Easy and Effective Sound Points-To Analysis

YANNIS SMARAGDAKIS, University of Athens
GEORGE KASTRINIS, University of Athens

Virtually no realistic whole-program static analysis is sound, i.e., guaranteed to over-approximate all program behaviors. The main cause of such unsoundness is "opaque code" features such as reflection, dynamic loading, and native calls.

We present a defensive may-point-to analysis approach, which offers soundness even in the presence of arbitrary opaque code: all non-empty points-to sets computed are guaranteed to be over-approximations of the sets of values arising at run time. A key design tenet of the analysis is laziness: the analysis computes points-to relationships only for variables or objects that are guaranteed to never escape into opaque code. This means that the analysis misses some valid inferences, yet it also never wastes work to compute sets of values that are not "complete", i.e., that are missing elements due to opaque code. Laziness enables great efficiency, allowing us to perform a highly precise points-to analysis (such as a 5-call-site-sensitive, flow-sensitive analysis).

Despite its conservative nature, our analysis yields sound, actionable results for a large subset of the program code, achieving (under worst-case assumptions) 34-74% of the program coverage of an unsound state-of-the-art analysis for real-world programs.

ACM Reference format:
Yannis Smaragdakis and George Kastrinis. 2017. Easy and Effective Sound Points-To Analysis. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 21 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Static program analysis is an established research area, with a history tracing back to several decades and applications in virtually all modern programming technologies. Static analysis attempts to answer questions about program behavior under all possible inputs. Therefore, virtually all interesting static program analysis questions are undecidable—indeed the prototypical undecidable problem, the *halting problem*, is a static program analysis question: will a program terminate under all inputs?

Since static analysis attempts to solve undecidable problems, practical analysis algorithms are of one of two categories: either the analysis intends to under-approximate the precise answer, in which case it is called a *must-analysis*, or it intends to over-approximate the precise answer, in which case it is called a *may-analysis*.

The concept of *soundness* (often colloquially used as a synonym of "correctness") similarly depends on the flavor of analysis. A may-analysis is *sound* if it guarantees to over-approximate program behavior, i.e., to capture *all feasible behaviors*. A must-analysis is *sound* if it guarantees to under-approximate program behavior, i.e., to capture *only behaviors certain to be feasible*.

Given the prevalence of static analyses, and especially may-analyses, it is a rather surprising fact that practically *no implemented whole-program static may-analysis is sound*, i.e., correctly does what it intends to, for realistic programs (Livshits et al. 2015). For instance, no realistic may-point-to analysis computes a true over-approximation of the points-to sets that arise during program execution.

The main culprit for this apparent paradox is complex language features, such as native (non-analyzable) code, dynamic loading, reflection, as well as constructs such as the recent Java `invokedynamic`. In the context of this paper, we will collectively refer to these features as *opaque code*.

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

In this work, we propose *defensive analysis*: a static analysis architecture that aims for sound handling of opaque code. To achieve this goal, defensive analysis has a different logical form from past analyses. Instead of adding sophisticated handling of opaque code, defensive analysis redefines the analysis logic for regular, common language features, to defensively protect against the possibility that the analysis information potentially depends on opaque code. In essence, a defensive analysis produces inferences only when these are guaranteed to hold because of existing code and language features, and cannot possibly be violated by other, unknown code. Although this seems like a straightforward mode of operation, applying it to yield a realistic static analysis with useful coverage is a challenge.

We designed and implemented a defensive may-point-to (henceforth just "points-to") analysis for Java. The analysis guarantees that any points-to set it computes is over-approximate, i.e., captures all possible dynamic behaviors.

One of the hallmark features of our defensive points-to analysis is *laziness*: the analysis does not compute points-to sets unless it can also establish that they will contain a *bounded* number of objects. By implication, the analysis represents points-to sets with an unbounded number of objects (e.g., sets that depend on reflection or dynamic loading) as the *empty set*. An empty analysis result merely means "I don't know", which could signify that the points-to set is affected by opaque code, or simply that the analysis cannot establish that it is *not* affected by opaque code. Laziness yields high efficiency: the analysis can fall-back to an empty set (i.e., implicitly unbounded) without performing any computation or occupying space.

The defensive nature of the analysis combined with laziness result in a very simple specification. The analysis does not need to integrate over-approximate escape or alias reasoning (i.e., "can this object *ever* escape into opaque code?"), but only under-approximate escape reasoning (i.e., "here are simple, safe cases, when the object cannot possibly be affected by opaque code"). Failure to establish non-escaping merely means that the points-to set remains empty, to denote "I don't know" or "potentially unbounded".

In terms of applications, soundness is a requirement for ambitious clients, such as automatic optimization or semantics-preserving refactoring. Such clients are currently simply not feasible. In particular, inter-procedural automatic optimization has long been hindered by the absence of sound points-to analysis information. Thus, our work has significant potential for wide practical application.

Concretely, the work makes the following contributions:

- We offer the first realistic, general static may-point-to analysis that is sound in the presence of opaque code. (Arguably other approaches (Hirzel et al. 2004, 2007; Lattner et al. 2007) achieve soundness, but not for the full problem. Additionally, any analysis can be nominally sound by bailing out: rejecting wholesale any program that employs features that the analysis does not handle. This is also far from a solution. We compare more thoroughly in the related work section.)
- The analysis puts forward a very simple, yet effective, form. It starts from a sound state: all points-to sets are empty, denoting "potentially unbounded". It can then build from the ground up, i.e., to only reason about known safe cases. More sophisticated machinery (such as expensive escape or alias analysis) can grow the set of these safe cases, but is not a prerequisite for soundness. The analysis is also modular: it can be applied to any subset of the program, and will merely leave more points-to sets empty if other parts are unknown.
- The analysis is extremely efficient, leveraging its lazy representation of points-to sets. As a result, it can be made precise, beyond the limits of standard whole-program points-to analyses—e.g., for a 5-call-site-sensitive and flow-sensitive analysis.
- We show that the analysis, though quite defensive, yields useful coverage. In measurements over large Java benchmarks, our analysis computes guaranteed over-approximate points-to sets for 34-74% of the local variables of a conventional unsound analysis. (This number is much higher than that of a conventional sound but intra-procedural analysis.) Similar effectiveness is achieved for other metrics (e.g., number of calls de-virtualized), again with actionable, guaranteed-sound outcomes.

2 ANALYSIS ILLUSTRATION

We next describe the setting of defensive analysis and illustrate its principles and behavior.

2.1 Soundness and Design Decisions

Defensive analysis is a may-point-to analysis based on access paths, i.e., expressions of the form "*var*(.*fld*)*". That is, the analysis computes *the abstract objects* (i.e., allocation sites in the program text) *that an access path may point to*. The analysis is *flow-sensitive*, hence we will be computing points-to information per program point. Both of these design decisions are integral elements of the analysis, as we will justify in Section 2.2.

Soundness in this setting means that the analysis computes an over-approximation of any points-to set—i.e., the analysis computes (abstractions of) all objects that may occur in an actual execution. However, since not all allocation sites are statically known (due to dynamically loaded code), such an over-approximation cannot be explicit: not all possible values in a points-to set can be listed. Thus, there needs to be a special value, $\top$, to denote "unknown", i.e., that the analysis cannot bound the contents of a points-to set.

The above observation holds for any sound points-to analysis. Defensive analysis takes it one step further, by employing a *lazy* approach: it never populates a points-to set if it cannot guarantee that it is bounded. Thus, an empty points-to set for an access path signifies that (as far as the analysis knows) the access path can point to anything.[1] In other words, an empty set is equivalent to a $\top$ value.

With this representation choice, the analysis does not need to expend effort in order to be sound. All points-to sets (for any valid access path, of any length) start off empty, i.e., the analysis has implicitly assigned them $\top$ values, meaning "the set can contain anything". This is a sound answer, and is only subsequently refined. Defensive analysis conservatively keeps a set empty until it has enough information to compute finite contents for it.

The lazy evaluation and implicit $\top$ representation mean that defensive analysis does not need to employ sophisticated mechanisms to simply be sound. For instance, instead of a precise over-approximate escape analysis (that will avoid over-computing objects that escape into opaque code, so as to avoid invalidating all possible points-to sets by adding $\top$ values), defensive analysis can use a simple analysis (including none at all) to compute straightforward cases when an object is guaranteed to never escape into opaque code.

2.2 Illustrating Design Decisions

We can see the rationale behind our design decisions through simple examples.

The large potential for opaque code to affect analysis results has prevented past analyses from being sound. For instance, consider a simple heap load instruction:

```
x = y.fld;
```

Imagine that the analysis has (somehow) soundly computed all the objects that `y` may point to. It may also know all the places in the code where field `fld` is assigned and what is assigned to it. However, the analysis still cannot compute soundly the points-to set of `x` unless it also knows that all objects referenced by `y` can never escape to opaque code. This is hard to establish: not only do all sites of opaque code (reflection, unknown instructions, potential dynamic code generation sites, and more) need to be marked, but the analysis needs to know an over-approximation of which objects these sites can reach. This requires to have pre-computed an over-approximate (i.e., sound) points-to analysis, which is the problem we are trying to solve in the first place. Past work has dealt with this problem with unrealistic assumptions. For instance, Sreedhar et al. (2000) present a call-specialization analysis that can handle dynamic class loading, but only if given the results of a sound may-point-to analysis as input. Any solution along these lines will likely involve a complex interplay between points-to and escape reasoning.

[1] We use an explicit abstract value for `null`, therefore a points-to set that only contains `null` is not empty. This is standard in flow-sensitive analyses, anyway. (In flow-*in*sensitive analyses, `null` is typically a member of every points-to set, so it is profitable to not represent it, and hence have an empty set mean a `null`-only reference. No such benefit would arise in our flow-sensitive setting.)

Instead, defensive analysis pessimistically computes that a points-to set is ⊤ (i.e., can contain anything) unless it is certain that its contents are bounded. A guarantee of bounded contents typically comes from having tracked the contents of a variable or field all the way from its last assignment, and having established that no other code could have interfered. For instance, let us expand our earlier example:

```
1 y.fld = new A(); // abstract object a1
2 ... // analyzable, non-interfering code
3 x = y.fld;
```

The analysis can now know that the points-to set of `x` is {`a1`}, i.e., the singleton containing the allocation site for `A` objects on line 1. For this to be true, the analysis has to establish that all code between the store instruction to `y.fld` and the subsequent load does not interfere with the value of `y.fld`. For example, we can be certain of such non-interference if the code does not contain a store to field `fld` of *any* object, does not call any methods, and no other thread can change the heap at that segment of the program.These are simple, local conditions that the analysis can easily establish.

The above example serves well to illustrate the design choices of defensive analysis: it is a flow-sensitive analysis because it needs to track all points-to information that is guaranteed to hold, per-instruction, following closely all possible control-flow of the program. (As we shall see, a feature of the analysis is that this tracking is also done inter-procedurally, up to a maximum context depth, with appropriate renaming of variables.) It is also an analysis computing points-to information on *access paths* because this gives significantly more ability to reason about the heap locally. For instance, in the above program fragment, we may not know which objects `y` may point to.[2] However, we do know that `y.fld` certainly points to abstract object `a1` after line 1 of our example!

Finally, consider the design choice of representing unbounded points-to sets as empty, i.e., to lazily compute the contents of points-to sets only if they can be proven to be finite. This choice is responsible for much more than avoiding storage and initialization of ⊤ values. It enables the analysis to give a convenient meaning to any finite points-to sets that arise. Instead of "*this set currently has bounded contents, but may become* ⊤ *in the future*", a set of values implies "*this set has bounded contents and is guaranteed to always have bounded contents*". By making this distinction, the analysis never wastes effort computing points-to sets with explicit (non-⊤) contents only to later discover that the points-to set is ⊤. For an example of how much wasted effort can be saved by being lazy, consider an example program involving a heap load and a virtual call:

```
1 y.fld = new A();  // abstract object a1
2 while (...) {
3   x = y.fld;
4   x.foo(y);
5 }
```

An analysis may have computed all the abstract objects that `y.fld` may point to at line 3. One of these computed objects may induce a different resolution of the call instruction (line 4), which can suddenly lead to the discovery that a `y.fld`-aliased object can enter opaque code (while this was not true based on what the analysis had computed earlier). Since the object referenced by `y.fld` can change in code that is not analyzed, the points-to set of `x` at the load instruction will need to be augmented with the implicit over-approximation special value, ⊤. This means that all previously computed values for the points-to sets of `x` and `y.fld` are subsumed by the single ⊤ value. Computing these values and all others that depend on them constitutes wasted effort. To make matters worse, this is more likely to happen for *large* points-to sets, i.e., the more work the analysis has performed on computing an

[2] In fact, even if we did know, these would be abstract objects. Static analysis would almost never be able to establish soundly what their `fld` field points to, because this information needs to capture the `fld` values of all *concrete* objects (not just the latest one, but also those allocated long ago) represented by the same abstract object.

explicit points-to set, the larger (and less precise) the set will be, and the more likely it is that the work will be wasted because the set will revert to ⊤.

The design principle of "laziness in order to avoid wasted effort" is responsible for the scalability of defensive analysis. As we show in our experiments, defensive analysis scales to be flow-sensitive, 5-call-site sensitive over large Java benchmarks and the full JDK. (In past literature, even a 2-call-site-sensitive, flow-*insensitive* analysis has been infeasible over these benchmarks (Smaragdakis et al. 2014).)

2.3 Soundness Assumptions

The soundness claims of defensive analysis is predicated on assumptions about the environment. These assumptions reflect well the setting of safe languages, such as Java:

- **Object isolation.** Objects can only be accessed via high-level references. This means that objects are isolated: an object can be referenced outside the dynamic scope of a method or by a different thread only if a reference to the object has escaped the method or current thread. (This restriction also implies that objects are not contained in one another, though they can contain references to each other.)
- **Stack frame isolation.** Local variables are isolated from each other, thread-private and private to their allocating method. No external code can access the local variables of a method, even if the code is executed (i.e., is a callee) under the dynamic scope of the method.
- **Concurrency model.** In the simplified model of the paper, soundness is predicated on the assumption that standard mutexes (or operations on volatile variables) are used to protect all shared memory data. We later discuss how our implementation removes this assumption.

Thus, our setting is clearly that of a safe language with near-unlimited potential for dynamic behavior. Notably, we can have unknown instructions; calls to native code with arbitrary behavior (over a well-typed, isolated heap); generation and loading of unknown code (which may also be called, via dynamic dispatch, by unsuspecting *known* code); arbitrary access to existing or unknown objects (both field read/writes and method calls) via reflection, i.e., without such access being identifiable in the program text; and more.

3 DEFENSIVE ANALYSIS, INFORMALLY

The discussion of analysis principles in the previous sections gives the main tenets of defensive analysis. However, these need to be concretely applied over all complex language features affecting points-to information: control-flow merging, heap manipulation, and method calls. We give informal examples next. Following these examples should significantly facilitate understanding the formal specification of the analysis, in later sections.

Control-flow merging. Consider a branching example:

```
1 if (complexCondition()) {
2   x = new A();  // abstract object a1
3    // x points-to set is {a1}
4 } else {
5   x = notFullyAnalyzed();
6    // x points-to set is {}
7 } // x points-to set is {}
```

The first branch of the above `if` expression establishes that the points-to set of variable `x` is {`a1`}. For a conventional analysis, this is enough for adding `a1` to the points-to set of `x` at the merge point (after line 7). The defensive analysis, however, has to be conservative and not compute values that will later be overwritten with ⊤. Therefore, it will add `a1` to the final points-to set of `x` only if it can also prove that the points-to set of `x` in the second branch is bounded. If the analysis is not certain of this, the points-to sets of `x`, both in the second branch and at the merge point, stay empty, i.e., ⊤. Inability to bound the points to set of `x` in the second branch can be

due a variety of reasons: e.g., there can be opaque code inside `notFullyAnalyzed`, or the analysis may reach its maximum context depth, so that the return value of the method is not tracked precisely.

Heap manipulation. Similar treatment applies to all cases of points-to sets (e.g., for complex access paths) when information is merged: the analysis yields a non-empty result *only if it is certain* that the result is an over-approximation of the dynamic behavior. For instance, consider the following example of heap store instructions:

```
1 x.fld = new A();  // abstract object a1
2  // x.fld points-to set is {a1}
3 y.fld = notFullyAnalyzed();
4  // x.fld points-to set is {}
```

After the first instruction, the points-to set of access path `x.fld` is computed to be {`a1`}. However, in most cases, the analysis will not be able to ascertain that `x` and `y` are not aliased. Therefore, after the second instruction, the points-to set of `x.fld` will be empty, i.e., unknown. This reflects well the defensive nature of the analysis: whenever uncertain, points-to sets will default to empty, i.e., ⊤.

Generally, since the analysis is access-path based, store instructions certain to operate on the same object perform *strong updates*, while store instructions that *possibly* operate on the same object perform *weak updates*:

```
1 x.fld = new A();  // abstract object a1
2  // x.fld points-to set is {a1}
3 x.fld = new B();  // abstract object b1
4  // x.fld points-to set is {b1}
5 y.fld = new B();  // abstract object b2
6  // x.fld points-to set is {b1,b2}
```

In this case, the points-to information of access path `x.fld` is set to {`b1`} after the second store instruction, ignoring the previous contents! (The example assumes that types `A` and `B` are both compatible with the static type of `x.fld`.) After the third store instruction, however, a new element is added to the points-to set—again, under the assumption that the analysis cannot determine whether `x` and `y` are aliased.

As before, if any of the involved points-to sets is empty, both strong and weak updates yield an empty points-to set. For instance, replacing any of the three allocations above with a call to opaque code would make all subsequent points-to sets of `x.fld` be empty.

The treatment of these features gives a taste of the defensive nature of the analysis. The handling of other features (e.g., virtual method calls—see Section 4) poses interesting challenges, but the principle of defensiveness guides the overall design: whenever unsure, the analysis infers nothing, i.e., the points-to sets remain empy, signifying an implicit ⊤.

Method calls. Defensive analysis computes sound may-point-to information simultaneously with *sound call-graph* information. The analysis employs the same principles for the call-graph representation as for points-to: a finite set of method call targets means that the set is guaranteed bounded, while an empty set of method call targets means that the analysis cannot (yet) establish that all target methods are known.

To compute a sound over-approximation of method call targets, one needs a bounded may-point-to set for the receiver. Otherwise, the receiver object could be unknown—e.g., an instance of a dynamically loaded class—resulting in an unsound call-graph.

When the set of method call targets is not bounded, dynamic calls cannot be resolved and the analysis has to be very conservative. For instance, in the example below, a conventional unsound analysis would resolve the virtual call `x.foo()` to, at least, the method `A::foo`, i.e., `foo` in class `A`.

```
1 if (complexCondition()) {
2   x = new A();  // abstract object a1
3 } else {
4   x = notFullyAnalyzed();
5 }
6 x.foo();
```

In contrast, recall that for a defensive analysis the points-to set of `x` at the point of the call to `foo` is empty. Accordingly, the defensive analysis does *not* resolve the virtual call at all. This means that all heap information (i.e., all access-path points-to information, except for *trivial* access paths consisting of a single local variable and no fields) that held before the method call ceases to hold after it! (There are notable exceptions—e.g., for access paths with final fields, or for cases when an escape analysis can establish that some part of the heap does not escape into the called method. Section 5 discusses such intricacies.)

When method calls *can* be resolved, the target methods have to be analyzed under a context uniquely identifying the callee. A defensive analysis may know all methods that can get called at a certain point, but *it cannot know all callers of a method*. Consider the following example:

```
1 void callee(A y) { ... }
2 void caller() {
3   A x = new A(); // abstract object a1
4   callee(x); // call to callee
5 }
```

Assume that there is no other call to `callee` anywhere in the program. An unsound analysis would establish that variable `y` in `callee` points to abstract object `a1`. A defensive analysis, however, cannot do the same unconditionally. The points-to set of `y` without context information has to be the empty set! The reason is that there may be unknown callers of `callee`: either in existing code, via reflection, or in dynamically loaded code. Such callers could pass different objects as arguments to `callee` and the analysis cannot upper-bound the set of such arguments. Thus, the only safe answer for a defensive analysis is $\top$—i.e., an empty set.

Thus, in order to propagate analysis results inter-procedurally, a defensive analysis has to leverage context information. In the above example, what the analysis will establish is that `y` points to `a1` *conditionally*, under context `4`, signifying the call-site instruction (line 4 in our listing).

The above implies that the use of context in a defensive analysis is rather different than in a traditional unsound points-to analysis. Contexts in standard points-to analysis can be *summarizing*: a single context can merge arbitrary concrete (dynamic) executions, as long as any single concrete execution maps uniquely to a context. For instance, a 1-object-sensitive analysis (Milanova et al. 2005) merges all calls to a method as long as they have the same abstract receiver object, independently of call sites.

Context in a conventional analysis only adds *precision*, relative to a context-insensitive analysis. In contrast, context in a defensive analysis is necessary for *correctness*: since information is collected per-program-point, propagating points-to sets from a call site to a callee can only be done under a context that identifies the call-site program point. Contexts cannot freely summarize multiple invocation instructions, because there may be others, yet unknown, invocations that would result in the same context.

Therefore, a context-sensitive defensive analysis has to be, at a minimum, *call-site sensitive* (Sharir and Pnueli 1981; Shivers 1991): the call site of an analyzed method has to be part of the context (as will, for deeper context, the call site of the caller, the call site of the caller's caller, etc.). Other kinds of context (e.g., object-sensitive context (Milanova et al. 2005)) can be added for extra precision.

Instruction	*Operand Types*	*Description*
i : `v = new T()`	$I \times V \times T$	Heap Allocations
i : `v = u`	$I \times V \times V$	Assignments
i : `v = u.f`	$I \times V \times V \times F$	Field Loads
i : `v.f = u`	$I \times V \times F \times V$	Field Stores
i : `v.meth(*)`	$I \times V \times M \times V^n$	Virtual Calls
i : `return`	I	Method Returns
i : `<unknown>`	I	Unknown instruction (i.e., any other)

Fig. 1. Intermediate Representation instruction Set.

4 A MODEL OF DEFENSIVE ANALYSIS

We next present a rigorous model of our defensive analysis. The model uses a minimal intermediate language that captures the essence of the approach. The language can be straightforwardly enhanced with features such as arrays, static members and calls, exceptions, etc.

4.1 Preliminaries

Figure 1 shows the form of the input language. The domains of the analysis (and meta-variables used subsequently, plain or primed) comprise:

- `v`, `u` $\in V$, a set of variables,
- `T`, `S` $\in T$, a set of types,
- `f`, `g` $\in F$, a set of fields,
- `meth` $\in M$, a set of methods,
- $i, j \in I$, a set of instruction labels,
- $c, d \in C$, a set of contexts,
- $\widehat{o}, \widehat{o_i} \in O$, a set of abstract objects, potentially identifying their allocation instruction,
- `p` $\in P$, a set of access paths (i.e., the set $V(.F)*$),
- $n \in \mathbb{N}$, the set of natural numbers.

The analysis input consists of a set of instructions, linked into a control-flow graph, via relation $i \xrightarrow{next} j$ (over $I \times I$). The input program is assumed to be in a single-return-per-method form. We employ type information as well as other symbol table information, accessed through some auxiliary functions and predicates:

- $\texttt{meth}_{\texttt{T}}$ is the result of looking up method signature `meth` in type `T`.
- `meth`(n) is the n-th instruction of method `meth`.
- We overload the $\in$ operator to more than set membership, in unambiguous contexts, namely: $i \in$ `meth` (instruction is in method), `f` $\in$ `p` (field is in access path), $\widehat{o} \in$ `T` (abstract object is of type), `v` $\in$ `T` (variable is of type).
- $arg_n^{\texttt{meth}}$ and arg_n^i denote the n-th formal or actual arg of a method and invocation instruction, respectively. (By convention, the `this`/base variable of a method invocation is assumed to be the 0-th argument.)
- `p`[`v`/`u`] is the access path resulting from changing the base of access path `p` from `v` to `u` (if applicable).

4.2 Analysis Structure

Figure 2 shows the analysis specification. We recommend following the figure together with our text explaining the rules: although the rules are precise (transcribed from a mechanized logical specification) some are hard to follow without explanation of their intent beforehand.

The analysis computes the following relations:

$$\text{(ALLOC)}\ \frac{i:\ \mathtt{v} = \mathtt{new\ T}()\qquad i \in \mathtt{meth}\qquad \overline{\mathtt{meth}}^{c}}{i:\mathtt{v} \xrightarrow{\text{OUT}}_{c} \widehat{o_i}} \qquad \text{(MOVE)}\ \frac{i:\ \mathtt{v} = \mathtt{u}\qquad i:\mathtt{p} \xrightarrow{\text{IN}}_{c} \widehat{o}}{i:\mathtt{p}[\mathtt{u}/\mathtt{v}] \xrightarrow{\text{OUT}}_{c} \widehat{o}}$$

$$\text{(LOAD)}\ \frac{i:\ \mathtt{u} = \mathtt{v.f}\qquad i:\mathtt{v.f} \xrightarrow{\text{IN}}_{c} \widehat{o}}{i:\mathtt{u} \xrightarrow{\text{OUT}}_{c} \widehat{o}} \qquad \text{(STORE-1)}\ \frac{i:\ \mathtt{u.f} = \mathtt{v}\qquad i:\mathtt{v} \xrightarrow{\text{IN}}_{c} \widehat{o}}{i:\mathtt{u.f} \xrightarrow{\text{OUT}}_{c} \widehat{o}}$$

$$\text{(STORE-2)}\ \frac{i:\ \mathtt{u.f} = \mathtt{v}\qquad i:\mathtt{v} \xrightarrow{\text{IN}}_{c} \widehat{o}\qquad i:\mathtt{w.f} \xrightarrow{\text{IN}}_{c} \widehat{o'}\qquad \mathtt{w} \neq \mathtt{u}}{i:\mathtt{w.f} \xrightarrow{\text{OUT}}_{c} \widehat{o}\qquad i:\mathtt{w.f} \xrightarrow{\text{OUT}}_{c} \widehat{o'}}$$

$$\text{(CFG-JOIN)}\ \frac{j \xrightarrow{next} i\qquad j:\mathtt{p} \xrightarrow{\text{OUT}}_{c} \widehat{o}\qquad \forall k : (k \xrightarrow{next} i) \implies (k:\mathtt{p} \xrightarrow{\text{OUT}}_{c} *)}{i:\mathtt{p} \xrightarrow{\text{OUT}}_{c} \widehat{o}}$$

$$\text{(FRAME-1)}\ \frac{i:\mathtt{v} \xrightarrow{\text{IN}}_{c} \widehat{o}\qquad \neg(i:\mathtt{v} = *)\qquad \neg(i:\ \texttt{<unknown>})}{i:\mathtt{v} \xrightarrow{\text{OUT}}_{c} \widehat{o}}$$

$$\text{(FRAME-2)}\ \frac{i:\mathtt{p} \xrightarrow{\text{IN}}_{c} \widehat{o}\quad \mathtt{p} = \mathtt{v.}*\quad \neg(i:\ *\mathtt{.meth}(*))\quad \neg(i:\ *\mathtt{.g} = *)\quad \neg(i:\mathtt{v} = *)\quad \neg(i:\ \texttt{<unknown>})}{i:\mathtt{p} \xrightarrow{\text{OUT}}_{c} \widehat{o}}$$

$$\text{(FRAME-3)}\ \frac{i:\ *\mathtt{.f} = *\qquad i:\mathtt{p} \xrightarrow{\text{IN}}_{c} \widehat{o}\qquad \mathtt{f} \notin \mathtt{p}}{i:\mathtt{p} \xrightarrow{\text{OUT}}_{c} \widehat{o}}$$

$$\text{(CALL)}\ \frac{i:\ \mathtt{v.meth}(*)\qquad i:\mathtt{v} \xrightarrow{\text{IN}}_{c} \widehat{o}\qquad \widehat{o} \in \mathtt{T}\qquad c' = \mathcal{NC}(i,c,\widehat{o})}{\overline{\mathtt{meth_T}}^{c'}\qquad i \xrightarrow{calls}{}^{c}_{c'}\ \mathtt{meth_T}}$$

$$\text{(ARGS)}\ \frac{i \xrightarrow{calls}{}^{c}_{c'}\ \mathtt{meth}\qquad i:\mathtt{p} \xrightarrow{\text{IN}}_{c} \widehat{o}\qquad j = \mathtt{meth}(0)}{j:\mathtt{p}[arg_n^i/arg_n^{\mathtt{meth}}] \xrightarrow{\text{IN}}_{c'} \widehat{o}}$$

$$\text{(RET)}\ \frac{\begin{array}{l} j:\ \mathtt{return}\qquad j \in \mathtt{meth}\qquad i \xrightarrow{calls}{}^{c}_{d}\ \mathtt{meth}\qquad j:\mathtt{p} \xrightarrow{\text{IN}}_{d} \widehat{o}\qquad \mathtt{p} = \mathtt{v.}* \\ \forall \mathtt{meth}', c' : (i \xrightarrow{calls}{}^{c}_{c'}\ \mathtt{meth}') \implies \\ (\exists j', \mathtt{q}' : (j':\ \mathtt{return}) \wedge (j' \in \mathtt{meth}') \wedge (\mathtt{p} = \mathtt{q}'[arg_n^{\mathtt{meth}'}/arg_n^i]) \wedge (j':\mathtt{q}' \xrightarrow{\text{IN}}_{c'} *)) \end{array}}{i:\mathtt{p}[arg_n^{\mathtt{meth}}/arg_n^i] \xrightarrow{\text{OUT}}_{c} \widehat{o}}$$

Fig. 2. Inference Rules for Defensive Points-to Analysis

- The "access path points-to" relation, in two varieties, before and after an instruction: $i : p \xrightarrow{\text{IN}}_c \widehat{o}$ and $i : p \xrightarrow{\text{OUT}}_c \widehat{o}$ (p may point to $\widehat{o}$ before/after instruction i executed under context c). This is our sound may-point-to relation: if, at the end of the analysis, the set of $\widehat{o}$s for given i, p, c is not empty, it will be a superset of the abstract objects $\widehat{o}$ pointed by p at the given program point and context during any dynamic execution.[3]
- The "may-call" relation, i.e., our sound call-graph representation: $i \xrightarrow{calls}_{c'}^{c}$ meth (instruction i executed under context c may call method meth and the resulting context will be c').
- The "reachable" relation, $\overline{\texttt{meth}}^{\texttt{c}}$, denoting that method meth is reachable under context c, and should, thus, be analyzed. This relation is partially populated when the analysis starts: it holds an initial set of methods, under the empty context **EMPTY**, that should be analyzed.

Alloc, Move, Load, Store-1. The first four rules of the analysis are rather straightforward. The ALLOC rule is the only one with some minimal subtlety: if an object is freshly allocated, we know that the variable it is directly assigned to points to it. This inference is valid in any reachable context, even the initial, making-no-assumptions, **EMPTY** context. Therefore this rule is responsible for kickstarting the analysis, producing the first points-to inferences (valid locally) that will then propagate.

Store-2. The STORE-2 rule is the first one exhibiting the defensive and lazy features of the analysis. The rule performs a "weak update" on points-to sets of possibly affected access paths, as long as they are guaranteed to be bounded, i.e., they are non-empty. At a store instruction, u.f = v , if an access path w.f has a base explicitly different from u (with f being the same), then its points-to set is augmented with any element ($\widehat{o}$) of the points-to set of v, while maintaining its original elements ($\widehat{o'}$). This rule defensively adds more information to guarantee an over-approximation in the case of access paths that may be aliases for the same object. The subtlety of the rule lies in its handling of empty points-to sets. If *either* of the points-to sets (of v or of w.f) is empty before the instruction, the rule does not match, hence the points-to set of w.f after the instruction does not acquire any contents. This is consistent with our sound handling: if the earlier contents or the update cannot be upper-bounded, then the resulting points-to set cannot be, either.

CFG-Join. The next rule deals with merging information from an instruction's predecessors (or merely propagating it, in the case of a single predecessor).

Informally, the rule states that if *some* predecessor instruction, j, has established that p can point to $\widehat{o}$, *and* if all other predecessors, k, establish that p points to *something* (so that its points-to set is non-empty, i.e., bounded) then the information is propagated to the points-to relation of the successor instruction. (We use * to mean "any value", throughout the rules.) Note the defensive handling: if even a single predecessor has an unbounded (i.e., empty) points-to set for p, then the rule is not triggered and the resulting points-to set remains empty.

Frame-1, Frame-2, Frame-3. The next three rules are *frame rules*, responsible for the propagation of unchanged information.

Informally, the first rule merely says that points-to information for local variables (i.e., an access path consisting of just "v") is maintained after an instruction, if it existed before it, as long as the instruction does not directly assign the local variable (as is the case for a load, move, or allocation directly into this local variable). The soundness of this rule is predicated on our earlier assumption of *stack frame isolation*: local variables are isolated from each other, thread-private, and private to their allocating method. Therefore their points-to set cannot change, except with instruction such as the above.

[3] To be precise, concrete objects arise during execution but we are considering their standard mapping to abstract objects, per allocation site.

This is the first time we see a treatment of `<unknown>` instructions, which can encode any richer instruction set than our basic intermediate language. The analysis conservatively avoids propagating any points-to information over an unknown instruction.

The next two rules apply in the case of complex access paths, i.e., of length 2 or more. (Actually rule FRAME-3 also applies to variable-only access paths, but not meaningfully: that case is subsumed by FRAME-1.) First, similarly to the earlier rule, points-to information for the access path is maintained after an instruction (assuming it held before it) unless the instruction assigns the same base variable (again via a load, move, or allocation), or is a call, store, or unknown. Second, points-to information for complex access paths is propagated over all store instructions that affect fields not participating in the access path.

The soundness of these rules is predicated on the *object isolation* and *concurrency model* assumptions of Section 2.3. Under these assumptions, the only way to change the points-to set of an access path is via store instructions (on the same field), changing the base of the access path, invoking (potentially opaque) methods, and executing unknown instructions (including `monitorenter`/`monitorexit`). The rules have strong preconditions to preclude these cases. At the level of the model, we only care about soundness under the given assumptions, no matter how strict. In Section 5 we will discuss practical enhancements—e.g., when method calls are fine because the analysis has computed the full potential of their effects on the heap.

Call. The next rule uses points-to information to establish a sound call-graph. The $i \xrightarrow{calls}{}^{c}_{c'}\ \texttt{meth}$ relation over-approximates information using the same approach as points-to sets: for a given invocation site, i, and context, c, the relation holds either an empty set (i.e., no matching values exist for (i, ctx)—denoting an unbounded set of destinations—or an over-approximation (i.e., a superset) of all possible targets of the invocation at i under c.

The rule is mostly a straightforward lookup of the target method, based on the receiver object's type. There are a couple of subtleties, however. The receiver object needs to have an upper-bounded (i.e., non-empty) points-to set, a new context is constructed using function $\mathcal{NC}$, and the target method is considered reachable under the new context. The exact definition of $\mathcal{NC}$ will determine the context-sensitivity of the analysis. (We will return to this point promptly.)

Args. The ARGS rule handles points-to information propagation over calls, from caller to callee. Points-to information for rebased access paths is established for the first instruction (j = `meth`(0)) of a called method, under the callee's established context. The rule examines all access paths whose base variable is an actual argument of the call, as long as they have some points-to information (before the invocation).

Recall our discussion of Section 3 regarding method calls and the use of context. The points-to information established at a callee cannot be conflating different callers—there may be unknown callers for the same method, either in existing code (e.g., via reflection) or in dynamically loaded code. Therefore, if we might mix callers, the only sound inference for local points-to sets is `T`: we cannot bound the values that all callers may pass. Instead, we need to have contexts that uniquely identify the caller, so that we can safely propagate bounded points-to sets.

A straightforward way to ensure that the pair $(\texttt{meth}, c')$ uniquely identifies invocation instruction i and context c is to use *call-site sensitivity* (Sharir and Pnueli 1981; Shivers 1991): c' is formed by combining i and c—that is, $\mathcal{NC}(i, c, \widehat{o}) = cons(i, c)$. (Contexts can typically grow only up to a pre-determined depth, at which point the $\mathcal{NC}$ function will not return anything, the CALL rule will fail to make a $\xrightarrow{calls}$ inference, hence the current rule will not fire, leaving points-to sets at the callee empty, i.e., `T`.)

Ret. The final rule perform a similar propagation of values, this time from callee to caller. The rule is significantly complicated by its last condition (the forall-exists implication), which is key for soundness. The rule states that if some callee has points-to information for complex access path `p` at a return point, then this information is propagated to the caller, provided that *all other callees* for the same instruction, i, and caller context, c, also have *some* (i.e., non-empty) points-to information for the same access path `p` at their return point. A further complication

is that access path `p` will appear rebased differently for each one of the callees—e.g., access path `actual.field` may appear as `formalA.field` and `formalB.field` in two callees `A` and `B`. The rule has to also account for such rebasing.

Note also the earlier condition that access path `p` be complex, i.e., to have length greater than 1. This reflects call-by-value semantics for references: for a call `foo(actual)` to a method with signature `foo(F formal)`, the points-to information of access path `formal` is not reflected back to the caller, yet the points-to information of longer access paths, e.g., `formal.fld`, is.

The handling of a method return is the only point where a context can become stronger. Facts that were inferred to hold under the more specific context, c', are now established, modulo rebasing, under c. Since c' has to uniquely identify c, typically c will be shorter by one context element.

4.3 Reasoning

We prove the soundness of the analysis under an informal language model. We do not attempt to formalize the full effects of opaque code (e.g., what reflection or native code can or cannot do). Verified low-level correctness is not the level of assurance that our work attempts to achieve. We instead want to establish that the rules, as stated, are careful to always compute over-approximate finite points-to sets, or otherwise to produce empty sets, under the standard understanding of our assumptions of Section 2.3. For instance, it is clear from the "stack frame isolation" assumption that local variables cannot change values except by action of the current instruction, i.e., that rule FRAME-1 is alone responsible for soundly transfering such points-to information from the program point before an instruction to after. A detailed model that formally captures "stack frame isolation" is perhaps desirable assurance (in the vein of verified compilers) but adds nothing to the effort to *invent* a realistic, sound points-to analysis. By analogy, sound compiler optimizations (i.e., ones that do not break the program) exist in virtually all mainstream compilers, but a minuscule fraction of those have been formally verified.

There are two main properties of the defensive analysis:

- Soundness: the analysis computes an over-approximation of points-to sets that may arise during any program execution: either an empty set (trivially over-approximate) or a superset. (Alternatively, one can state the property as "all non-empty sets computed are over-approximate", but we prefer to fold in the special case of empty sets into the definition of "over-approximate" to avoid repeatedly adding a special case for empty in all our statements.)
- Laziness: the analysis does not waste work; elements that enter a points-to set are never removed by reverting the set to the $\top$ value (i.e., an empty set).

THEOREM 4.1. *There exists an evaluation order of the rules, such that the defensive analysis model is sound, i.e., it over-approximates all relevant points-to sets arising during program execution, under the assumptions of Section 2.3.*

PROOF. The proof is inductive. Initially, all points-to/call-target sets encoded in relations $i : \mathtt{p} \xrightarrow{\text{IN}}_c$, $i : \mathtt{p} \xrightarrow{\text{OUT}}_c$, $i \xrightarrow{calls}_{c'}^{c}$ are empty. (We treat relation $i : \mathtt{p} \xrightarrow{\text{IN}}_c \widehat{o}$ as encoding a set of $\widehat{o}$s for given $i, \mathtt{p}, c$; relation $i \xrightarrow{calls}_{c'}^{c} \mathtt{meth}$ as encoding a set of `meth`s for given i, c, c', etc.)

Therefore, we start from a trivially over-approximate state.

Importantly, the inductive step does *not* hold for a single application of a rule. Intermediate states of evaluation may not be over-approximate: an element may enter a set before the rest of its contents. (For instance, consider a statement `v = u` and prior points-to set $\{\widehat{o_1}, \widehat{o_2}\}$ for `u`. A single application of the MOVE rule for $\widehat{o_1}$ will leave the points-to set of `v` in a non-over-approximate state: the set will be missing the $\widehat{o_2}$ value.)

Thus, the inductive step applies to states after past rules have been evaluated fully. Consider a rule R as a monotonic update to a set of values d. That is, $R(d) \supseteq d$. A rule has been fully evaluated at fixpoint, i.e., when $R(d) = d$. The next inductive step considers the state after a full evaluation of any rule.

The inductive step of the proof is captured in a lemma:

LEMMA 4.2. *The analysis rules preserve soundness under full single-rule evaluation. That is, if relations* $i : \mathtt{p} \xrightarrow{IN}_{c}$, $i : \mathtt{p} \xrightarrow{OUT}_{c}$, *and* $i \xrightarrow{calls}^{c}_{c'}$ *encode over-approximate points-to/call-target sets before a full evaluation of a rule, they will encode over-approximate sets after a full evaluation.*

PROOF SKETCH OF LEMMA 4.2. The lemma is established by exhaustive examination of the rules. We mentioned key parts of the reasoning in our earlier presentation of the rules. All rules over complex access paths (i.e., of length ≥ 2 affect the heap and require the "concurrency model" and "object isolation" assumptions of Section 2.3. Rules on plain-variable access paths use the "stack-frame isolation" assumption. Every rule is careful to produce values for points-to/call-target sets only if all input sets are non-empty, and to consider all possible such values. For rules CALL, ARGS, and RET the lemma holds only under the previously-stated assumption on the $\mathcal{NC}$ constructor: the pair $(\mathtt{meth}, c')$ needs to uniquely identify invocation instruction i and context c. Consider, for example, rule ARGS. We need to establish that the points-to set $j : \mathtt{p}[arg_n^i / arg_n^{\mathtt{meth}}] \xrightarrow{\text{IN}}_{c'}$ is over-approximate given that $i : \mathtt{p} \xrightarrow{\text{IN}}_{c}$ is. (The rule form makes the former be a superset of the latter, we need to reason that they are actually the same set.) Instruction j uniquely identifies method `meth` and actual-to-formal access-path rebasing can never merge access paths (since different formal variables cannot have the same names). If c' and `meth` arise for only a single call-site and caller-context pair, (i, c), then the property holds. □

The lemma establishes the inductive step of our proof. The sets computed by the analysis are initially over-approximate and remain over-approximate after every full evaluation of a single rule. At fixpoint, when full evaluation of any rule no longer changes the output sets, the property holds, concluding the theorem's proof. □

An interesting question is whether *any evaluation order of the rules* is guaranteed to yield sound points-to sets at fixpoint. The answer is "almost yes". All but one analysis rules are monotonic (in the usual domain of sets, i.e., with the empty set at the bottom), therefore yield a confluent evaluation: any order will yield the same result at fixpoint. (We have a machine-checked proof of the latter property, by encoding the rules in the Datalog language, which allows only recursion through monotonic inferences.) The single exception is the RET rule. There is hidden non-monotonicity in the ∀ iteration over call-graph edges. If the CALL rule is not fully evaluated when the RET rule applies, it is possible to produce points-to sets that will later be invalidated, because more callees will be discovered (for whom the points-to relationship does not hold for the given access path). Therefore, for soundness to hold, the analysis rules have to always apply in such a fashion that the CALL rule is fully evaluated (not globally but on its own, per the earlier definition) before the RET rule is considered. This evaluation order should be enforced by any sound implementation of the rules of Figure 2.

Based on the above observation on the rules' monotonicity, we also establish our laziness result.

THEOREM 4.3. *A points-to set encoded in our analysis relations never switches from non-empty to empty, as long as the* Ret *rule is applied only during local fixpoints (i.e., after full evaluation) of the* Call *rule.*

5 IMPLEMENTATION AND DISCUSSION

We have implemented defensive analysis in the Datalog language and integrated it with the DOOP Datalog framework for may-point-to analysis of Java bytecode (Bravenboer and Smaragdakis 2009). The implementation consists of over 400 logical rules in 3KLoC, yet the minimal model of Section 4 captures well its essential features. We also completed a second, largely equivalent, implementation on the Souffle Datalog engine (Scholz et al. 2016).

The defensive analysis model admits several enhancements and refinements, as well as gives rise to observations. We discuss such topics next, especially noting those that pertain to our full-fledged implementation of the analysis.

Observations. A defensive analysis is naturally modular, yet the question is whether it can produce useful results. The analysis can be applied to any subset of the code of an application or library and it will produce sound inferences. Omitting code merely means that more points-to sets will end up being empty: the analysis only infers points-to sets when an upper-bound of their contents is known based on the current code under analysis. This defensive approach, however, may end up computing too many empty points-to sets. Therefore, a useful concept is that of the analysis's *coverage*: for how many program elements (e.g., local variables) can the analysis produce non-empty points-to information?

Additionally, a defensive analysis is not in competition with a conventional, unsound analysis, but instead complements it. The defensive analysis computes which of the points-to sets have known upper bounds and which are potentially undetermined. If, instead of an empty set, a client desires to receive the (incomplete) subset of known contents for non-bounded points-to sets, the results of the two analyses can be trivially combined.

Pragmatics. With minor adaptation, the analysis logic can work on static single assignment (SSA) input. Our implementation is indeed based on an SSA intermediate language. The benefit is that for trivial access paths (just a single variable) points-to information does not need to be kept per-instruction: the points-to set remains unchanged, since the variable is not re-assigned.

A full-fledged analysis should cover more language features than the model of Section 4. Our implementation handles, in a manner similar to the earlier rules, features such as static and special method invocations, static fields, final fields, constructors (also implicitly initializing fields to `null`), and more.

Expanding the Analysis Reach. Defensive analysis is naturally pessimistic. Its key feature is that it will populate points-to sets only when it can establish that they are bounded. However, the analysis uses simplistic techniques to establish such boundedness, i.e., it recognizes guaranteed-safe cases.

There are several sound inferences that the analysis could make but the model of Section 4 does not. Although defensive analysis will never reach the inferences of an unsound analysis (even without any opaque code), it can be enhanced to approach it. Arbitrarily complex mechanisms can be added to increase the coverage of the analysis (i.e., the true properties it can infer precisely):

- The rule shown earlier for control-flow merge points is conservative. Information propagates at control-flow merge points if all of the predecessors have some points-to information for the access path in question. This condition is too strict: several predecessors will not have points-to information for an access path simply because the access path is not even assigned in the predecessor branch (e.g., it is based on a local variable that is set on a different branch only). Consider a program fragment:

```
1 x.f = new A();
2 while (...) {
3   y = x.f;
4 }
```

  The head of the loop has two control-flow predecessors: one due to linear control flow and one due to the loop back-edge. However, the loop itself does not change the points-to set of `x.f`. It is too conservative to demand that the back-edge also have a bounded points-to set for `x.f` before considering the linear control-flow edge.

  In our implementation we have special support for detecting that a program path does not affect an access path. We use this to limit the ∀ quantification of the rule to range over "relevant" predecessors. We note that this scenario only applies to complex access paths in practice, due to the SSA form of our input.
- When an unknown method call is encountered, the analysis assumes worst-case behavior with respect to its heap information. This can be relaxed arbitrarily by modeling system methods and annotating them appropriately. Possible information about calls includes "this library call does not affect user-level objects", "this method only affects its arguments", "this method does not affect static variables", etc. Additional manual modeling includes library collections (including arrays) which can be represented as abstract objects.

Our current implementation does some minimal modeling of library collections and annotates only a handful of methods, as a proof-of-concept. A representative example is that of method `Float.floatToRawIntBits`. This native method is called by the implementation of the `put` operation in Java `HashMap`s and, since it is opaque, would prevent all propagation of points-to information beyond a `put` call.

- The analysis coverage can be expanded by employing it jointly with a *must-alias* analysis, an *escape* analysis, and a *thread-escape* analysis. A must-alias analysis will increase the applicability of the rule for heap loads, and can be combined with the rule for heap stores to enable more strong updates. An escape analysis will result in less conservativeness in the propagation of information to further instructions (i.e., in frame rules). A thread-escape analysis can help relax our concurrency model. We currently support simple, conservative versions of all three analyses in our implementation, but do not enable them by default.

Context depth. As seen earlier, a defensive analysis may compute empty (`⊤`) points-to sets because it has reached its maximum context depth. It is worth pointing out, however, that method calls *further away* than the maximum context depth can influence the points-to inferences of a method. For an easy example, consider the case of a large number, N, of methods that form a call chain and unconditionally return to their callers what their callee returns to them. If the final (N-th) method returns a new object, then that object will propagate all the way back to the first method of the call chain, regardless of the maximum context depth, D. The limitation of context depth only concerns properties that *depend on* conditions established more than D calls back in the call-stack.

6 EVALUATION

There are five research questions that our evaluation seeks to answer:

- **RQ1:** Does defensive analysis produce coverage for large parts of realistic programs? Or do points-to sets overwhelmingly stay empty?
- **RQ2:** Does the coverage of defensive analysis benefit from its advanced features (i.e., inter-procedural handling, as well as handling of control-flow merging)?
- **RQ3:** Does defensive analysis have an acceptable running time, given that it is flow-sensitive and context-sensitive?
- **RQ4:** Does defensive analysis yield results that can benefit a client that requires soundness, such as an optimization?
- **RQ5:** Can benefits be obtained for a fully relaxed concurrency model, as opposed to the model of Section 2.3?

Setup. For comparisons, we use a baseline 2-object-sensitive/heap-sensitive analysis (*2objH*). This is the most precise analysis in the DOOP framework that still manages to scale to the majority of the DaCapo benchmarks. We use static best-effort reflection handling (`--enable-reflection-classic` flag), i.e., the analysis tries to statically resolve all reflection calls based on string matching.

We analyze, under JDK 1.7.0_75, the DaCapo benchmark programs (Blackburn et al. 2006) v.2006-10-MR2 as well as v.9.12-Bach. The 9.12-Bach version contains several different programs, as well as more recent versions of some of the same programs. (We show results for all of the v.2006-10-MR2 benchmarks and for those of the v.9.12-Bach benchmarks that could be analyzed by the DOOP framework in under 3 hours.) We also use two benchmarks (NTI, jFlex) from the Julia set by Nikolić and Spoto (2012). (The rest of the benchmarks in that set are Android applications, which the DOOP framework currently does not analyze for purely engineering reasons.)

We use the LogicBlox Datalog engine, v.3.10.14, on a Xeon E5-2667 v.2 3.3GHz machine with only one thread running at a time and 256GB of RAM.

Defensive analysis is run with a 5-call-site-sensitive context (*5def* for short). 3 instances (of 44 total) did not finish with the default precision in 3hrs: the 2objH baseline did not finish for jython and h2; xalan did not finish for the 5def analysis. In these cases we used lower precision: context-insensitive for the unsound analysis and 4-call-site-sensitive (*4def*) for defensive.

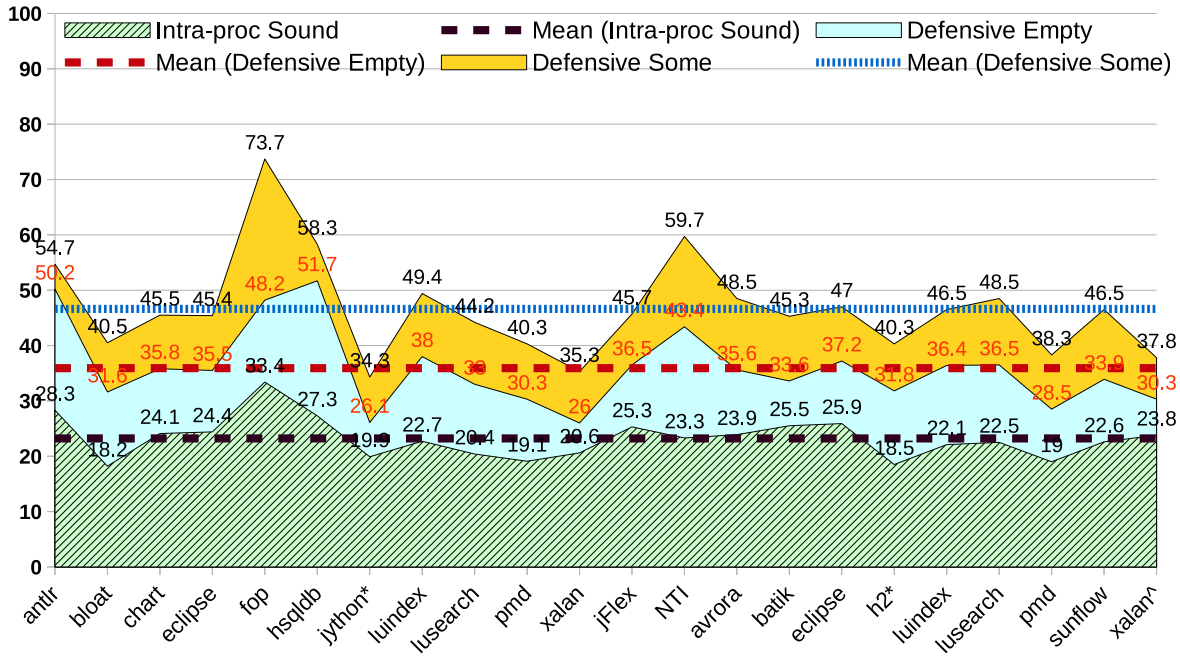

Fig. 3. Percentage of application variables (deemed reachable by baseline 2objH analysis) that have non-empty points-to sets for defensive analysis under some context and empty context (no assumptions). Intra-procedural sound points-to analysis (defensive minus the complex cases) shown as baseline. Arithmetic means are plotted as lines.

Coverage. Figure 3 shows the coverage of defensive analysis, i.e., the number of non-empty points-to sets computed for all benchmarks. The analysis yields non-empty points-to sets for a significant portion of each program—the median benchmark has **45.6%** of variables with points-to information for *some* context, while **35.5%** have points-to information for a context **EMPTY** (i.e., unconditionally). Both numbers are meaningful and the first probably more so. Recall that the meaning of inferences under *some* context is quite different for a defensive analysis: Many of the useful inferences of a defensive analysis will be under *some* context even when the inference holds under *all known* contexts (since there may be other calling contexts in opaque, and possibly not yet existing, code).

Thus, the defensive analysis achieves a large proportion of the benefits of an unsound analysis, while guaranteeing these results against uses of opaque code. We can answer **RQ1** affirmatively: defensive analysis covers a large part of realistic programs (over one-third unconditionally; close to one half under specific calling conditions), despite its conservative nature.

Figure 3 also helps answer **RQ2**. It contains results for an intra-procedural baseline analysis that captures the low-hanging fruit of sound reasoning: local variables that directly or transitively (via "move" instructions) get assigned an allocated object. That is, the "Intra-proc Sound" analysis does not contain any of the "defensive" logic for the hard-to-reason-about cases (control-flow merging, heap manipulation, and inter-procedural propagation), but is otherwise identical. The result answers **RQ2** affirmatively: defensive analysis has significantly higher coverage than the baseline intra-procedural analysis. (And the difference only grows when considering an actual client, in later experiments.)

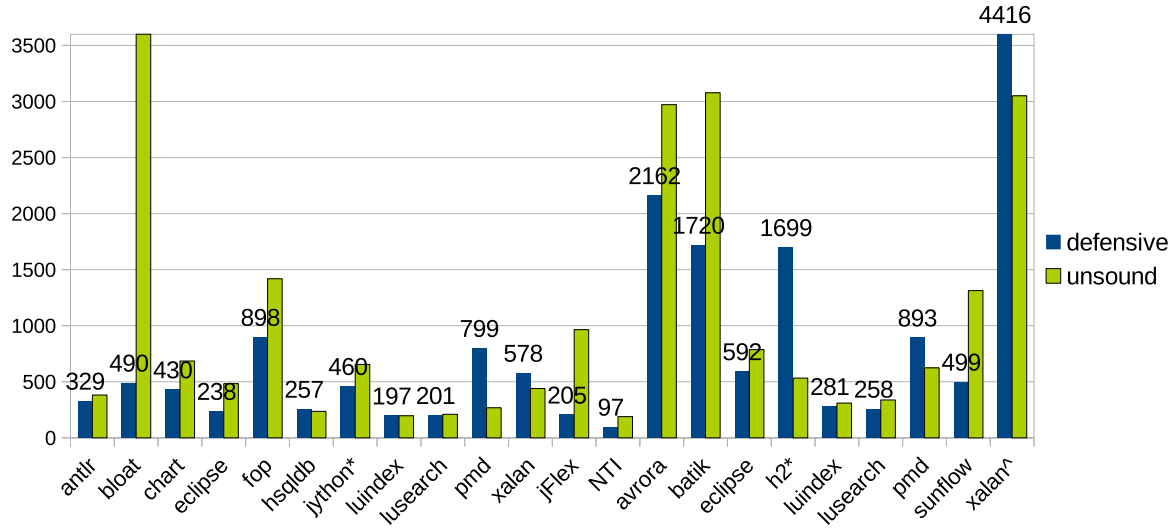

Fig. 4. Running time of defensive analysis, with running time of 2objH (with full unsound reflection handling) shown as a baseline. Numeric labels are for defensive analysis only to avoid crowding the plot.

Running time. Figure 4 show the running times of the analysis, plotted next to that of 2objH, for reference. This is not a comparison, since the two analyses are entirely dissimilar. However, the 2objH running times can serve as a baseline. As can be seen, the running times of defensive analysis are quite acceptable, although it is, at first glance, a prohibitively heavy analysis: flow-sensitive and 5-call-site-sensitive. This answers **RQ3** and confirms the benefits of laziness: a defensive analysis that only populates points-to sets once they are definitely bounded, achieves scalability for deep context.

Client analysis: devirtualization. Our baseline analysis, 2objH, is highly precise and effective in challenges such as devirtualizing calls (resolving virtual calls to a single target method). On average, it can devirtualize 89.3% of the calls in the benchmarks studied (min: 78.5%, max: 95.2%). However, these results are unsound. For optimization clients, such as devirtualization, soundness is essential. Figure 5 shows the virtual calls that defensive analysis devirtualizes, as a percentage of those devirtualized by the unsound analysis.

As can be seen, defensive analysis manages to recover a large part of the benefit of an unsound analysis (median **44.8%** for optimization under a context guard, **38.7%** for unconditional, **EMPTY** context, optimization), performing much better than the baseline intra-procedural must-analysis (at 14.6%). This answers **RQ4** affirmatively: the coverage of defensive analysis translates into real benefit for realistic clients.

Concurrency model. A compiler (JIT or AOT) author may (rightly) remark that the concurrency model of Section 2.3 is not appropriate for automatic optimizations. The Java concurrency model permits a lot more relaxed behaviors, so the analysis is not sound for full Java as stated. However, the benefit of defensive analysis is that it starts from a sound basis and can add to it conservatively, only when it is certain that soundness cannot possibly be violated. Accordingly, we can remove the assumption that all shared data are accessed while holding mutexes, by applying the load/store rules only when objects trivially do not escape their allocating thread. We show the updated numbers for the devirtualization client (now fully sound for Java!) in Figure 6. The difference in impact is minimal: **43%** of virtual call sites can be devirtualized conditionally, under some context, while **36%** can

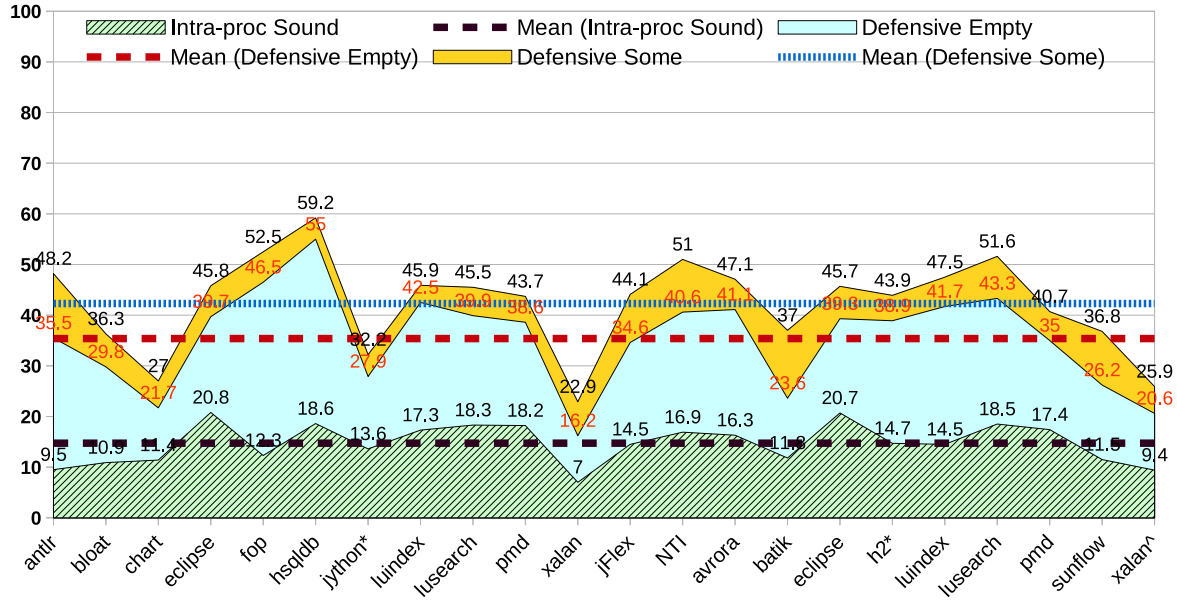

Fig. 5. Virtual call sites that are found to have receiver objects of a single type. These call sites can be soundly devirtualized. Numbers are shown as percentages of devirtualization achieved by unsound 2objH analysis.

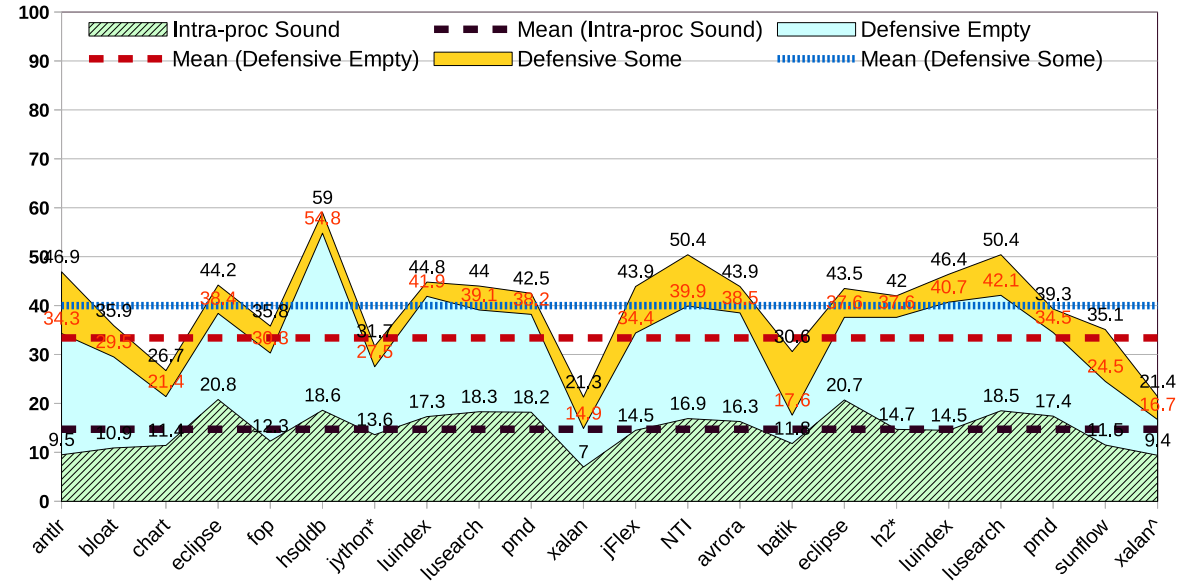

Fig. 6. Virtual call sites (percentage of 2objH) that are found to have receiver objects of a single type. Updates Figure 5, this time with soundness under a relaxed memory model.

be devirtualized unconditionally. This helps answer **RQ5**: defensive analysis can yield actionable results for a well-known optimization, under the Java memory model, for a large portion of realistic programs.

7 RELATED WORK

There are several pieces of past work that attempt to ensure a sound whole-program analysis, but none matches the generality and applicability of our approach. We selectively discuss some close relatives.

The standard past approach to soundness for a careful static analysis has been to "bail out": the analysis detects whether there are program features that it does not handle soundly, and issues warnings, or refuses to produce answers. This is a common pattern in abstract-interpretation (Cousot and Cousot 1977) analyses, such as Astrée (Delmas and Souyris 2007), which have traditionally emphasized sound handling of conventional language features. However, this is far from a solution to the problem of being sound for opaque code: refusing to handle the vast majority of realistic programs can be argued to be sound, but is not usefully so. In contrast, our work handles *all* realistic programs, but returns partial (but sound) results, i.e., produces non-empty points-to sets for a subset of the variables. It is an experimental question to determine whether this subset is usefully large, as we do in our evaluation.

Hirzel et al. (2004, 2007) use an online pointer analysis to deal with reflection and dynamic loading by monitoring their run-time occurrence, recording their results, and running the analysis again, incrementally. However, this is hardly a *static* analysis and its cost is prohibitive for precise (context-sensitive) analyses, if applied to all reflection actions.

Lattner et al. (2007) offer an algorithm that can apply to incomplete programs, but it assumes that the linker can know all callers (i.e., there is no reflection—the analysis is for C/C++) and the approach is closely tied to a specific flow-insensitive, unification-based analysis logic, necessary for simultaneously computing inter-related points-to, may-alias, and may-escape information.

Sreedhar et al. (2000) present the only past approach to explicitly target dynamic class loading, although only for a specific client analysis (call specialization). Still, that work ends up making many statically unsound assumptions (requiring, at the very least, programmer intervention), illustrating well the difficulty of the problem, if not addressed defensively. The approach assumes that only the public API of a "closed world" is callable, thus ignoring many uses of reflection. (With reflection, any method is callable from unknown code, and any field is accessible.) It "[does] not address the Java features of reloading and the Java Native Interface". It "optimistically assumes" that "[the extant state of statically known objects] remains unchanged when they become reachable from static reference variables". It is not clear whether the technique is conservative relative to adversarial native code (in system libraries, since the JNI is ignored). Finally, the approach assumes the existence of a sound may-point-to analysis, even though none exists in practice!

Traditional conservative call-graph construction (*Class Hierarchy Analysis (CHA)* (Dean et al. 1995) or *Rapid Type Analysis (RTA)* (Bacon and Sweeney 1996)) is unsound. Such algorithms explore the entire class hierarchy for matching (overriding) methods and consider all of them to be potential virtual call targets. However, CHA or RTA will miss classes that can be generated and loaded dynamically during program execution.

The conventional handling of reflection in may-point-to analysis algorithms for Java (Fink et al. 2013; Li et al. 2014, 2015; Livshits 2006; Livshits et al. 2005; Smaragdakis et al. 2015) is unsound, instead relying on a "best-effort" approach. Such past analyses attempt to statically model the result of reflection operations, e.g., by computing a superset of the strings that can be used as arguments to a `Class.forName` operation. The analyses are unsound when faced with a completely unknown string: instead of assuming that *any* class object can be returned, the analysis assumes that *none* can. The reason is that over-approximation (assuming any object is returned) would be detrimental to the analysis performance and precision. Even with an unsound approach, current algorithms are heavily burdened by the use of reflection analysis. For instance, the documentation of the WALA

library directly blames reflection analysis for scalability shortcomings (Fink et al. 2013),[4] and enabling reflection on the DOOP framework slows it down by an order of magnitude on standard benchmarks (Smaragdakis et al. 2015). Furthermore, none of these approaches attempt to model dynamic loading—a ubiquitous feature in Java enterprise applications.

8 CONCLUSIONS

Static analysis has long suffered from unsoundness for perfectly realistic language features, such as reflection, native code, or dynamic loading. We presented a new analysis architecture that achieves soundness by being *defensive*. Despite its conservative nature, the analysis manages to yield useful results for a large subset of the code in realistic Java programs, while being efficient and scalable. Additionally, the analysis is modular, as it can be applied to any subset of a program and will yield sound results.

We expect this approach to open significant avenues for further work. The analysis architecture can serve as the basis of other sound analysis designs. The defensive analysis itself can be combined with several other analyses (may-escape, must-alias) that have so far been hindered by the lack of a sound substrate.

REFERENCES

David F. Bacon and Peter F. Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proc. of the 11th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*. ACM, New York, NY, USA, 324–341.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of the 21st Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. DOI: http://dx.doi.org/10.1145/1167473.1167488

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '09)*. ACM, New York, NY, USA.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '77)*. ACM, New York, NY, USA, 238–252. DOI: http://dx.doi.org/10.1145/512950.512973

Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proc. of the 9th European Conf. on Object-Oriented Programming (ECOOP '95)*. Springer, 77–101. DOI: http://dx.doi.org/10.1007/3-540-49538-X_5

David Delmas and Jean Souyris. 2007. *Astrée: From Research to Industry*. Springer Berlin Heidelberg, Berlin, Heidelberg, 437–451. DOI: http://dx.doi.org/10.1007/978-3-540-74061-2_27

Stephen J. Fink and others. 2013. WALA UserGuide: PointerAnalysis. http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis. (2013).

Martin Hirzel, Amer Diwan, and Michael Hind. 2004. Pointer Analysis in the Presence of Dynamic Class Loading. In *Proc. of the 18th European Conf. on Object-Oriented Programming (ECOOP '04)*. Springer, 96–122. DOI: http://dx.doi.org/10.1007/978-3-540-24851-4_5

Martin Hirzel, Daniel von Dincklage, Amer Diwan, and Michael Hind. 2007. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.* 29, 2 (2007). DOI: http://dx.doi.org/10.1145/1216374.1216379

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA.

Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-Inferencing Reflection Resolution for Java. In *Proc. of the 28th European Conf. on Object-Oriented Programming (ECOOP '14)*. Springer, 27–53.

Yue Li, Tian Tan, and Jingling Xue. 2015. Effective Soundness-Guided Reflection Analysis. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings (Lecture Notes in Computer Science)*, Sandrine Blazy and Thomas Jensen (Eds.), Vol. 9291. Springer, 162–180. DOI: http://dx.doi.org/10.1007/978-3-662-48288-9_10

[4] "*Reflection usage and the size of modern libraries/frameworks make it very difficult to scale flow-insensitive points-to analysis to modern Java programs. For example, with default settings, WALA's pointer analyses cannot handle any program linked against the Java 6 standard libraries, due to extensive reflection in the libraries.*" (Fink et al. 2013)

Benjamin Livshits. 2006. *Improving Software Security with Precise Static and Runtime Analysis*. Ph.D. Dissertation. Stanford University.

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. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. DOI: http://dx.doi.org/10.1145/2644805

Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*. Springer, 139–160. DOI: http://dx.doi.org/10.1007/11575467_11

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41. DOI: http://dx.doi.org/10.1145/1044834.1044835

Durica Nikolić and Fausto Spoto. 2012. Definite Expression Aliasing Analysis for Java Bytecode. In *Proc. of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC '12)*, Vol. 7521. Springer, 74–89. DOI: http://dx.doi.org/10.1007/978-3-642-32943-2_6

Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. 196–206. DOI: http://dx.doi.org/10.1145/2892208.2892226

Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program flow analysis: theory and applications*, Steven S. Muchnick and Neil D. Jones (Eds.). Prentice-Hall, Inc., Englewood Cliffs, NJ, Chapter 7, 189–233.

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. Carnegie Mellon University.

Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Proc. of the Asian Symp. on Programming Languages and Systems (APLAS '15)*. Springer.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 485–495. DOI: http://dx.doi.org/10.1145/2594291.2594320

Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. 2000. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 196–207.