QC
QUIRK CLASSICS

PRIDE AND PREJUDICE AND ZOMBIES

BY JANE AUSTEN AND SETH GRAHAME-SMITH

Declarative Static Analysis and Zombies

Yannis Smaragdakis
University of Athens
with
Martin Bravenboer,
George Kastrinis, George Balatsouras,
Tony Antoniadis, George Fourtounis, Neville Grech
and
Kostas Ferles, Nikos Filippakis,
Sifis Lagouvardos, Yue Li, Petros Pathoulas,
Kostas Saidis, Tian Tan, Konstantinos Triantafyllou

Declarative Static Analysis and Soundness

Yannis Smaragdakis
University of Athens
with
Martin Bravenboer,
George Kastrinis, George Balatsouras,
Tony Antoniadis, George Fourtounis, Neville Grech
and
Kostas Ferles, Nikos Filippakis,
Sifis Lagouvardos, Yue Li, Petros Pathoulas,
Kostas Saidis, Tian Tan, Konstantinos Triantafyllou

European Research Council
Established by the European Commission

European Union
European Social Fund
OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS, CULTURE & SPORTS
MANAGING AUTHORITY
NSRF 2007-2013
programme for development
EUROPEAN SOCIAL FUND
Co-financed by Greece and the European Union

LOGICBLOX

Overview

- What do we do?
  - static program analysis
    - “discover program properties that hold for all executions”
- Vision: a system that knows more about your program than you do
- How do we do it?
  - declarative (logic-based specification)
    - fast, powerful, new insights

Our Research: Doop
and friends: CClyzer, MadMax

- Since 2008:
  - Doop: a powerful framework for analyzing Java bytecode
    - building on *pointer analysis*
      - now just a substrate for more analyses
  - declarative, using the Datalog language
- Lots of offshoots
  - Cclyzer, for LLVM bitcode
  - MadMax/Gigahorse for Ethereum VM bytecode [OOPSLA'18 Distinguished Paper Award]
    - 38MLoC in 8 hours

Pointer Analysis: A Complex Domain

- flow-sensitive
- field-sensitive
- heap cloning
- context-sensitive
- binary decision diagrams
- inclusion-based
- unification-based
- on-the-fly call graph
- k-cfa
- object sensitive
- field-based
- demand-driven

Results 1 - 20 of 2,343

Sort by relevance in expanded form

Save results to a Binder

Result page: **1** 2 3 4 5 6 7 8 9 10 next >>

1 Semi-sparse flow-sensitive pointer analysis
January 2009 **POPL '09:** Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages
Publisher: ACM
Full text available: Pdf (246.09 KB) Additional Information: full citation, abstract, references, index terms
Bibliometrics: Downloads (6 Weeks): 34, Downloads (12 Months): 34, Citation Count: 0

Pointer analysis is a prerequisite for many program analyses, and the effectiveness of these analyses depends on the precision of the pointer information they receive. Two major axes of pointer analysis precision are flow-sensitivity and context-sensitivity, ...

Keywords: alias analysis, pointer analysis

2 Efficient field-sensitive pointer analysis of C
David J. Pearce, Paul H.J. Kelly, Chris Hankin
November 2007 **Transactions on Programming Languages and Systems (TOPLAS)**, Volume 30 Issue 1
Publisher: ACM
Full text available: Pdf (924.64 KB) Additional Information: full citation, abstract, references, index terms
Bibliometrics: Downloads (6 Weeks): 31, Downloads (12 Months): 282, Citation Count: 1

The subject of this article is flow- and context-insensitive pointer analysis. We present a novel approach for precisely modelling struct variables and indirect function calls. Our method emphasises efficiency and simplicity and is based on a simple ...

Keywords: Set-constraints, pointer analysis

3 Cloning-based context-sensitive pointer alias analysis using binary decision diagrams
John Whaley, Monica S. Lam
June 2004 **PLDI '04:** Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation
Publisher: ACM

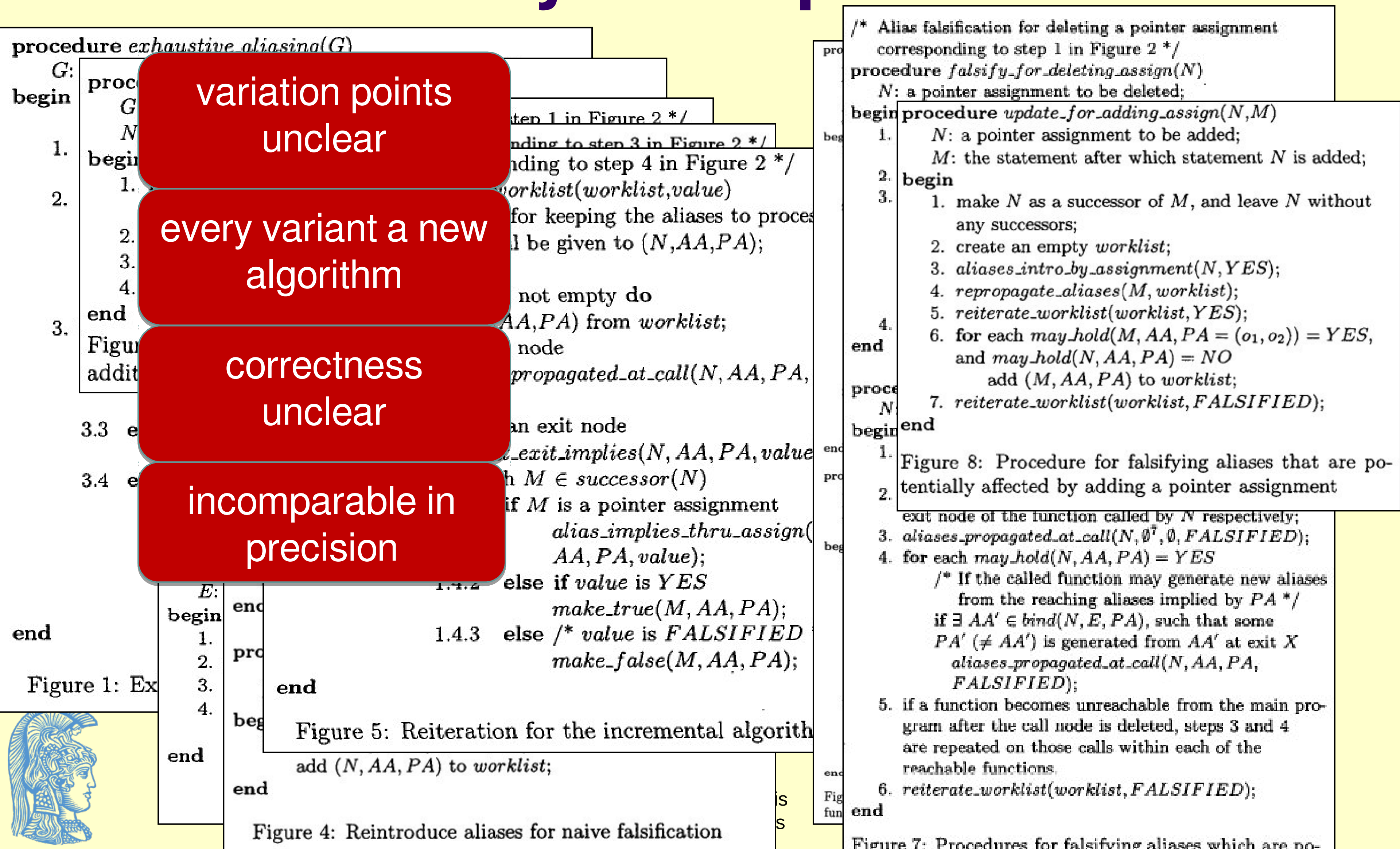
Algorithms Found In a 10-Page Pointer Analysis Paper

variation points unclear

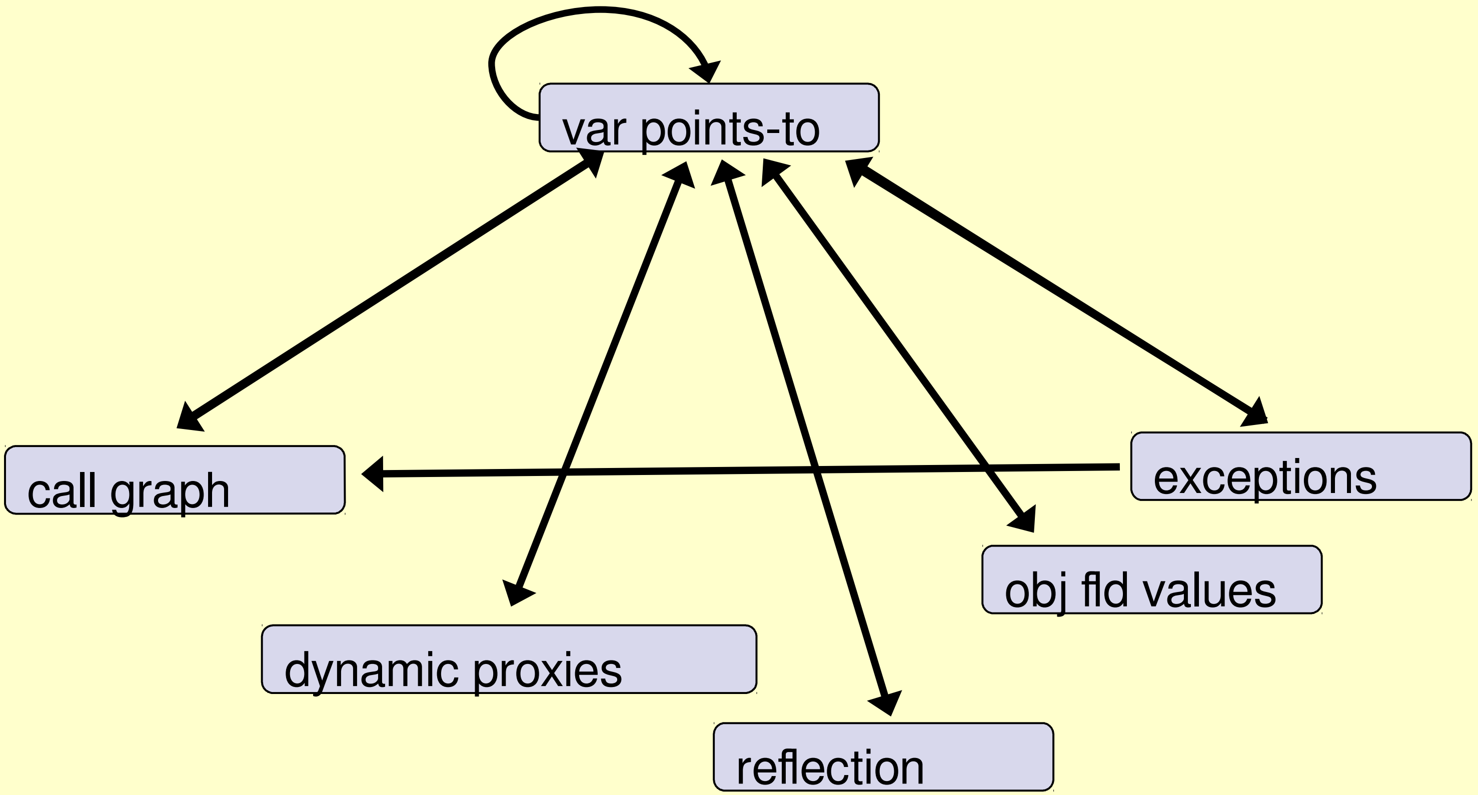
every variant a new algorithm

correctness unclear

incomparable in precision



Program Analysis: a Domain of Mutual Recursion



Holistic Program Analysis: "Everything Is Connected"

A Vision Within Reach

- *An intelligent system that knows more about your program than you do*
- “Everything is connected”
  - all analysis aspects encoded separately, all benefitting each other
- The Doop framework serves to illustrate
- Key: a declarative specification of all sorts of static analyses
- In Doop: use of Datalog

Datalog To The Rescue!

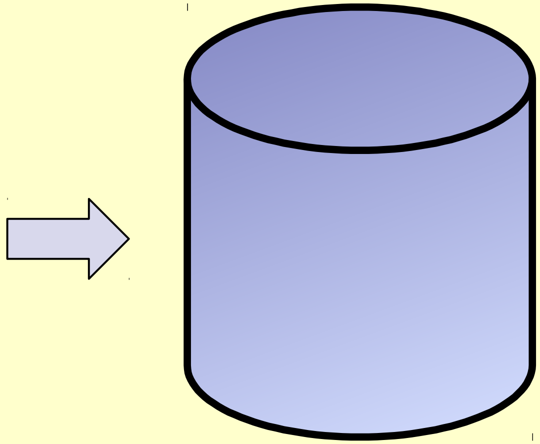
- Datalog is relations + recursion
- Limited logic programming
  - SQL with recursion
  - Prolog without complex terms (constructors)
- Captures PTIME complexity class
- Strictly declarative
  - e.g., as opposed to Prolog
    - conjunction commutative
    - rules commutative
  - monotonic

Less programming, more specification

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```



Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

rules

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

head

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

head relation

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

bodies

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

body relations

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

join variable

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

recursion

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

a	new A()
b	new B()
c	new C()

1st rule result

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source
`a = new A();`
`b = new B();`
`c = new C();`
`a = b;`
`b = a;`
`c = b;`

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

a	new A()
b	new B()
c	new C()

2nd rule evaluation

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

a	new A()
b	new B()
c	new C()
a	new B()

2nd rule result

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

a	new A()
b	new B()
c	new C()
a	new B()
b	new A()
c	new B()
c	new A()

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

The Doop Framework

- Datalog-based static analysis framework for Java
- Declarative: what, not how
- Sophisticated, very rich set of analyses
  - subset-based analysis, fully on-the-fly call graph discovery, field-sensitivity, context-sensitivity, call-site sensitive, object sensitive, thread sensitive, context-sensitive heap, abstraction, type filtering, precise exception analysis
- Support for full semantic complexity of Java
  - jvm initialization, reflection analysis, threads, reference queues, native methods, class initialization, finalization, cast checking, assignment compatibility

http://doop.program-analysis.org

Past Approaches and Declarative Analysis

- Past approaches have flirted with declarative analysis
- But no purely declarative approach
  - specification and algorithm confused
- Declarativeness considered unscalable in both complexity and performance
  - "*the first time I write an analysis it is typically in Datalog, but then, once I'm convinced it's precise, I throw it out and I write it in Java, when I want to focus on scalability.*" *(Naik, 2010)*

Doop Makes Declarative Analysis Real

- Complete, complex pointer analyses in Datalog
  - core specification: ~1500 logic rules
  - parameterized by a handful of rules per analysis flavor
- Efficient algorithms from specification
  - order of magnitude performance improvement
  - allowed to explore more analyses than past literature
- Approach: heuristics for searching algorithm space
  - targeted at recursive problem domains
- Demonstrated scalability with explicit representation
  - no BDDs

Not Expected

- Expressed complete, complex pointer analyses in Datalog

  "*[E]ncoding all the details of a complicated program analysis problem [on-the-fly call graph construction, handling of Java features] purely in terms of subset constraints may be difficult or impossible.*" *(Lhotak)*

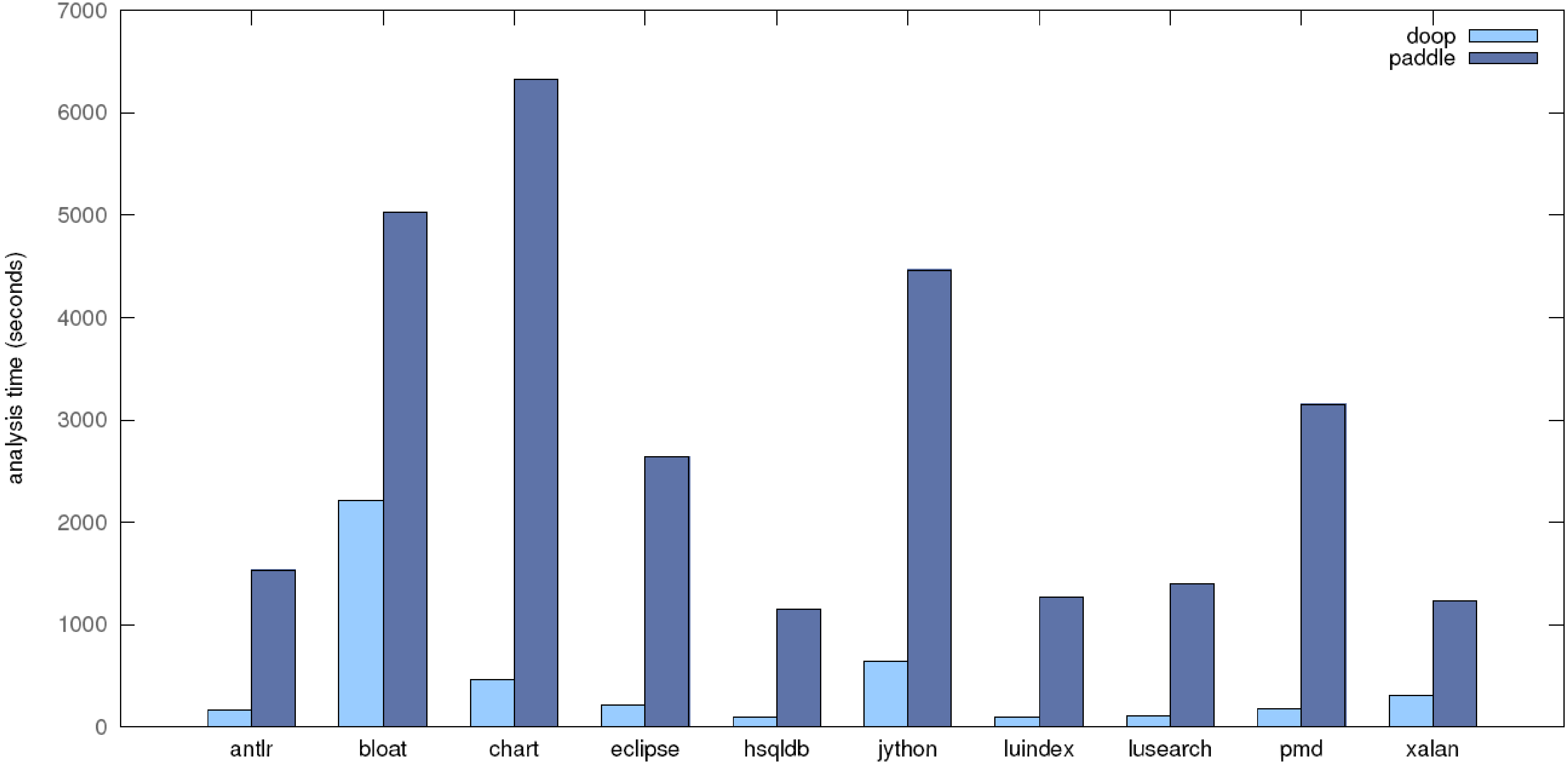
- Scalability and Efficiency

  "Efficiently implementing a 1H-object-sensitive analysis without BDDs will require new improvements in data structures and algorithms"

Impressive Performance, Implementation Insights

[OOPSLA'09, ISSTA'09]

Large Speedup For Realistic Analyses



Better Understanding of Existing Algorithms, More Precise and Scalable New Algorithms

[PLDI'10, POPL'11, CC'13, PLDI'13, PLDI'14, FSE'18, OOPSLA'18]

Many More Work Threads

- Set-based pre-analysis [OOPSLA’13]
  - universal optimization technique
- Completing a partial program [OOPSLA’13]
  - making sense out of missing libraries
- Soundness [CACM 2/15, ECOOP'18 **(distinguished paper)**]
- Reflection and dynamic loading [APLAS'15, ECOOP'18, ISSTA'18]
- Port to Souffle: a parallel Datalog engine [SOAP’17]
- Must-alias analysis [SOAP’17, CC'18]
- Taint analysis using points-to algorithms [OOPSLA’17]
- Integrating heap snapshots in static analysis [OOPSLA’17, ISSTA'18]

Now Zombies
(ahem, soundness)

Soundness in Static Analysis

- We all want it!
- ***Sound*: *AnalysisClaim(P)* → *P***
- E.g., for a (*may-*) value-flow analysis: is *every* possible run-time value modeled statically?
- Soundness is a design property of an analysis
  - often broken up by language feature
    - basically “do you fully handle this feature?”
  - e.g., “do you handle arrays soundly?”

Soundiness Manifesto [CACM 2/15]

- *"There is **no** practical static whole-program may-analysis that is sound"*
  - whole-program: models the heap
- What about all these soundness proofs?
  - proof is for a limited language
  - unsoundness due to dynamic features: reflection, dynamic loading, eval

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

This Work: [ECOOP'18, Distinguished Paper Award]
Truly Sound Analysis, for Full Language

- Key elements:
  - I. different form of soundness theorem
  - II. **defensive** design that withstands *opaque* code
    - i.e., code that could be doing (nearly) anything
  - III. laziness necessary for a realistic implementation

Part I. Motivation: Different Form of Soundness Theorem

Conventional Soundness Theorem (formulation by Xavier Rival)

- *for all programs in stated language subset and all executions in stated exec. subset*
  AnalysisClaim(P) → P

- Soundness is always qualified
- Problem: qualifications don't hold in practice
  - realistic programs use dynamic features

Even Worse: Perverse Incentives!

- *for all programs in stated language subset and all executions in stated exec. subset*
  AnalysisClaim(P) $\rightarrow$ ***P***

- Proof starts from formulation of analysis over input language
- Weaker analysis, easier soundness theorem!
  - vastly unsound analysis: easy soundness proof

Our Soundness Theorem Form

- *for all program points, π, in* **computed** *subset,* ***AnalysisClaim**$_\pi$**(P)** → **P**$_\pi$*

- The analysis works for (nearly) all language features, all executions
  - but qualifies which part of its results is guaranteed sound!

Our Soundness Theorem Form

- *for all program points, π, in* **computed** *subset,* ***$AnalysisClaim_{\pi}(P) \rightarrow P_{\pi}$***

- Important concept: *coverage*
  - how big is the subset of the program for which the analysis is sound

Part II. Approach: Defensive Design that Withstands *Opaque* Code

General Form of Sound Points-To Analysis

- Sound points-to information: need to compute ***all*** possible values that may ***ever*** arise at run time
- For the analysis to certify points-to set as sound, it needs to:
  - closely track information <u>all the way</u> from its source
  - ensure no possible interference
- Need precise analysis:
  - context-sensitive, flow-sensitive, over access paths

When Can We Be Sound? Hello-World Case

program

```
void foo() {
  Object a = new A1();
  Object b = id(a);
}

void bar() {
  Object a = new A2();
  Object b = id(a);
}

Object id(Object a) {
  return a;
}
```

points-to

a	new A1()
a	new A2()
a	???

When Can We Be Sound? Hello-World Case

program

```
void foo() {
  Object a = new A1();
  Object b = id(a);
}

void bar() {
  Object a = new A2();
  Object b = id(a);
}

Object id(Object a) {
  return a;
}
```

points-to

```
a        new A1()

a        new A2()

a (foo) new A1()  +
a (bar) new A2()
```

When Can We Be Sound? Hello-World Case

program

```
void foo() {
  Object a = new A1();
  Object b = id(a);
}

void bar() {
  Object a = new A2();
  Object b = id(a);
}

Object id(Object a) {
  return a;
}
```

points-to

```
a         new A1()
b         new A1()

a         new A2()
b         new A2()

a (foo) new A1()  +
a (bar) new A2()
```

More Illustration

- What do we know after this statement?

```
x.fld = new A();   // abstract object a1
```

  - that program expression `x.fld` refers to `a1`
  - regardless of what x refers to
    - access paths!
  - also that any `z.fld` needs to be augmented

- ... if followed by:

```
...   // analyzable code, no interference
y = x.fld;
```

we know y also refers to `a1`

Defensiveness Examples

- When the analysis is uncertain, it has to refuse to certify the soundness of a points-to set

```
if (P()) {
   x.fld = new A();   // abstract object a1
} else {
   x.foo();           // opaque
}
```

- `x.fld` has an unknown points-to set after `if`
  - `x.foo()` could invoke dynamic code, do reflection, or merely be too complex to analyze precisely
    - e.g., reach maximum context-sensitivity depth

Method Calls

- Let's analyze the example further:
  when is a call **not** opaque code?

```
if (P()) {
   x.fld = new A();   // abstract object a1
} else {
   x.foo();           // opaque
}
```

  - `x` has known points-to set (i.e., known `foo`)
  - all possible `foo` do not perform opaque actions on an access path
- Involved topic, more in the paper

Part III. Technique: Laziness for Realistic Implementation

Laziness

- A flow-sensitive, context-sensitive algorithm over access paths cannot scale
- Idea: compute points-to set only when we can prove the set is sound
- Implication: an *empty* set means *unbounded*
  - the analysis could not compute all its possible contents

Laziness, Concretely

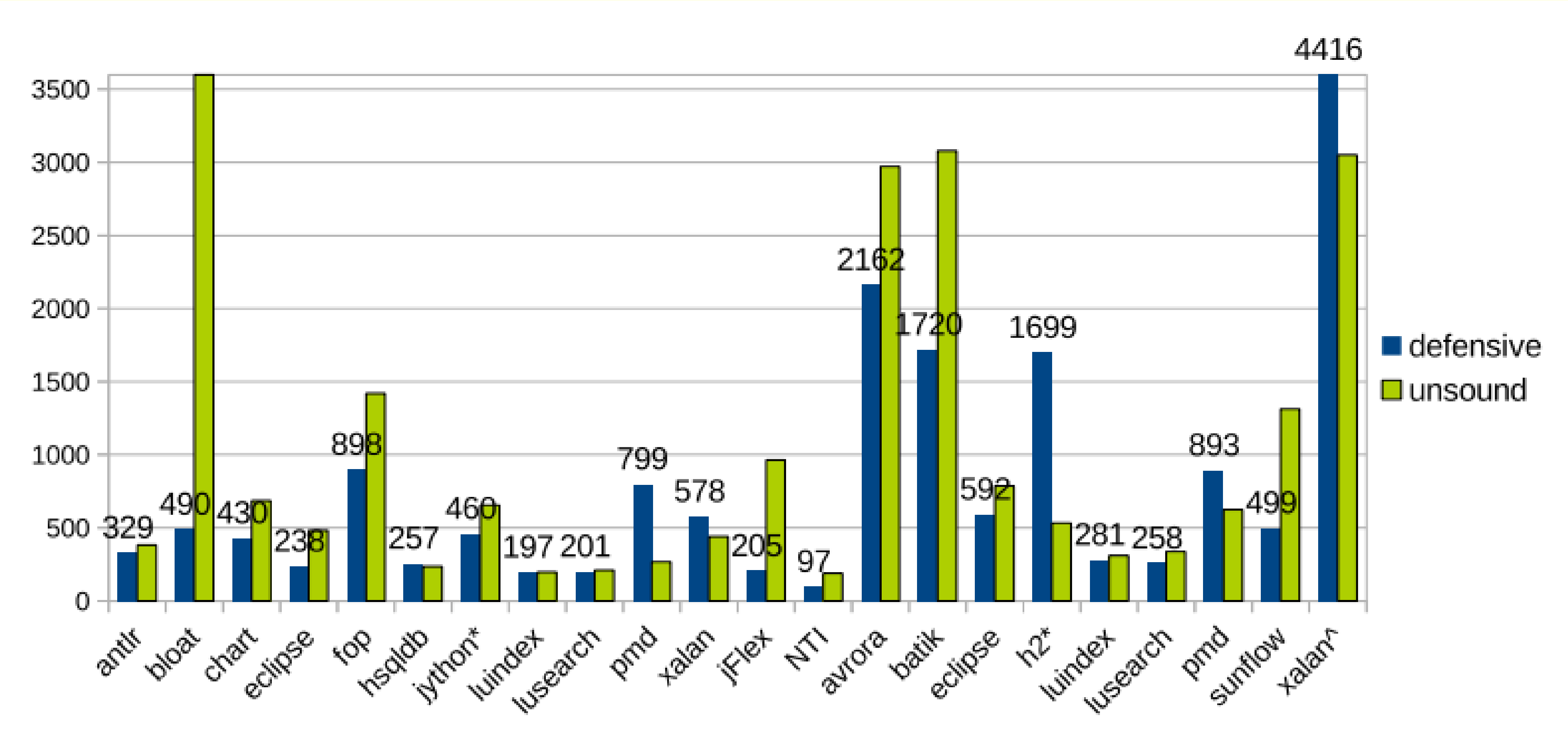
- All points-to sets start empty
- Only compute a points-to set (i.e., make it non-empty) when
  - all other points-to sets feeding into it are known
  - and are non-empty themselves
- Any points-to set that remains empty at end of analysis is marked $\top$

Laziness Benefits

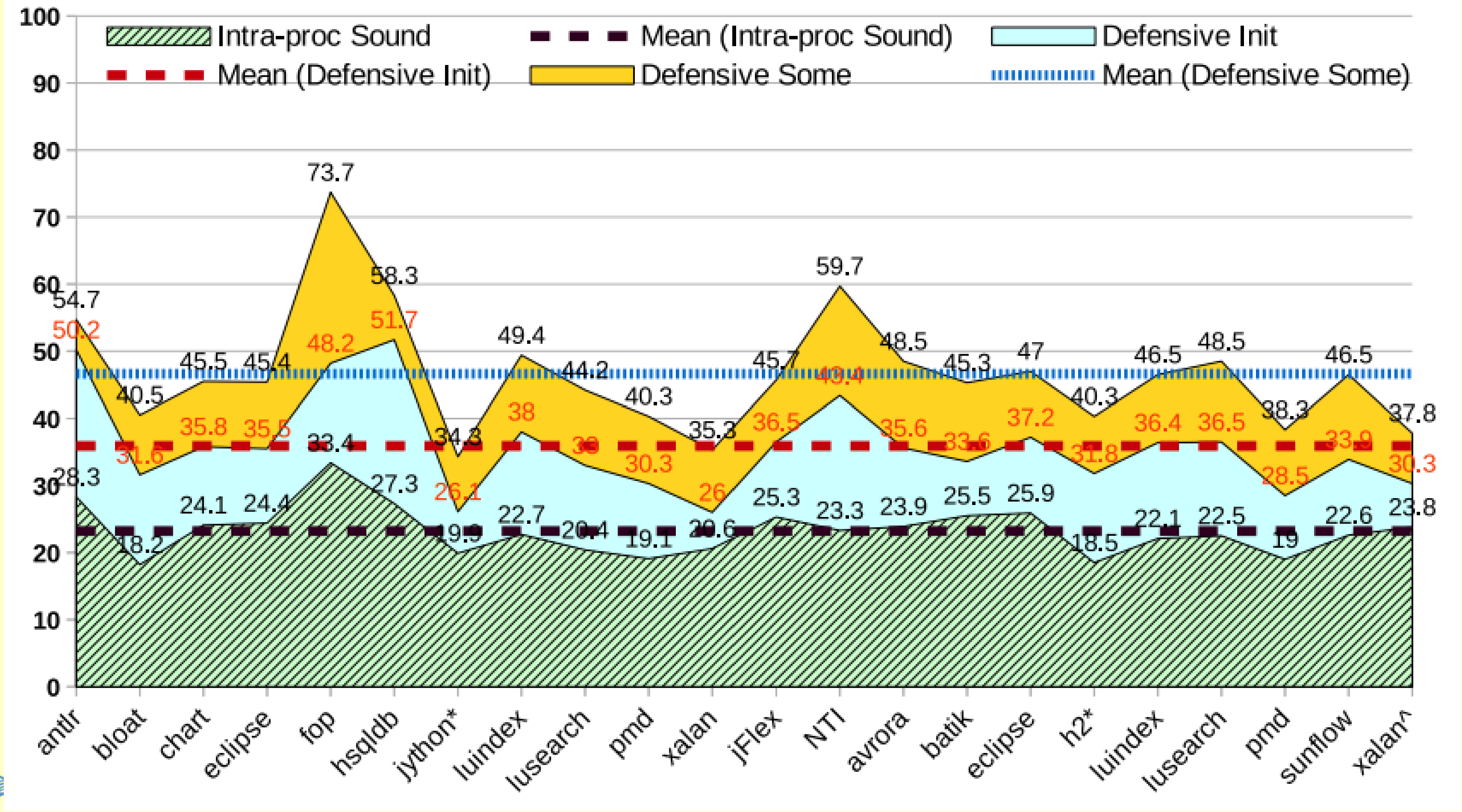
- Scalable analysis
- Avoids wasted work! Never compute a points-to set, only to have the addition of more information make its contents non-certifiably sound!
  - i.e., $\top$

Evaluation Results

Running Time

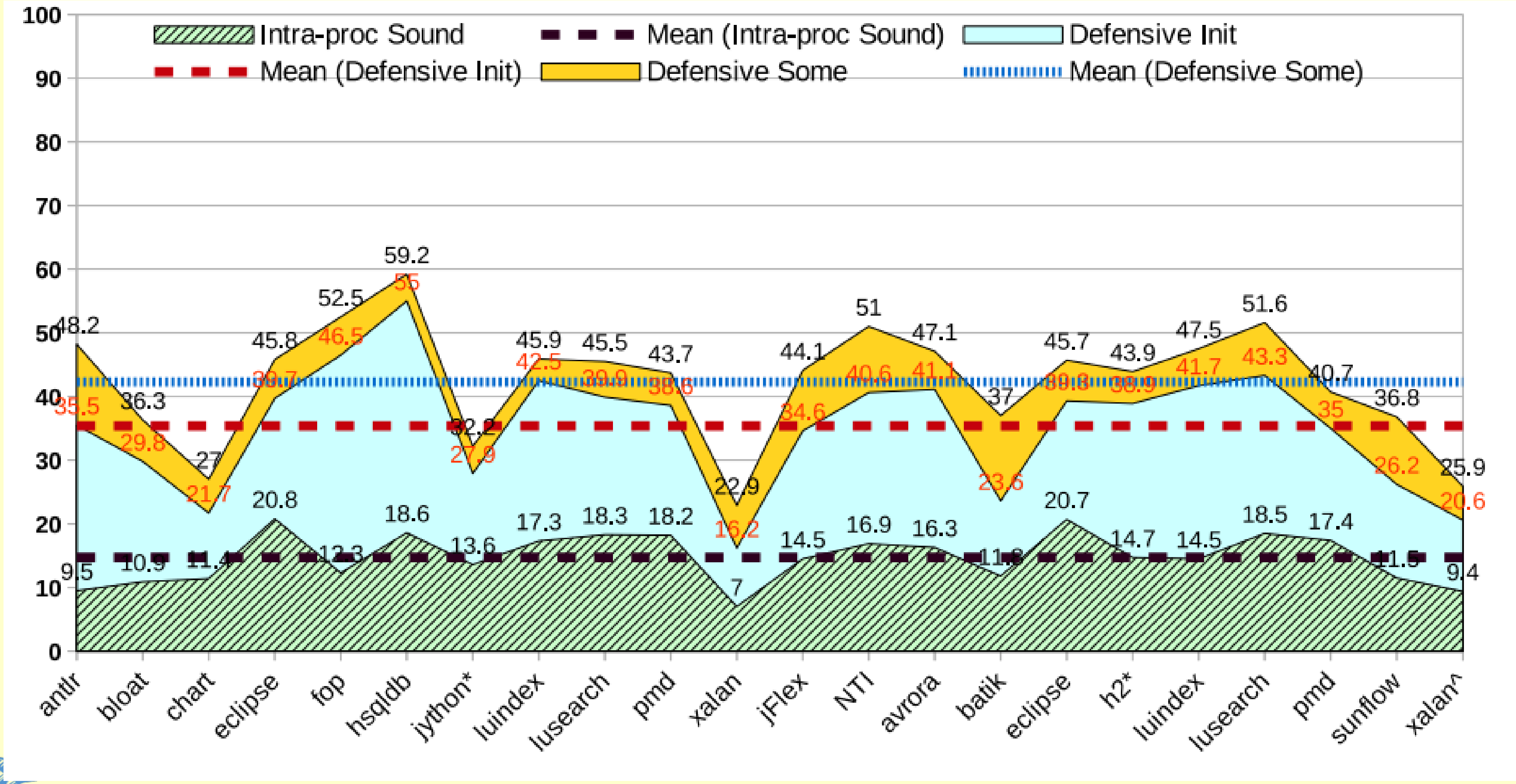


Coverage



Intra-proc Sound
Mean (Intra-proc Sound)
Defensive Init
Mean (Defensive Init)
Defensive Some
Mean (Defensive Some)
100
90
80
70
60
50
40
30
20
10
0
antlr
bloat
chart
eclipse
fop
hsqldb
jython*
luindex
lusearch
pmd
xalan
jFlex
NTI
avrora
batik
eclipse
h2*
luindex
lusearch
pmd
sunflow
xalan^
54.7
50.2
28.3
40.5
31.6
18.2
45.5
35.8
24.1
45.4
35.5
24.4
73.7
48.2
33.4
58.3
51.7
27.3
34.3
26.1
19.9
49.4
38
22.7
44.2
33
20.4
40.3
30.3
19.1
35.3
26
20.6
45.7
36.5
25.3
59.7
43.4
23.3
48.5
35.6
23.9
45.3
33.6
25.5
47
37.2
25.9
40.3
31.8
18.5
46.5
36.4
22.1
48.5
36.5
22.5
38.3
28.5
19
46.5
33.9
22.6
37.8
30.3
23.8

Devirtualization Client



Conclusions

- Doop: early instance of intelligent system that just *knows* things about your program
- Also: fully sound analysis, for realistic languages, is possible!
- Different form of soundness theorem, *coverage* as important concept