Across all the programs in PointerBench, Boomerang achieves 100% recall and 90% precision, DA achieves 95% recall and 82% precision, and SB achieves 97% recall and 87% precision with respect to alias pairs.

The main reason for the loss of recall for DA and SB is the test case `AccessPath`. A limitation of the two analyses is that they only support queries on local variables. Therefore, it is not possible to check if two access graphs `a.f.g` and `b.h.i` alias. Additionally, DA reports a false negative for the test case `ObjectSensitivity`. This test case creates an object and passes it to an static method that returns its parameter (i.e., it is an identity function). In the caller of that function, the argument to the call and the return value should alias. DA incorrectly models static methods, which leads to this false negative.

In the group `Collections`, the test cases contain operations on `HashSet` and `HashMap` (e.g., inserting and retrieving elements). These collections store elements in arrays, but all three analyses are array-insensitive. Therefore, arrays are treated as a single assignment to the heap. All three analyses encounter a precision loss here.

For the test case `FlowSensitivty`, DA and SB report a false positive since they are both flow-insensitive. An additional false positive for DA is reported on the test case `Null`. Given the statements `a = null; b = a;`, DA reports `a` and `b` as an alias pair, even though the points-to sets of `a` and `b` are empty. We call this the *null-aliasing property.*

Allocation Sites. For each program in PointerBench, the last two columns of Table 1 show false negatives, false positives and true positives in terms of allocation sites reported by Boomerang and SB. The alias analysis DA has been excluded from this part of the evaluation as it does not produce any points-to information.

Across all test cases in PointerBench, Boomerang and SB achieve 100% and 98% recall, respectively. Similar to the aliasing case, SB cannot handle the `AccessPath` case. With respect to precision, Boomerang and SB achieve 86% and 69%, respectively. The main reason for SB's drop in precision are the test cases `Interprocedural` and `FieldSensitivity` due to SB being flow-insensitive.

[4] Available for download at `https://github.com/secure-software-engineering/PointerBench`

Table 1 The precision and recall of Boomerang, SB, and DA with respect to alias pairs and allocation sites on PointerBench. ⊖= false negatives, ⊕= false positives, ✓= true positives.

		Alias pairs			Allocation sites	
	Tests	Boomerang	DA	SB	Boomerang	SB
Basic	SimpleAlias	✓✓	✓✓	✓✓	✓	✓
	Loops	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓
	Interprocedural	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓	✓✓⊕⊕⊕⊕
	Parameter	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓	✓✓
	Recursion	✓	✓	✓	✓	✓
	ReturnValue	✓✓✓✓✓	✓✓✓✓✓⊕	✓✓✓✓✓	✓✓✓	✓✓✓⊕
	Branching	✓✓	✓✓	✓✓	✓✓	✓✓
General Java	AccessPath	✓✓	⊖⊖	⊖⊖	✓	⊖
	ContextSensitivity	6×✓	6×✓	6×✓	6×✓	6×✓
	FieldSensitivity	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓	✓✓⊕⊕⊕⊕
	FlowSensitivity	✓	✓⊕	✓⊕	✓	✓⊕
	ObjectSensitivity	✓✓✓✓	✓✓✓⊕⊕⊖	✓✓✓✓	✓✓	✓✓⊕
	StrongUpdate	✓✓✓⊕	✓✓✓⊕	✓✓✓⊕	✓✓⊕⊕	✓✓⊕⊕
Corner Cases	Exception	✓✓✓	✓✓✓⊕	✓✓✓⊕	✓✓	✓✓⊕
	Interface	✓✓	✓✓	✓✓	✓	✓
	Null		⊕			
	OuterClass	✓✓⊕	✓✓⊕	✓✓⊕	✓⊕	✓⊕
	StaticVariables	✓✓	✓✓	✓✓	✓	✓
	SuperClasses	✓✓⊕	✓✓⊕	✓✓⊕	✓⊕	✓⊕
Collections	Array	✓✓⊕	✓✓⊕	✓✓⊕	✓⊕	✓⊕
	List	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓	✓✓
	Map	✓✓⊕	✓✓⊕	✓✓⊕	✓⊕	✓⊕
	Set	✓⊕⊕	✓⊕⊕	✓⊕⊕	✓⊕	✓⊕
	Precision	0.90	0.82	0.87	0.86	0.69
	Recall	1.0	0.95	0.97	1.0	0.98

RQ2. How does the use of Boomerang affect the performance of a client analysis?

The usefulness of a support analysis is best evaluated in combination with a concrete client analysis. We evaluate the performance of FlowDroid, a context- and flow-sensitive taint analysis for Android [1]. We see FlowDroid as a representative of taint and other analyses of similar precision (e.g., typestate analysis). In this experiment, we compare the performance of FlowDroid on real-world applications from the Google Play Store when using Boomerang, SB and DA.

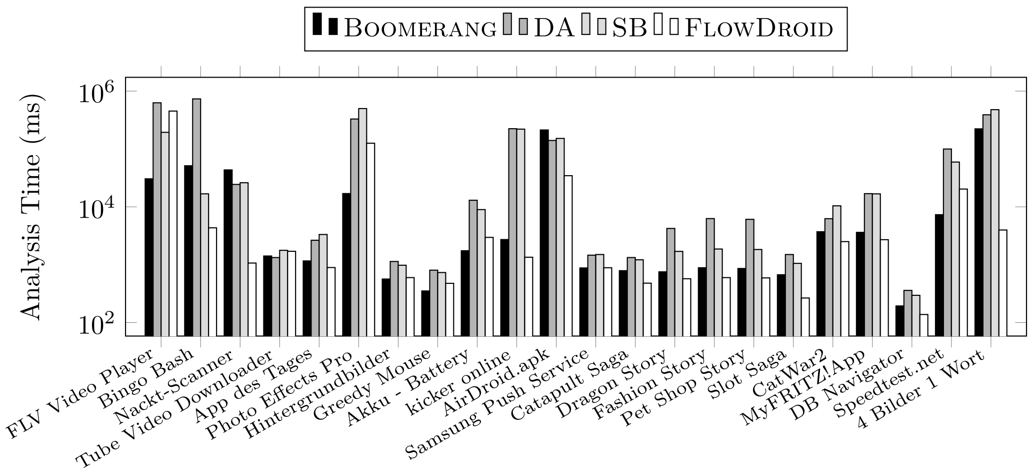
Like many taint or typestate analyses, FlowDroid requires alias information at field write statements and on return from callees to call sites. These are points of indirection where FlowDroid might indirectly taint other data-flow facts (FlowDroid access paths). At those statements, FlowDroid needs to obtain all aliases of the tainted access path. FlowDroid then taints those aliases and continues the taint propagation.

DA and SB's integration into FlowDroid need additional post-processing for the computation of all-alias sets, as there interface does not deliver the right information. To construct such a set A for a given FlowDroid access path `a.f.g`, all assign statements of the method containing the field-write statement or the call site are considered. For an assign statement, `c = d`, the analysis must check if the variable `d` aliases with `a`. In such case, the access path `c.f.g` is added to A. Similarly, for a field read statement `c = d.f`, where the field matches the first field of the access path, the analysis enquires whether `d` and `a` alias. If so, the access path `c.g` is also added to A. A field-write statement is handled likewise, but the appropriate field is prepended to the access path.

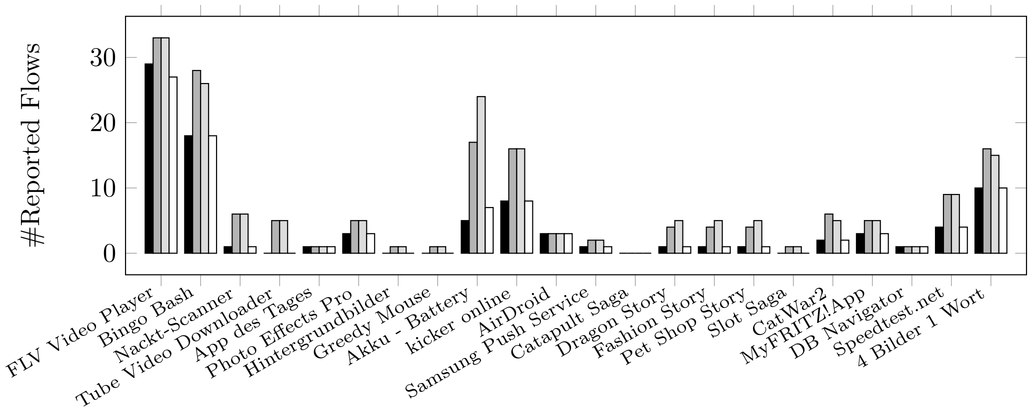
In contrast, when using Boomerang, FlowDroid requires only one query per field write or call site. Boomerang directly returns an alias set, whose access graphs only need to be translated to FlowDroid access paths. For the client-driven context-resolution system (Section 4.2) and the integration into FlowDroid, each calling context is enriched by one FlowDroid path edge. For the initial calling context, the edge that triggered the query is used. The calling-context function uses the *incoming map* [17] of FlowDroid's IFDS solver to compute the calling contexts. The incoming map describes how path edges in the caller are mapped to the path edges in the callee and guarantees the context-sensitivity for IFDS. In Boomerang, for a calling context enriched with a path edge p, for each relationship $q \to p$ of the incoming map, the calling-context function returns a calling context enriched by the path edge q at the appropriate call site. Boomerang then backtracks the path(s) FlowDroid's data-flow fact took and computes only the alias information necessary for the client. In the case of FlowDroid, calling contexts are *value contexts* since the calling context holds additional information about the data-flow value that reached the statement.

We set a timeout of one second per query for all three integrated pointer analyses. We also limit the overall analysis time for each application to 15 minutes. We ran FlowDroid with the three alias strategies, and, as a baseline, with FlowDroid's own intertwined alias strategy [1] on 100 randomly chosen apps from the top 500 apps of the Google PlayStore. The experiment was conducted using JDK version 1.7.0_85 on a 64 core (Intel Xeon E5-4640) with 32GB heap space.

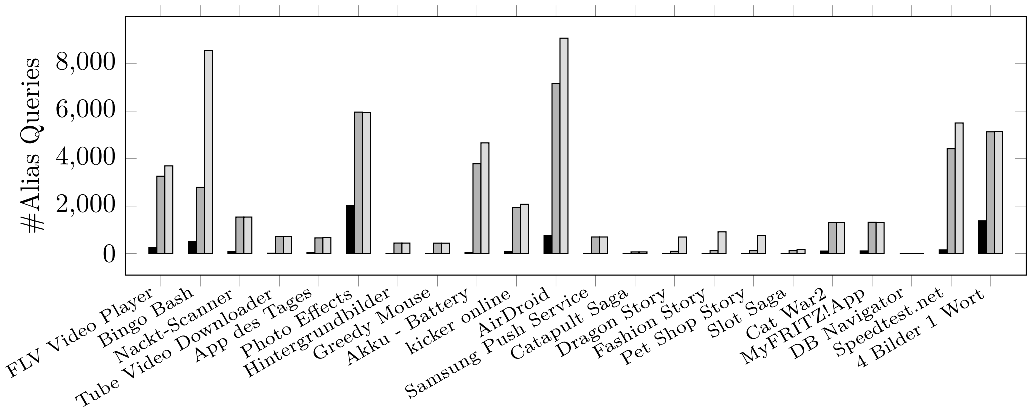
FlowDroid supported by Boomerang successfully analyzed 53 applications within the overall time limit, while the support of DA, SB, and the original alias strategy of FlowDroid could only analyze 23, 30, and 43 applications, respectively. Figure 11a shows the analysis time of the 22 applications for which all four strategies did not time out. As the analysis times vary a lot, the diagram uses a logarithmic y-axis. On average, using Boomerang, FlowDroid is 2.6x faster than using DA and 3.9x faster than using SB. When comparing to FlowDroid, the original alias strategy performs 1.6x times better than



(a)



(b)



(c)

Figure 11 Comparison of the three alias strategies with respect to a) analysis time, b) reported flows, and c) triggered alias queries.

Boomerang. The reason is that FlowDroid's alias strategy also follows the incoming map [1].

Figure 11b shows that the choice of the alias strategy affects the number of taint flows reported by FlowDroid. In total, FlowDroid reports 92 flows with both Boomerang and the original FlowDroid aliasing strategy, whereas it reports 168 flows with DA and 174 flows with SB. When running the analyses on the FlowDroid test cases, we noticed that SB and DA reported many spurious flows. The main cause for those false positives is that both analyses are flow-insensitive and do not consider calling contexts. On average, the use of DA and SB cause FlowDroid to report 1.8x more flows when compared to Boomerang or the original FlowDroid aliasing strategy.

Figure 11c presents the total number of alias queries issued by FlowDroid for the 22 fully analyzed applications. The integration of FlowDroid with Boomerang issues an average of 19.9x fewer queries when compared to DA and 29.4x fewer queries when compared to SB. This shows that Boomerang issues significantly fewer queries, due to the richer pointer information Boomerang provides.

In summary, (1) using DA and SB, the FlowDroid analysis is slower, (2) more alias queries are triggered and (3) more taint flows are reported when compared to the integration with Boomerang.

RQ3. How much influence does the client-driven context-resolution have on the search space for Boomerang?

As described in **RQ2**, for the integration into FlowDroid, Boomerang uses the client-driven context-resolution system and follows the incoming map of FlowDroid's IFDS solver to find appropriate calling context at which Boomerang continues. Incorporating the context-resolution system, can be seen as a filtering of the call graph's edges before Boomerang continues with its computation in the callers. Hence, the filtering will reduce the search space for Boomerang in advance. For DA's and SB's integrations, all calling contexts are directly considered, even though they might be of non-interest to the client.

To evaluate the effectiveness of the filtering, we measured and report the following parameters for the 53 applications that Boomerang could analyze within the time limit:

- *Context requests*: the number of times Boomerang requests calling context information from FlowDroid,
- *All contexts*: the number of call sites in the call graph (hence, the number of calling contexts in which the analysis could continue) and
- *Client contexts*: the filtered number of contexts that FlowDroid returns to Boomerang.

In the worst case, FlowDroid provides all possible contexts. In each of the returned calling context, new queries to Boomerang are triggered (see Section 4.2). Boomerang then has to evaluate the queries for all of these calling contexts. This worst-case scenario is similar to the behavior of DA and SB; both propagate along all possible calling contexts.

Figure 12 shows the percentage of filtered calling contexts as a function of the number of context requests made for each application. The high ratio of filtering significantly reduces the search space for Boomerang in advance. We also observed that FlowDroid tends to filter out less calling contexts for applications where Boomerang requests fewer contexts. In summary, the consideration of the calling context of interest by the client early in the computation phase significantly reduces the search space of Boomerang.

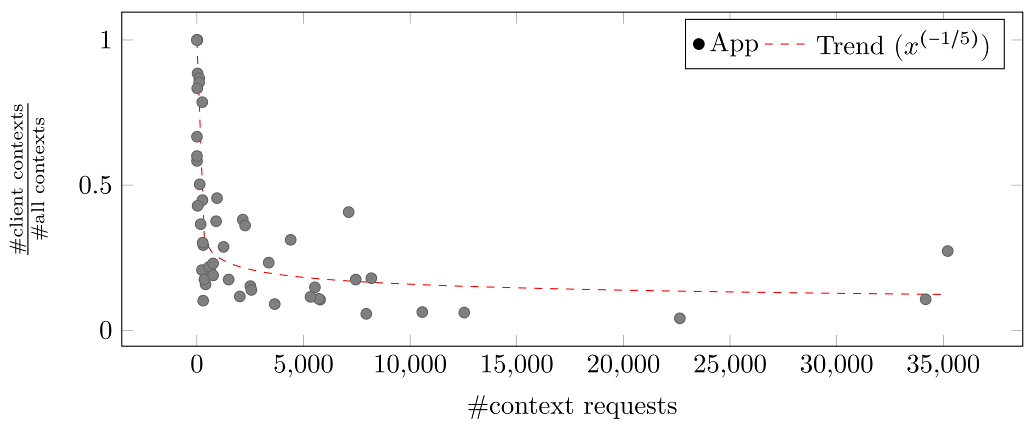


Figure 12 The relation between the number of context requests to the effectiveness of the pre-filter for BOOMERANG.

5.4 Discussion

BOOMERANG is particularly designed for demand-driven clients, such as FLOWDROID, seeking high precision and requiring all-alias sets. BOOMERANG is well-suited for object-tracking analyses, such as taint or typestate analyses, as it makes those clients easier to implement, and returns results faster, and with a higher precision than existing approaches.

If client analyses require simpler points-to or alias information, they might not benefit from using BOOMERANG. For example, for a race-checking client analysis, the alias analysis DA seems to be a more reasonable choice than BOOMERANG because for such a client a boolean answer to the question whether two given variables alias is usually sufficient. In future work, we plan to test the applicability of BOOMERANG as a general-purpose pointer analysis for other types of client analyses and also invite others to do so. As others have argued before [11], the choice of the optimal pointer analysis heavily depends on the client. But it is exactly for this reason that we advocate to evaluate future pointer analyses also with respect to how clients would need to query them and how that impacts the performance of those clients.

6 Related Work

Pointer analyses provide information about which pointers point-to which memory locations in a given program. Such information is necessary for many static client analyses such as escape analysis [20], shape analysis [8], and call graph construction [9, 26, 27]. This demand causes significant effort towards developing various pointer-analysis techniques. Ryder [21] provides a comprehensive taxonomy of many dimensions that affect the precision and cost of a given pointer analysis. Sridharan et al. [24] present a survey of the state-of-the-art alias analyses for object-oriented programs. For the purpose of this work, we categorize previous work on pointer analysis into two broad themes: *whole-program pointer analyses* and *demand-driven pointer analyses*.

6.1 Whole-Program Pointer Analyses

Lhoták et al. [14] developed SPARK, a flexible framework for points-to analysis of Java programs. SPARK provides a SOOT [29] transformation that constructs the call graph of the input program on-the-fly while calculating the points-to sets. SPARK's analysis is context- and flow-insensitive. Paddle [2] is a drop-in replacement for SPARK that allows for a context-sensitive but flow-insensitive whole-program points-to analysis using binary-decision diagrams.

Bravenboer et al. [5] present DOOP, a Datalog-based points-to analysis framework for Java programs. DOOP can also construct the call graph on-the-fly while computing the points-to sets. DOOP offers various context-sensitivity configurations, such as call-site-sensitive, object-sensitive, and heap-abstraction-sensitive. DOOP can support limited, method-local flow-sensitivity by performing the pointer analysis on static single assignment (SSA) form. Some support for flow-sensitivity is in preparation.

De et al. [6] propose a whole-program flow- and context-sensitive analysis that uses access paths. They use a similarly rich representation as BOOMERANG, allowing for strong updates. The approach is based on the call-strings approach and was evaluated with a call-string length of 1. This approach quickly becomes imprecise in contrast to the functional approach we use in BOOMERANG.

6.2 Demand-Driven Pointer Analyses

Heintze et al. [10] introduce a demand-driven approach for pointer analysis that performs just enough computation to determine the points-to sets for a given list of pointer variables. However, the information obtained is context- and flow-insensitive.

Sridharan et al. [25] show that points-to information can be encoded as a context-free language (CFL). The problem of on-demand pointer analysis can then be re-defined as a CFL-reachability problem that can be solved on a graph representation. Labeling the edges of the graph enables field-sensitivity. As opposed to BOOMERANG, this formulation is flow-insensitive.

Based on this CFL-reachability, Shang et al. [22] present DYNSUM, a demand-driven context-sensitive but flow-insensitive points-to analysis. DYNSUM uses method summaries to improve performance for multiple queries within the same method. The answer to a query to DYNSUM is a points-to set. As we have previously discussed, a result to BOOMERANG encodes alias information, as well as points-to sets.

Yan et al. [30] propose a novel on-demand alias analysis for Java that is formulated as a CFL-reachability problem. For methods with a high in-degree (a preset threshold) in the call graph, summaries for the alias information are computed. Because it is an alias analysis, no points-to sets are generated and also not all aliasing variables can be directly retrieved as it is possible with BOOMERANG.

7 Conclusion

We have presented BOOMERANG, a demand-driven pointer analysis that provides rich pointer information to the client, comprising all allocation sites the queried pointer can point to, as well as access graphs representing all pointers that, at the queried statement, may point-to that same allocation site.

The analysis uses two instantiations (backward/forward) of the IFDS framework. Its procedure summaries provide efficiency, while flow-sensitivity and unlimited context-sensitivity provide high precision. An additional outer fixed-point iteration within Boomerang yields field-sensitivity and soundiness [15]. The recursive evaluation of the backward and forward-analysis passes is carefully coordinated to construct only the minimal part of the exploded super graph necessary to answer a given query. Boomerang further allows a client to specify calling contexts to tailor the computational effort to specific parts of interest.

We have shown that Boomerang is more precise than the state-of-the-art demand-driven analyses on PointerBench. We have also shown that using Boomerang, clients compute their results issuing fewer points-to analysis queries, which significantly accelerates the overall analysis.

Acknowledgements. We would like to thank all anonymous reviewers and our shepherd Atanas Rountev to improve this work. This work was supported by the Fraunhofer Attract program and the DFG Collaborative Research Center CROSSING.

References

1 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, page 29, 2014.

2 Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI*, pages 103–114. ACM Press, 2003.

3 Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE*, pages 5–14, 2010.

4 Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *SOAP 2012*, pages 3–8, July 2012.

5 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

6 Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for Java with strong updates. In *ECOOP*, pages 665–687, 2012.

7 Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, pages 230–241, 1994.

8 Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, pages 1–15, 1996.

9 David Grove and Craig Chambers. A framework for call graph construction algorithms. *TOPLAS*, 23(6), 2001.

10 Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *PLDI*, pages 24–34, 2001.

11 Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.

12 Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. Demand interprocedural dataflow analysis. In *FSE*, pages 104–115, 1995.

13 Amey Karkare, Amitabha Sanyal, and Uday P. Khedker. Heap reference analysis for functional programs. *CoRR*, 2007.

14 Ondrej Lhoták and Laurie J. Hendren. Scaling Java points-to analysis using SPARK. In *CC*, pages 153–169, 2003.

15 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

16 Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, pages 347–366, 2008.

17 Nomair A. Naeem, Ondrej Lhoták, and Jonathan Rodriguez. Practical extensions to the IFDS algorithm. In *CC*, pages 124–144, 2010.

18 Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *POPL*, pages 327–338, 2007.

19 Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

20 Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL*, pages 285–293, 1988.

21 Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *CC*, pages 126–137, 2003.

22 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.

23 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

24 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, 2013.

25 Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76, 2005.

26 Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *OOPSLA*, pages 264–280, 2000.

27 Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, 2000.

28 Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, pages 210–225, 2013.

29 Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC*, pages 18–34, 2000.

30 Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, pages 155–165, 2011.