Defensive Points-To Analysis: Effective Soundness via Laziness

Yannis Smaragdakis
Dept. of Informatics and Telecommunications, University of Athens, Greece
yannis@smaragd.org

George Kastrinis
Dept. of Informatics and Telecommunications, University of Athens, Greece
gkastrinis@di.uoa.gr

Abstract

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 may be missing elements due to opaque code. Laziness enables great efficiency, allowing 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.

2012 ACM Subject Classification Software and its engineering → Compilers; Theory of computation → Program analysis; Software and its engineering → General programming languages

Keywords and phrases static analysis, soundness, defensive analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.23

Acknowledgements We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE), a Facebook Research and Academic Relations award, and an Oracle Labs collaborative research grant. We thank Anders Møller for helpful presentation suggestions.

1 Introduction

Soundness is a coveted property of static analyses, to the extent that the term is often colloquially used as a synonym for "correctness". For a may-analysis, soundness means that the analysis abstraction overapproximates all concrete executions. A sound value-flow or points-to analysis is one that computes, per program point or per variable, value sets that represent (at least) all values that could possibly arise at the respective point during any possible execution.

Full soundness is hard to achieve in practice due to code that cannot be analyzed (e.g., dynamically generated/loaded code, binary/native code) or dynamic language features (e.g., reflection, `eval`, `invokedynamic`). We collectively refer to such features as *opaque code*. For instance, the Java code below invokes an unknown method, identified by string `methodName`, over an object, `obj`.

```
Method m = obj.getClass().getMethod(methodName);
m.invoke(obj);
```

String `methodName` could be a true run-time value—e.g., read from a file or external resource. Object `obj` could itself be of a type not available during analysis—e.g., `obj` could be obtained through the network and statically typed using a vague interface or root-of-hierarchy type.

Faced with such complications, all past analyses that claim soundness have done so under *a priori* qualifications. Prominently, abstract-interpretation-based [8] approaches, such as Astrée [10], have long emphasized soundness. The conceptual form of such a soundness result is as follows:[1]

> An *Analysis* of programs in language *Lang* is sound relative to language subset $Lang'$ and executions set $Exec'$ iff:
> $\forall$ program $P \in Lang$: $P \in Lang' \wedge e \in Exec' \implies e \in \gamma(Analysis(P))$
> (where γ is the concretization function that maps abstractions in the output domain of *Analysis* to concrete executions in a universe $Exec$, superset of $Exec'$).

The problem with this formulation of soundness is that, although it yields provable theorems, the *a priori* qualification excludes virtually all realistic programs. The $Lang'$ or $Exec'$ of published proofs disqualify the vast majority of modern programs "in the wild". Language subset $Lang'$ will typically exclude all dynamic features (e.g., reflection) and/or executions subset $Exec'$ will disqualify all behaviors that are deemed too-dynamic (e.g., invoking dynamically-loaded code). Reflection alone disqualifies ∼80% of Java programs in the 461-program corpus of the recent Landman et al. study [16].

The above issues have led several members of the static analysis community to proclaim that "*all published whole-program analyses are unsound*" [21], i.e., their soundness guarantee does not apply to realistic programs, and similarly that "*[there is not] a single realistic whole-program analysis tool [...] that does not purposely make unsound choices*". The problem is, therefore, both theoretical and practical. Soundness theorems do not give guarantees for realistic programs. Implementations of analyses in tools happily perpetuate the illusion: they handle soundly the language features one can prove theorems about, while cutting corners in the sound handling of all *other* features, in order to demonstrate greater scalability or precision. For instance, in our earlier Java code fragment, even if the type of `obj` is known, many implemented static analyses will not consider all its methods (which now form a small finite set) as possible values of `m`, but will instead ignore the code altogether. This phenomenon has led to the introduction of the term *soundy* [21] to characterize such analyses. Despite the derogatory tone, "soundy" analyses are the current *good* case of static analyses! They are realistic analyses that handle all "normal" language features soundly.

In this work, we propose *defensive analysis*: a static analysis architecture that addresses the above soundness shortcomings. The basis of defensive analysis can be seen as a different conceptual formulation of soundness.

> An *Analysis* of program P in language *Lang* computes results, $Analysis(P)$, together with soundness marker sets, $Claim(P)$. The *Analysis* is sound iff:
> $\forall$ program $P \in Lang$, execution e: $e[Claim(P)] \in \gamma(Analysis(P))[Claim(P)]$
> (where γ is as before, and $e[Claim(p)]$ is the restriction of an execution e to program points with soundness claims, and the definition is similarly lifted to sets of executions).

In other words, the analysis imposes no (or very liberal) *a priori* restrictions to its soundness claims, but instead *computes* the claimed domain of its soundness: the program parts for

[1] This formulation is due to Xavier Rival of the Astrée project (e.g., [25]).

which the analysis result is sound. The soundness theorem applies to all (or most) programs, under all execution conditions—instead of eagerly disqualifying the vast majority of real-world programs. The extent of soundness is now defined over program points and becomes an experimentally measurable quantity: the size of $Claim(P)$ (which we term the *coverage* of the analysis) can be measured to quantify for which percentage of a program's points the analysis is guaranteed to produce sound results.

The challenge of defensive analysis is, thus, to distinguish parts of the program that are certain to not be affected by opaque code. Delineating "safe" from "unsafe" parts of the program is an ambitious goal, since opaque code can do virtually anything: it can add dynamically-generated subclasses with never-seen methods that get called (via dynamic dispatch or first-class functions) at unsuspecting program points; it can call any existing method or alter any field via reflection; it can interpose its own implementations at every place where the program uses a reflective lookup; worst of all, it can wreak havoc on all parts of the heap reachable from any reference that escapes into opaque code.

We designed and implemented a defensive may-point-to (henceforth just "points-to") analysis for Java. The analysis follows the above form, explicitly designating points-to sets that are sound, i.e., that contain at least all the values that may ever arise in actual executions. Soundness guarantees carry over to the implementation: the soundness proof explicitly models all other language features as `<unknown>` instructions and makes only weak, semantically-justified assumptions (e.g., a type-safe heap) about them. Soundness reasoning is defensive in that it establishes when the analysis can be certain to know the full contents of a points-to set, no matter what opaque code can do (within the stated weak assumptions).

In our effort to implement defensive analysis in a realistic package, we found *laziness* to be an essential feature—the analysis cannot scale without it for real-world programs. Laziness means that the analysis does not compute points-to sets unless it can also claim their soundness. That is, program points outside of the $Claim(P)$ set do not get populated at any point—they remain empty throughout the analysis. Consequently, all points-to sets with a potentially unbounded number of objects (e.g., sets that depend on reflection or dynamic loading) are represented as the empty set: the analysis never computes any contents for them. 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 complex escape or alias reasoning (i.e., "can this object *ever* escape into opaque code?"), but only best-effort logic (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".
Concretely, the work makes the following contributions:

- We offer a general static may-point-to analysis that yields sound results for realistic programs in the presence of opaque code.
- The analysis is 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. 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.

- 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 ($Claim(P)$) 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 Overview

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 separate 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.

Defensive analysis takes the above observation 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.[2]

In other words, an empty set can be thought to represent a bottom ($\bot$) value during the analysis computation: it just marks a set as having no known values (yet). A set stops being empty only when all the possible ways (in known or unknown code) to contribute values to it have been examined and are found to have bounded contents. At the end of the analysis, all sets that have remained empty signify that the analysis could not bound their contents, i.e., they do not belong in the set $Claim(P)$ of program points with soundness claims. Therefore an empty set after termination of the analysis is conceptually equivalent to a top ($\top$) value: the set could contain anything. This is consistent with the defensive nature of the analysis: not knowing all the values of a set is considered just as bad as knowing it can point to anything.

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., if the analysis were to stop at that point it would report them as having $\top$ values, meaning "the set can contain anything". This is a sound answer, and is only subsequently refined.

[2] 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.)

This lazy evaluation means that defensive analysis does not need to employ sophisticated mechanisms to simply be sound. For instance, instead of a precise over-approximate escape analysis, 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 Background and Illustrating Design Decisions

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

Baseline intra-procedural resoning. It is easy for an analysis to be sound locally, in an *intra-procedural* setting. For instance, when a variable is freshly assigned with a newly allocated object, we are guaranteed to soundly know its points-to set:

```
x = new A();  // abstract object a1, x points-to set is {a1}
```

We can also propagate such information transitively through local assignments (a.k.a. "move" instructions), as long as no opaque code can interfere. In the case of local variables, standard concurrency models (for Java, C++, etc.) do not allow interference from other threads, hence points-to sets remain sound, as long as the code itself does not call out to opaque code:

```
x = new A();  // abstract object a1, x points-to set is {a1}
y = x;  // y points-to set is {a1}
z = y;  // z points-to set is {a1}
```

This approach is one often taken by traditional compilers (ahead-of-time or just-in-time alike) in order to perform intra-procedural optimizations, such as those based on traditional data-flow analysis. (Later, in our experimental evaluation, we compare against such a baseline "intra-procedural sound" analysis.)

However, the challenge is to also reason soundly about *inter-procedural* behavior. This includes reasoning about the heap (i.e., reading fields of objects) and about method calls and returns, whose resolution may be dynamic. This will be the focus of the defensive analysis specification.

Inter-procedural elements. The large potential for opaque code to affect inter-procedural 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. [33] 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.

Instead, defensive analysis pessimistically computes that a points-to set is $\top$ (i.e., can contain anything) unless it is certain that its contents are bounded. When can the analysis know this, however? Such a guarantee of bounded contents typically comes from having

precisely 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, y.fld points-to set is {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 may well be able to establish.

In practice, our defensive analysis will do a lot more: it will track method calls, up to a maximum context depth, to ascertain when they can interfere with points-to sets. (If any interference is detected, the points-to set propagated forward is empty.) For instance, in the example code below, the analysis can know with certainty the points-to set of `x` on line 6, whenever method `foo` is called from line 3 of the program fragment.

```
1 y.fld = new A(); // abstract object a1, y.fld points-to set is {a1}
2 z.otherFld = new B();
3 foo(y);
4
5 void foo(W w) {
6   x = w.fld;  // x-for-call-site-3 points-to set is {a1}
7 }
```

Note the elements that contribute to such reasoning: The result holds soundly only when `foo` is called from the specific call site. This result is established only by tracking the value of `y.fld` (renamed to `w.fld` inside method `foo`) instruction-by-instruction all the way to line 6. The heap store instruction on line 2 is guaranteed to not affect `y.fld` (regardless of whether `z` and `y` alias or not), since Java guarantees object isolation and the reference is to a different field. (More on language model assumptions in Section 2.3.)

The above example helps illustrate the design choices of defensive analysis: it is a flow-sensitive, context-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, even across calls. 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.[3] However, we do know that `y.fld` certainly points to abstract object `a1` after line 1!

Laziness. 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. Defensive analysis requires laziness for scalability. (Experimentally, a non-lazy analysis does not scale for any non-zero context depth, i.e., cannot be effective inter-procedurally.)

Laziness means skipping an explicit representation of $\top$, in favor of keeping points-to sets empy ($\bot$ in the usual lattice of sets) as long as possible. (As mentioned earlier, at the end

[3] 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) represented by the same abstract object.

of the analysis, all sets that stayed $\bot$ become implicitly $\top$.) This has the minor benefit of avoiding storage of $\top$ values, since empty sets are represented without consuming memory. More majorly, however, 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* $\top$ *during the course of the analysis*", a non-empty 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-$\top$) contents only to later discover that the points-to set is $\top$. 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, $\top$. This means that all previously computed values for the points-to sets of `x` and `y.fld` are subsumed by the single $\top$ 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 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 $\top$.

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 standard past literature for all-program-points analyses, even a flow-*insensitive*, 2-call-site-sensitive analysis has been infeasible over these benchmarks [31].[4])

2.3 Soundness Assumptions

The soundness claims of defensive analysis are 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 and fields 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

[4] It is worth emphasizing that, although defensive analysis is lazy, this is a very different form of laziness than that of *on-demand* points-to analysis (e.g., [32, 2]). An on-demand analysis only computes points-to information for program points that may affect a particular site of interest, instead of the entire program. The defensive analysis we describe is an all-program-points analysis: it computes points-to information for the entire program, i.e., for all possible points-to queries, including ones potentially devised in the future. Yet the analysis is lazy in that it only computes values for points-to sets that it can prove to have bounded contents.

in one another, though they can contain references to each other.) A field can only be accessed via a base object pointer and a unique field signature.

- **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.[5]

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 would result in 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 may later become $\top$. 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, i.e., non-empty. 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. 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

[5] The reason for the simplified concurrency model in the paper is that it allows presenting the analysis in its purest form, dealing with core language features such as heap loads/stores and calls, but unencumbered by auxiliary considerations (e.g., computing objects that do not escape into other threads).

only if it is certain that the result could not have been invalidated by any other code, available or not. For instance, consider the following example of heap store instructions:

```
x.fld = new A();  // abstract object a1
 // x.fld points-to set is {a1}
y.fld = notFullyAnalyzed();
 // 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., undetermined.

Generally, since the analysis is flow-sensitive and 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*:

```
x.fld = new A();  // abstract object a1
 // x.fld points-to set is {a1}
x.fld = new B();  // abstract object b1
 // x.fld points-to set is {b1}
y.fld = new B();  // abstract object b2
 // 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.

The different element in defensive analysis is that if any of the involved points-to sets is empty, both strong and weak updates yield an empty points-to set. For instance, replacing either of the last two allocations (`new B()`) above with a call to opaque (or not fully analyzed) code would make all subsequent points-to sets of `x.fld` be empty.

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 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`.

```
if (complexCondition()) {
  x = new A();  // abstract object a1
} else {
  x = notFullyAnalyzed();
}
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: per the lazy evaluation principle, there is no point of computing what *one* target

of the call will do, when other targets are unknown and full soundness (i.e., guaranteed over-approximation) is required. 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 caller() {
2   A x = new A(); // abstract object a1
3   callee(x); // call to callee
4 }
5
6 void callee(A y) {
7   ...
8 }
```

Assume that there is no other discernible call to `callee` anywhere in the program. An unsound analysis would establish that variable `y` in `callee` (i.e., immediately after line 6) 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 completely unknown callers of `callee`—e.g., 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 "undetermined"—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 `3`, signifying the call-site instruction (line 3 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 [23] 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* [27, 28]: 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 [23, 30]) 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)

Figure 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, in terms of constraints. Any solution satisfying these constraints has the desired soundness property and in Section 4.3 we discuss extra considerations so that the constraints can also be used to *compute* a solution. 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.

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

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

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

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

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

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

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

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

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

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

Figure 2 Inference Rules for Defensive Points-to Analysis

The analysis constraints define the following relations:

- 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.[6]
- 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 **Init**, 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, **Init** 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.

Note the contrast between rules STORE-1 and STORE-2. We do not need to determine precisely the aliasing relationship between base variables `u` and `w`. If there is a chance that the variables are aliased, it is safe to conservatively add more possible values to the points-to set of `w.f`. In the case of STORE-1, however, we could do better than the conservative treatment and perform a strong update.

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.,

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

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.

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. This is also used to handle concurrency, under our simplified model: both `monitorenter`/`monitorexit` instructions and all accesses to `volatile` variables in the input program are represented simply as `<unknown>` instructions in our intermediate language. (The treatment of `<unknown>` collectively by the analysis rules ensures that all heap information is dropped at that program point, i.e., points-to sets are empty after the 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 $i \xrightarrow{calls}{}^{c}_{c'}$ `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

($j = \texttt{meth}(0)$) 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 $\top$: 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* [27, 28]: c' is formed by combining i and c—that is, $\mathcal{NC}(i, c, \widehat{o}) = \mathit{cons}(i, c)$. (Contexts can typically grow only up to a predetermined 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., undetermined.)

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). Such a formalization would be tedious and partial, as new capabilities are added to reflection or dynamic loading APIs with every JDK version. Instead, we establish that the analysis rules always compute over-approximate finite points-to sets (or empty sets), and that this property cannot be affected by opaque code under the common informal understanding of the 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.

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. Any non-empty set contains a superset of its dynamic contents under any possible execution. Any empty set is considered trivially "over-approximate", to avoid special-casing all our statements. In effect, the analysis produces a set of soundness markers, $Claim(P)$, which coincide with the non-empty points-to sets. No claims are made about empty points-to sets.
- Laziness: the analysis does not waste work; elements that enter a points-to set are never removed.

▶ **Theorem 1.** *There exists an evaluation order of the rules, such that the defensive analysis model is sound: all points-to sets computed are over-approximate, i.e., are either empty or contain all possible values 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 meths 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 $\mathtt{v} = \mathtt{u}$ 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 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 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 (i.e., guaranteed over-approximate and bounded), 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 $j : \mathtt{p}[arg^i_n / arg^{\mathtt{meth}}_n] \xrightarrow{\text{IN}}_{c'}$ 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 of the 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 $\forall$ iteration over call-graph edges, which contains an implication. 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 3.** *A points-to set encoded in our analysis relations grows monotonically, 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 [5]. The full implementation consists of over 400 logical rules, yet the minimal model of Section 4 captures well its essential features. We also completed a second, largely equivalent, implementation on the Soufflé Datalog engine [26]. Both implementations are publicly available at `https://bitbucket.org/yanniss/doop`.

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, the key quality metric 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? Coverage has similarly been used as a key metric in other work that infers specifications modularly [6].

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 $\forall$ 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 implemen-

tation of the `put` operation in Java `HashMaps` 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 [15, 7, 35], an *escape* analysis [4, 11], 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 (undetermined) 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. Since defensive analysis is a unique beast, it is indeed an interesting question to ask what it can be compared against. As closest comparable (though still a very dissimilar analysis) we chose to compare to a highly-precise conventional analysis with state-of-the-art best-effort soundness: a 2-object-sensitive/heap-sensitive analysis (*2objH*) with reflection support. 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 [3] 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 the two non-Android benchmarks (NTI, jFlex) from the Julia set by Nikolić and Spoto [24].

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. We use diacritical marks (* and ˆ) in the figures to remind the reader of the different analysis setting for these benchmarks.

Coverage. Figure 3 shows the coverage of defensive analysis, i.e., the number of non-empty points-to sets (for local variables) computed for all benchmarks. The input program is in SSA form, therefore the points-to sets for variables are a normalized representation of all points-to information in the program: they reflect the analysis-computed values for all program expressions, separately for each control-flow point.

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 **Init** (i.e., unconditionally).[7] It is worth emphasizing that conditional points-to guarantees (under *some* context) are valuable in a defensive analysis: they are often the best any analysis can ever do! Recall our earlier discussion of Section 3: many of the useful inferences of a defensive analysis will be under *some* context even when the inference holds under *all known* contexts in existing code. No analysis can preclude the existence of other callers in opaque, and possibly not yet existing, code. Such callers can arise in dynamically generated code and can invoke existing methods, e.g., using reflection.

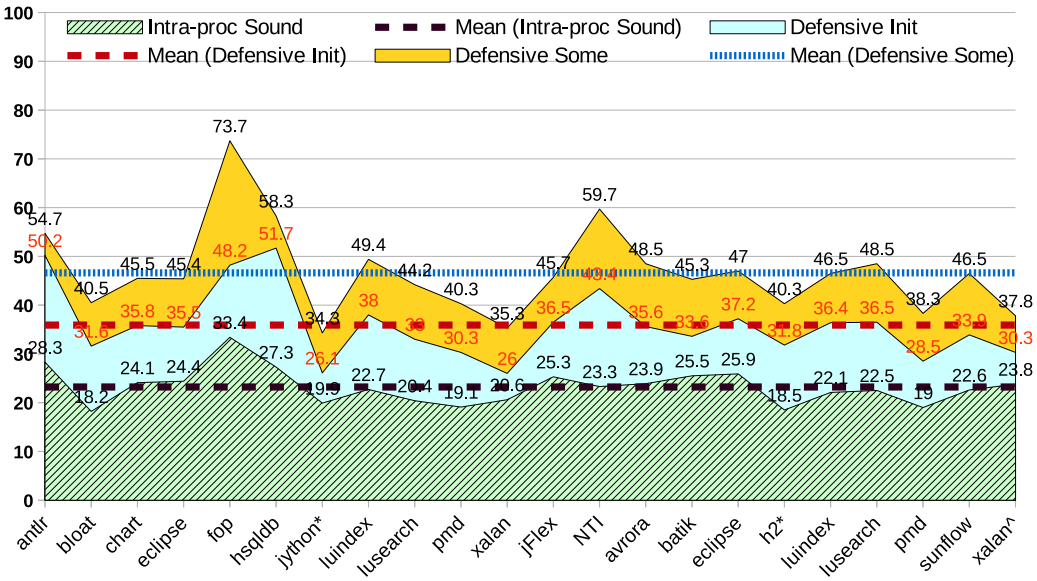

Figure 3 Percentage of application variables (deemed reachable by baseline 2objH analysis) that have non-empty points-to sets for defensive analysis under some context and Init context (no assumptions). Intra-procedural sound points-to analysis (defensive minus the complex cases) shown as baseline. Arithmetic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

[7] If a variable has a points-to value for context **Init**, then it also has that value under every specific context that arises for the variable. Therefore, points-to sizes for **Init** are always lower than conditional, context-specific sizes.

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.

Comparison with intra-procedural. We have earlier referred to the "easy", intra-procedural parts of the analysis reasoning: what a compiler or VM would likely do to perform sound local data-flow analysis. This is the subject of **RQ2**, also answered by Figure 3. The figure includes 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 is otherwise the same as the full "defensive" logic, with the exception of the new "interesting" cases (control-flow merging, heap manipulation, and inter-procedural propagation).

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.) Although the benefit is not broken down further in the figure, the handling of method calls alone (i.e., rules Call, Args and Ret) is responsible for the lion's share of the difference between the full defensive analysis and the intra-procedural sound analysis.

Running time. Figure 4 shows the running times of the analysis, plotted next to that of 2objH, for reference. Although the two analyses are dissimilar, 2objH is qualitatively the closest one can get to defensive analysis with the current state of the art: it is an analysis with high precision, run with best-effort soundness support. Therefore, 2objH can serve as a realistic point of reference. As can be seen, the running times of defensive analysis are realistically low, although its flow-sensitive and 5-call-site-sensitive nature suggests it would be a prohibitively heavy analysis. 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.

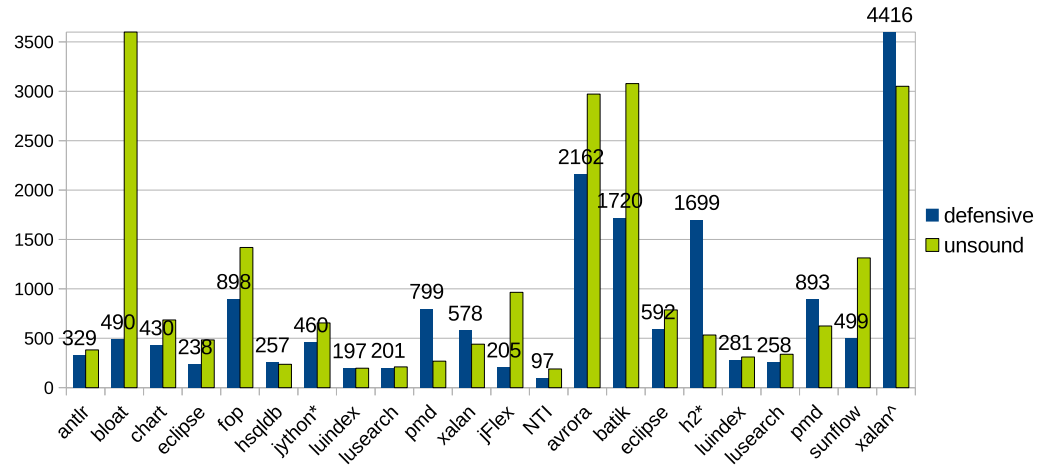

Figure 4 Running time (sec) of defensive analysis, with running time of 2objH (with unsound reflection handling) shown as a baseline. Labels are shown for defensive analysis only to avoid crowding the plot.

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 and a compiler cannot act upon them. For optimization clients, such as devirtualization, soundness is essential. Using sound results, a JIT compiler can skip dynamic tests (of the inline caching optimization) for all calls that the analysis soundly covers.

Figure 5 shows the virtual calls that defensive analysis devirtualizes, as a percentage of those devirtualized by the unsound analysis.

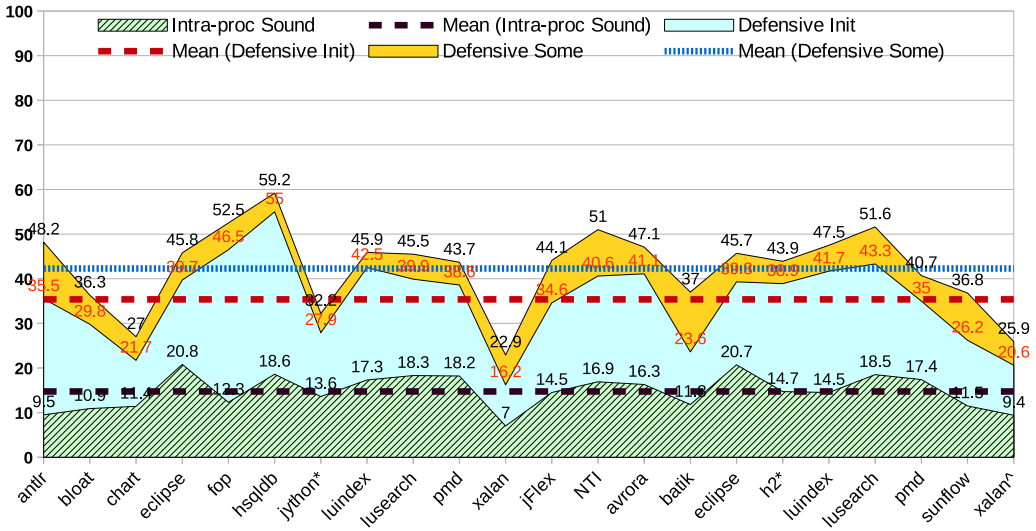

Figure 5 Virtual call sites that are found to have receiver objects of a single type. These call sites can be soundly devirtualized. Numbers are shown as percentages of devirtualization achieved by unsound 2objH analysis.

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, **Init** 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 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.

Points-to set sizes. Finally, it is interesting to quantify the precision of the defensive analysis, for the points-to sets it covers. This precision is expected to be high, since defensive analysis is flow- and context-sensitive, but exact figures help put it in perspective.

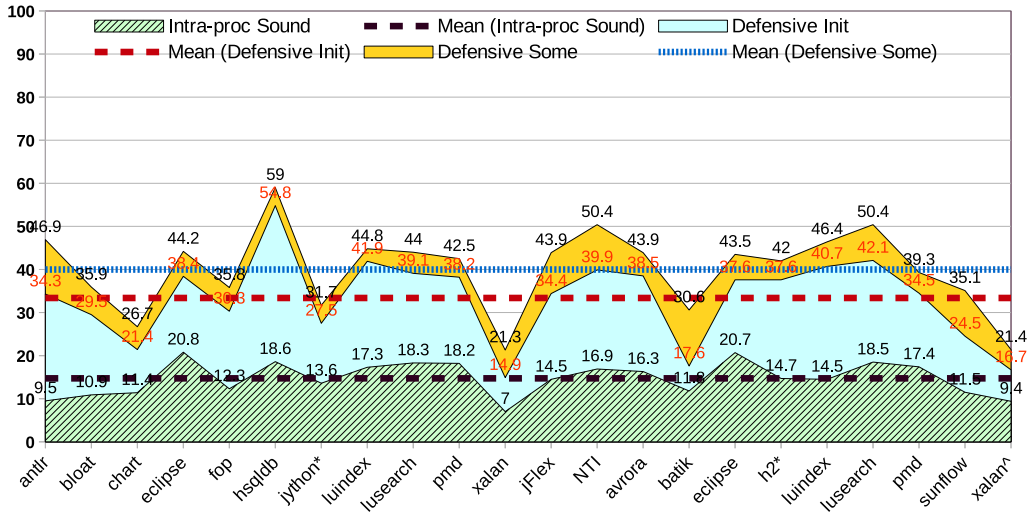

Figure 6 Virtual call sites (percentage of 2objH) that are found to have receiver objects of a single type. Updates Figure 5, this time with soundness under a relaxed memory model.

Figure 7 shows average points-to set sizes for the defensive analysis vs. the 2objH analysis. The sets (excluding null values) are computed over variables covered by both analyses, for non-empty defensive analysis sets and under context **Init** of the defensive analysis, i.e., unconditionally. (The numbers are for the simplistic concurrency model, but remain unchanged to two significant digits for the relaxed concurrency model.)

	Benchmark	Average points-to over same vars defensive	2objH
DaCapo 2006-10-MR2	antlr	1.01	1.10
	bloat	1.02	2.12
	chart	1.09	1.09
	eclipse	1.06	1.31
	fop	1.00	1.03
	hsqldb	1.01	1.04
	jython*	1.01	6.05
	luindex	1.02	1.02
	lusearch	1.04	1.06
	pmd	1.01	1.05
	xalan	1.05	1.12
	jFlex	1.01	1.02
	NTI	1.03	1.03
DaCapo 9.12-Bach	avrora	1.05	3.04
	batik	1.04	1.05
	eclipse	1.07	1.53
	h2*	1.04	2.07
	luindex	1.01	1.04
	lusearch	1.03	1.08
	pmd	1.01	1.04
	sunflow	1.05	1.08
	xalan^	1.04	1.19
	mean	1.03	1.51

Figure 7 Average number of abstract objects per variable, for variables for which both analyses compute results.

As can be seen, the defensive analysis is highly precise when it produces non-empty points-to sets, typically yielding points-to set sizes very close to 1. 2objH is also a very

precise analysis (for variables with bounded points-to sets), so it remains competitive, yet clearly less precise. Notably, points-to set sizes close to 1 are the Holy Grail of points-to analysis: such precision is actionable for nearly all conceivable clients of a points-to analysis.

7 Related Work

There is certainly past work that attempt to ensure a sound whole-program analysis, but none matches the generality and applicability of our approach. We selectively discuss representative approaches.

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 [8] analyses, such as Astrée [10], 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. [13, 14] 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. [17] 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 [34], necessary for simultaneously computing inter-related points-to, may-alias, and may-escape information.

Sreedhar et al. [33] 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)* [9] or *Rapid Type Analysis (RTA)* [1]) 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, even this is not sufficient for a sound static analysis of opaque code: classes can be generated and loaded dynamically during program execution. CHA cannot find target methods that do not even exist statically, yet modeling them is precisely what is needed for soundness in real-world conditions. For instance, Java applications, especially

in the enterprise (server-side) space, employ dynamic loading heavily, and patterns such as *dynamic proxies* have been standardized and used widely since the early Java days.

Furthermore, such heuristic "best-effort" over-approximation is detrimental to analysis precision and performance. CHA is an example of a loose over-approximation in an effort to capture most dynamic behaviors. Loose over-approximations compute many more possible targets than those that realistically arise. This yields vast points-to sets that render the analysis heavyweight and useless due to imprecision. (Avoiding such costs is exactly why past analyses have often opted for glaringly unsound handling of opaque code features.) Our lazy representation of "don't know"/"cannot bound" values as empty sets addresses the problem, by keeping all points-to sets compact.

The conventional handling of reflection in may-point-to analysis algorithms for Java [12, 18, 22, 20, 29, 19] 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 (which accepts a name string and returns a reflection object representing the class with that name). 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 [12],[8] and enabling reflection on the DOOP framework slows it down by an order of magnitude on standard benchmarks [29]. 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

1 David F. Bacon and Peter F. Sweeney. 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, pages 324–341, New York, NY, USA, 1996. ACM.

[8] The WALA documentation is explicit: "*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.*" [12]

2 Sandip K. Biswas. A demand-driven set-based analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, 1997.

3 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. 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, pages 169–190, New York, NY, USA, 2006. ACM. doi:10.1145/1167473.1167488.

4 Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, 1998.

5 Martin Bravenboer and Yannis Smaragdakis. 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, New York, NY, USA, 2009. ACM.

6 Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 289–300, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1480881.1480917, doi:10.1145/1480881.1480917.

7 Jong D. Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, 1993.

8 Patrick Cousot and Radhia Cousot. 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, pages 238–252, New York, NY, USA, 1977. ACM. URL: http://doi.acm.org/10.1145/512950.512973, doi:http://doi.acm.org/10.1145/512950.512973.

9 Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of the 9th European Conf. on Object-Oriented Programming*, ECOOP '95, pages 77–101. Springer, 1995. doi:10.1007/3-540-49538-X_5.

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

11 Alain Deutsch. On the complexity of escape analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 358–371, New York, NY, USA, 1997. ACM. URL: http://doi.acm.org/10.1145/263699.263750, doi:10.1145/263699.263750.

12 Stephen J. Fink et al. WALA UserGuide: PointerAnalysis. http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis, 2013.

13 Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *Proc. of the 18th European Conf. on Object-Oriented Programming*, ECOOP '04, pages 96–122. Springer, 2004. doi:10.1007/978-3-540-24851-4_5.

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

15 Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 329–341, 1998.

16 Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of Java reflection – literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017.

17 Chris Lattner, Andrew Lenharth, and Vikram Adve. 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, New York, NY, USA, 2007. ACM.

18 Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for Java. In *Proc. of the 28th European Conf. on Object-Oriented Programming*, ECOOP '14, pages 27–53. Springer, 2014.

19 Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 162–180. Springer, 2015. URL: http://dx.doi.org/10.1007/978-3-662-48288-9_10, doi:10.1007/978-3-662-48288-9_10.

20 Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, December 2006.

21 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. URL: http://doi.acm.org/10.1145/2644805, doi:10.1145/2644805.

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

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

24 Durica Nikolić and Fausto Spoto. Definite expression aliasing analysis for Java bytecode. In *Proc. of the 9th International Colloquium on Theoretical Aspects of Computing*, volume 7521 of *ICTAC '12*, pages 74–89. Springer, 2012. doi:10.1007/978-3-642-32943-2_6.

25 Xavier Rival. Comment on "what is soundness in static analysis" post, in the PL Enthusiast blog. http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/#comment-1265, November 2017.

26 Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 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*, pages 196–206, 2016. URL: http://doi.acm.org/10.1145/2892208.2892226, doi:10.1145/2892208.2892226.

27 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program flow analysis: theory and applications*, chapter 7, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

28 Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, may 1991.

29 Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of Java reflection. In *Proc. of the Asian Symp. on Programming Languages and Systems*, APLAS '15. Springer, 2015.

30 Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

31 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594320.

32 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 22:1–22:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2016.22, doi:10.4230/LIPIcs.ECOOP.2016.22.

33 Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. 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, pages 196–207, New York, NY, USA, 2000. ACM.

34 Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

35 Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *Proc. of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM. doi:10.1145/1328438.1328464.