Samsung Research Funding &
Incubation Center for Future Technology

Data-Driven Program Analysis

Hakjoo Oh

Korea University

12 February 2019 @Furiosa AI

PL/SE Research @KU

- Research areas: programming languages, software engineering, software security
  - program analysis and testing
  - program synthesis and repair
- Publication: top-venues in PL, SE, Security, and AI:
  - PLDI('12,'14), OOPSLA('15,'17,'17,'18,'18), TOPLAS('14,'16,'17,'18,'19), ICSE('17,'18,'19), FSE'18, ASE'18, S&P'17, IJCAI('17,'18), etc

http://prl.korea.ac.kr

Heuristics in Program Analysis

- Practical program analysis tools use many heuristics
  - E.g., context/flow-sensitivity, variable clustering, unsoundness, trace partitioning, path selection/pruning, state merging, etc
- Developing a good heuristic is an art
  - Empirically done by analysis designers: nontrivial & suboptimal

Vision

Automatically Generating Analysis Heuristics from Data

- Use data to make empirical decisions in program analysis

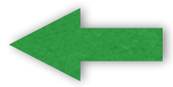
machine learning

context-sensitivity heuristics
flow-sensitivity heuristics
unsoundness heuristics
path-selection heuristics
…

- Automatic: little reliance on analysis designers
- Powerful: machine-tuning outperforms hand-tuning
- Stable: can be tuned for target programs

Example: Context-Sensitivity

```
    int h(n) {ret n;}

    void f(a) {
c1:   x = h(a);
      assert(x > 0);  // Query
c2:   y = h(input());
    }

c3: void g() {f(8);}

    void m() {
c4:   f(4);
c5:   g();
c6:   g();
    }
```

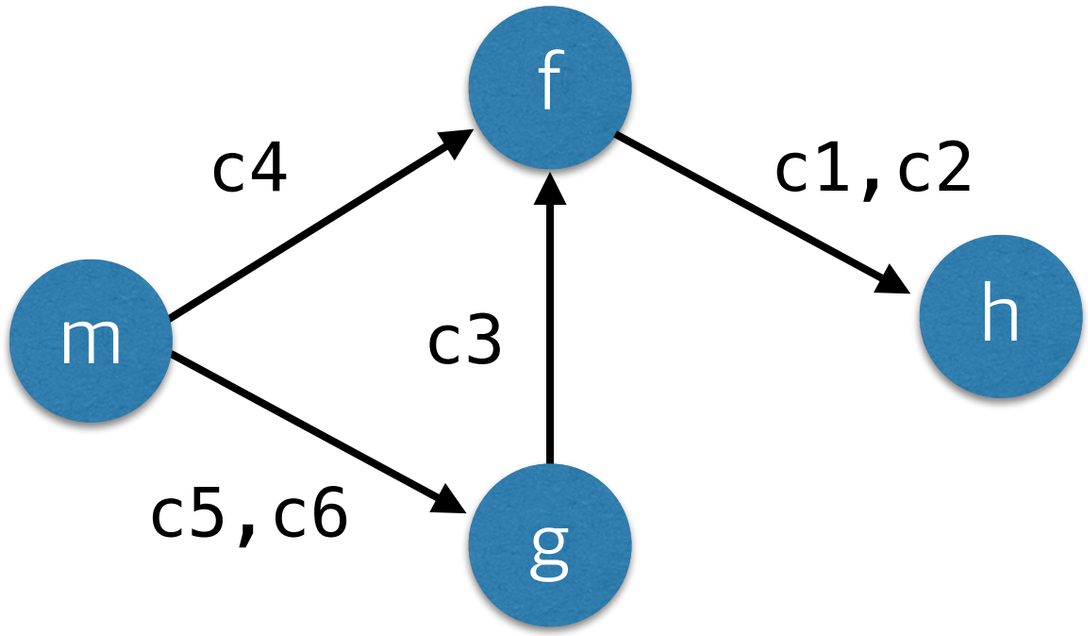


holds always

Context-Insensitive Analysis

- Merge calling contexts into single abstract context

```
    int h(n) {ret n;}

    void f(a) {
c1:   x = h(a);
      assert(x > 0);
c2:   y = h(input());
    }

c3: void g() {f(8);}

    void m() {
c4:   f(4);
c5:   g();
c6:   g();
    }
```

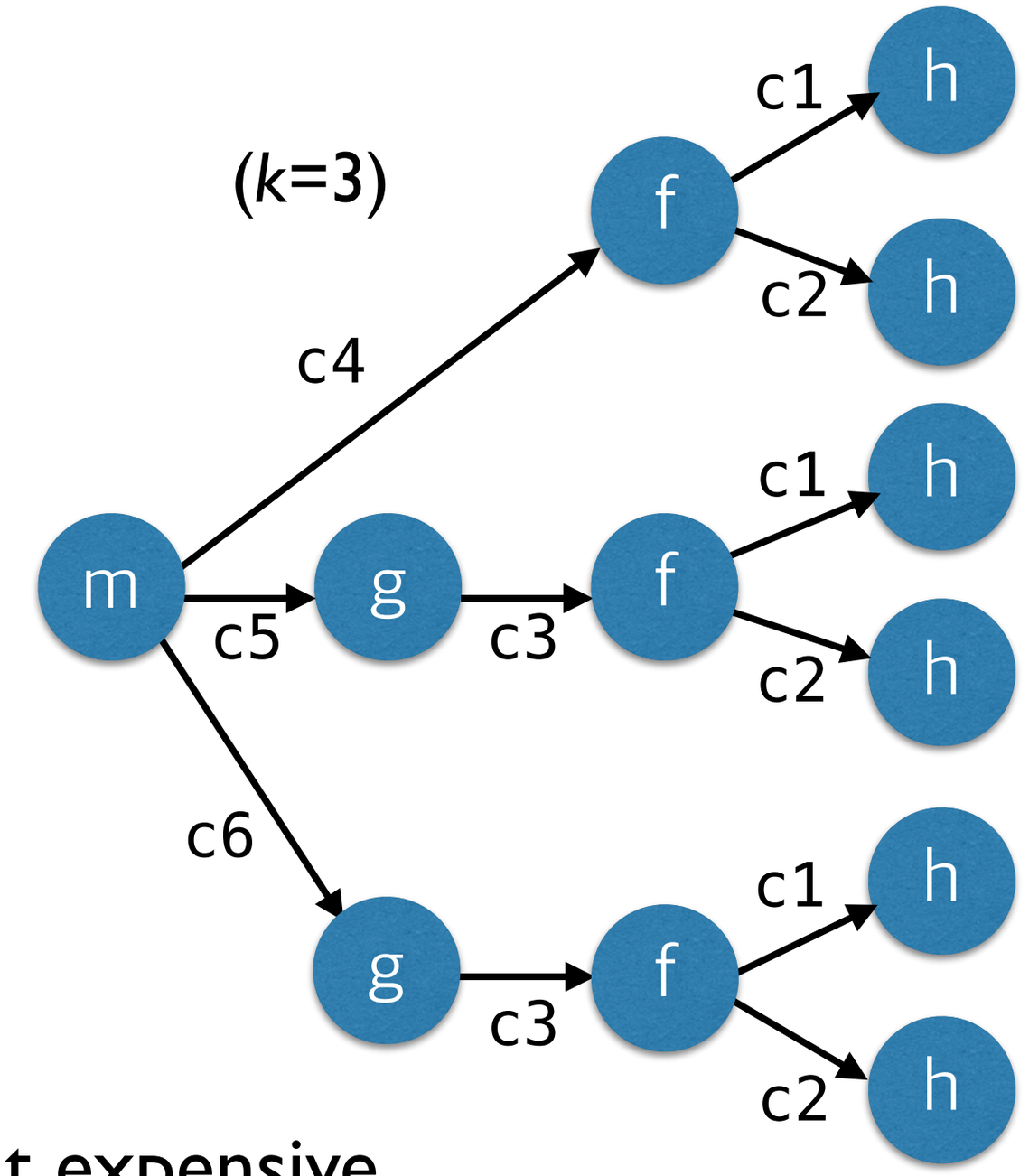


cheap but imprecise

k-Context-Sensitive Analysis

- Analyze functions separately for each calling context

```
    int h(n) {ret n;}

    void f(a) {
c1:   x = h(a);
      assert(x > 0);
c2:   y = h(input());
    }

c3: void g() {f(8);}

    void m() {
c4:   f(4);
c5:   g();
c6:   g();
    }
```



precise but expensive

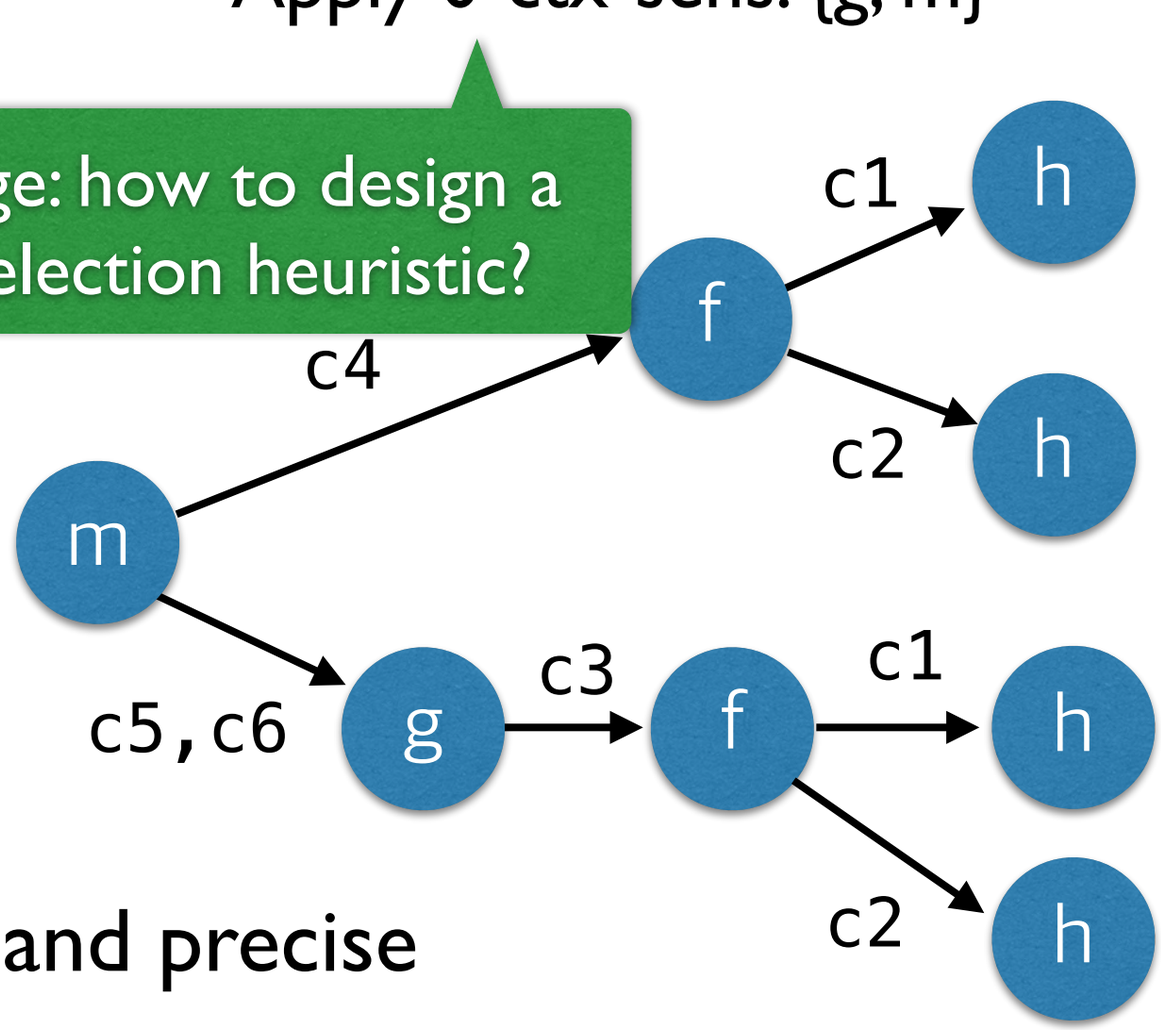
Selective Context-Sensitivity

- Selectively differentiate contexts only when necessary

```
    int h(n) {ret n;}

    void f(a) {
c1:   x = h(a);
      assert(x > 0);
c2:   y = h(inpu
    }

c3: void g() {f(8);}

    void m() {
c4:   f(4);
c5:   g();
c6:   g();
    }
```

Apply 2-ctx-sens: {h}
Apply 1-ctx-sens: {f}
Apply 0-ctx-sens: {g, m}

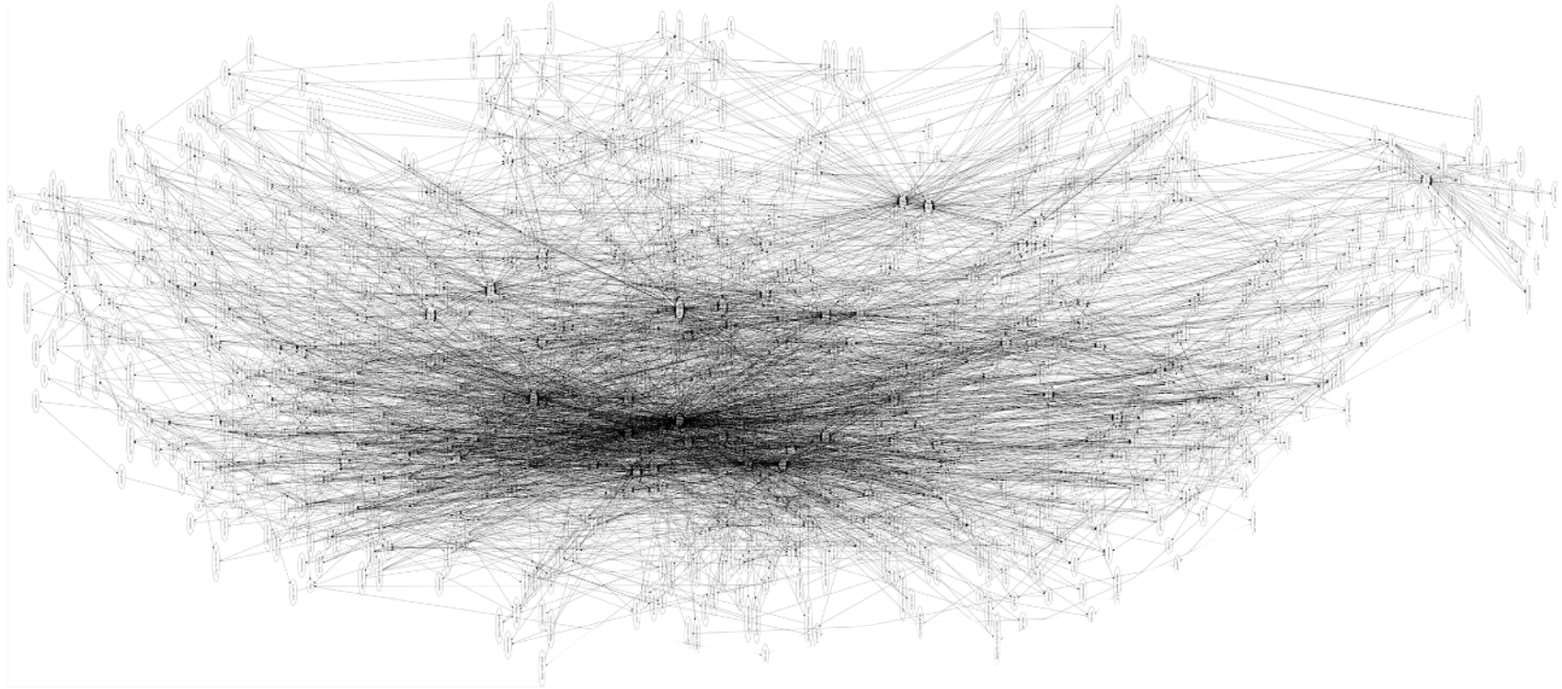
Challenge: how to design a good selection heuristic?



cheap and precise

Hard Search Problem

- Intractably large and sparse search space, if not infinite
  - e.g., S^k choices where $S = 2^{|Proc|}$ for *k*-context-sensitivity
- Real programs are complex to reason about
  - e.g., typical call-graph of real program:



A fundamental problem in program analysis
=> New data-driven approach

Existing Approaches

- Selection heuristics manually crafted by analysis experts:
  - pre-analysis [PLDI'14a, PLDI'14b, OOPSLA'18c]
  - dynamic analysis [POPL'12]
  - online refinement [PLDI'14c, POPL'17]
- Our claim: manual approaches are inherently limited:
  - nontrivial, sub-optimal, and unstable

Our direction: automatically generate heuristics via learning

Direction and Achievement

- Learning algorithms for data-driven program analysis
  - learning models [OOPSLA'15, OOPSLA'17a]
  - optimization algorithms [TOPLAS'19]
  - feature engineering [OOPSLA'17b]
- State-of-the-art program analyses enabled by algorithms
  - interval / pointer analysis [OOPSLA'18a, ICSE'19, TOPLAS'18]
  - symbolic analysis / execution [ICSE'18, ASE'18]
  - others program analyses [FSE'18, OOPSLA'18b]

OOPSLA'17a
TOPLAS'19

Learning Algorithm Overview

Parametric program analyzer

Training data (programs w/o labels)

Atomic features (a1,a2,…,a25)

e.g., procedures have invocation stmt, procedures return strings, etc

Learning Algorithm

Learned heuristic for applying context-sensitivity:

f2: procedures to apply 2-context-sensitivity

$$1 \wedge \neg 3 \wedge \neg 6 \wedge 8 \wedge \neg 9 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25$$

f1: procedures to apply 1-context-sensitivity

$$(1 \wedge \neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \neg 8 \wedge 6 \wedge \neg 9 \wedge \neg 15 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$$
$$(\neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge 10 \wedge 11 \wedge 12 \wedge 13 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$$
$$(\neg 3 \wedge \neg 9 \wedge 13 \wedge 14 \wedge 15 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$$
$$(1 \wedge 2 \wedge \neg 3 \wedge 4 \wedge \neg 5 \wedge \neg 6 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge \neg 10 \wedge \neg 13 \wedge \neg 15 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22$$
$$\wedge \neg 23 \wedge \neg 24 \wedge \neg 25)$$

cf) Atomic Features

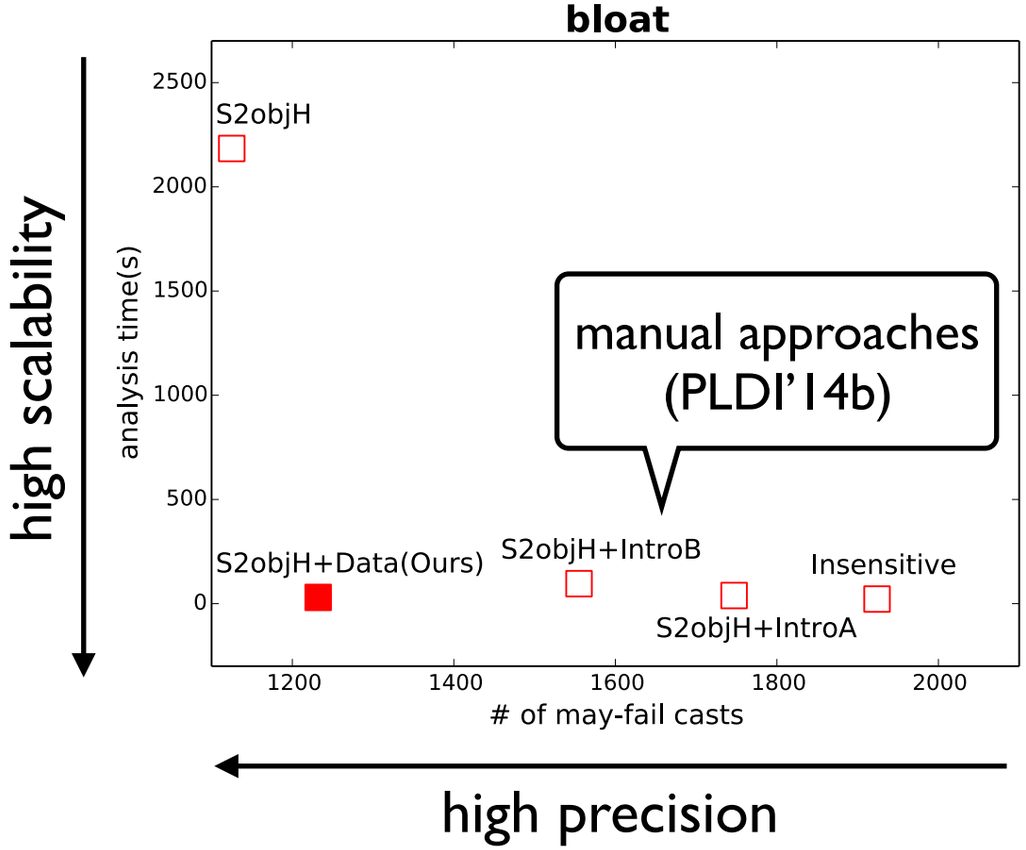
Signature features									
#1	“java”	#3	“sun”	#5	“void”	#7	“int”	#9	“String”
#2	“lang”	#4	“()”	#6	“security”	#8	“util”	#10	“init”

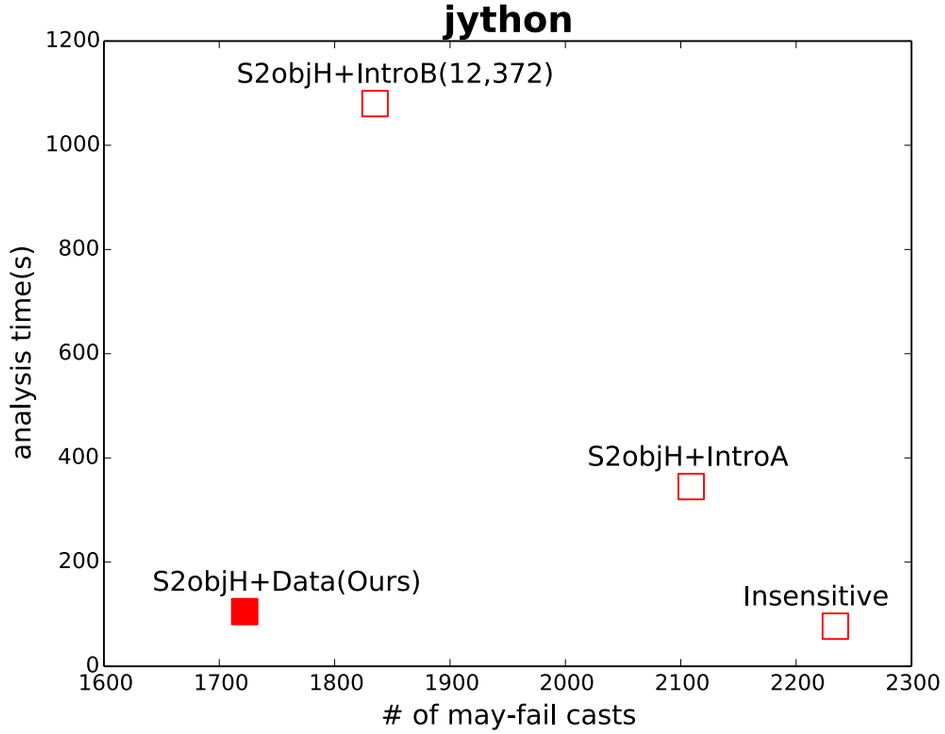
Statement features					
#11	AssignStmt	#16	BreakpointStmt	#21	LookupStmt
#12	IdentityStmt	#17	EnterMonitorStmt	#22	NopStmt
#13	InvokeStmt	#18	ExitMonitorStmt	#23	RetStmt
#14	ReturnStmt	#19	GotoStmt	#24	ReturnVoidStmt
#15	ThrowStmt	#20	IfStmt	#25	TableSwitchStmt

OOPSLA'17a

Application to Pointer Analysis

- Achieved state-of-the-art pointer analysis for Java
  - foundational static analysis for bug-finders, verifiers, etc
- Trained with 5 small programs from the DaCapo benchmark and tested with 5 remaining large programs

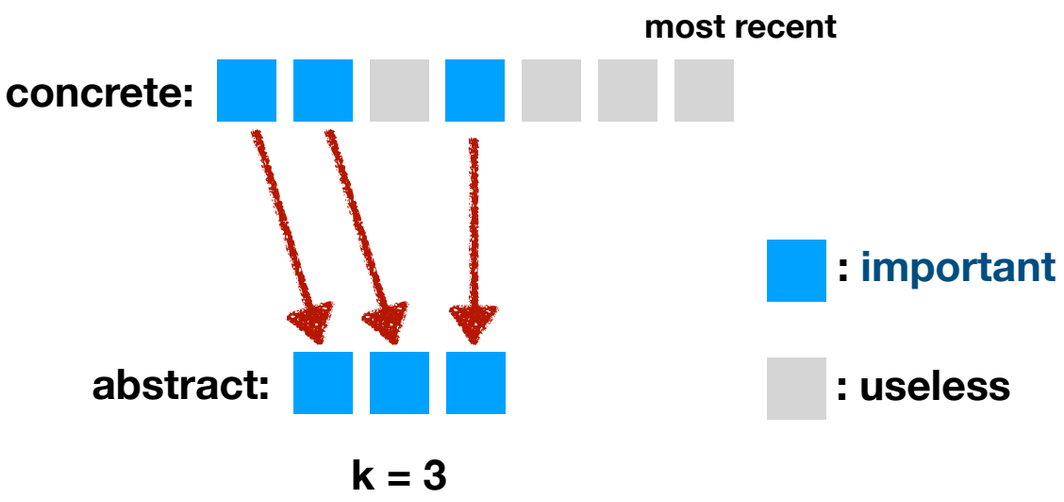


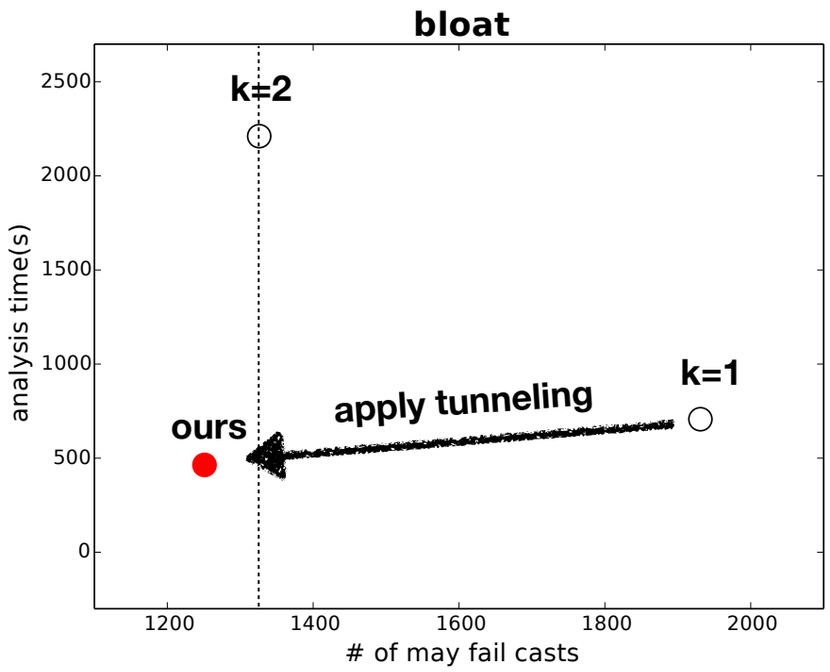


OOPSLA'18a

Application to Pointer Analysis

- Cracked down the precision limit of pointer analysis
- Key enablers:
  - keep most important *k*, rather than most recent *k*
  - data-driven method determines importance





Concolic Testing (Dynamic Symbolic Execution)

- Concolic testing is an effective software testing method based on symbolic execution

- Key challenge: path explosion
- Our solution: mitigate the problem with good search heuristics

Limitation of Random Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Probability of the error? ($0 \leq x,y \leq 100$)

< 0.4%

- random testing requires 250 runs
- concolic testing finds it in 3 runs

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=22, y=7	x=α, y=β
	true

1st iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);
  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=22, y=7, z=14	x=α, y=β, z=2*β
	true

1st iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

	Concrete State	Symbolic State
1st iteration	x=22, y=7, z=14	x=α, y=β, z=2*β 2*β ≠ α

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
	Solve: $2*\beta = \alpha$ Solution: $\alpha=2, \beta=1$
x=22, y=7, z=14	x=α, y=β, z=2*β $2*\beta \neq \alpha$

1st iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=2, y=1	x=α, y=β
	true

2nd iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
  z := double (y);
  if (z==x) {
    if (x>y+10) {
      Error;
    }
  }
}
```

	Concrete State	Symbolic State
z := double (y);	x=2, y=1, z=2	x=α, y=β, z=2*β
if (z==x) {		true

2nd iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {
    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=2, y=1, z=2	x=α, y=β, z=2*β 2*β = α

2nd iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=2, y=1, z=2	x=α, y=β, z=2*β
	2*β = α ∧ α ≤ β+10

2nd iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
	Solve: $2^*\beta = \alpha \wedge \alpha > \beta+10$ Solution: $\alpha=30,\ \beta=15$
$x=2, y=1,$ $z=2$	$x=\alpha, y=\beta, z=2^*\beta$ $2^*\beta = \alpha \wedge$ $\alpha \leq \beta+10$

2nd iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=30, y=15	x=α, y=β
	true

3rd iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);
  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=30, y=15, z=30	x=α, y=β, z=2*β
	true

3rd iteration

Concolic Testing

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {
    if (x>y+10) {
      Error;
    }
  }
}
```

	Concrete State	Symbolic State
←	x=30, y=15, z=30	x=α, y=β,z=2*β 2*β = α

3rd iteration

Concolic Testing

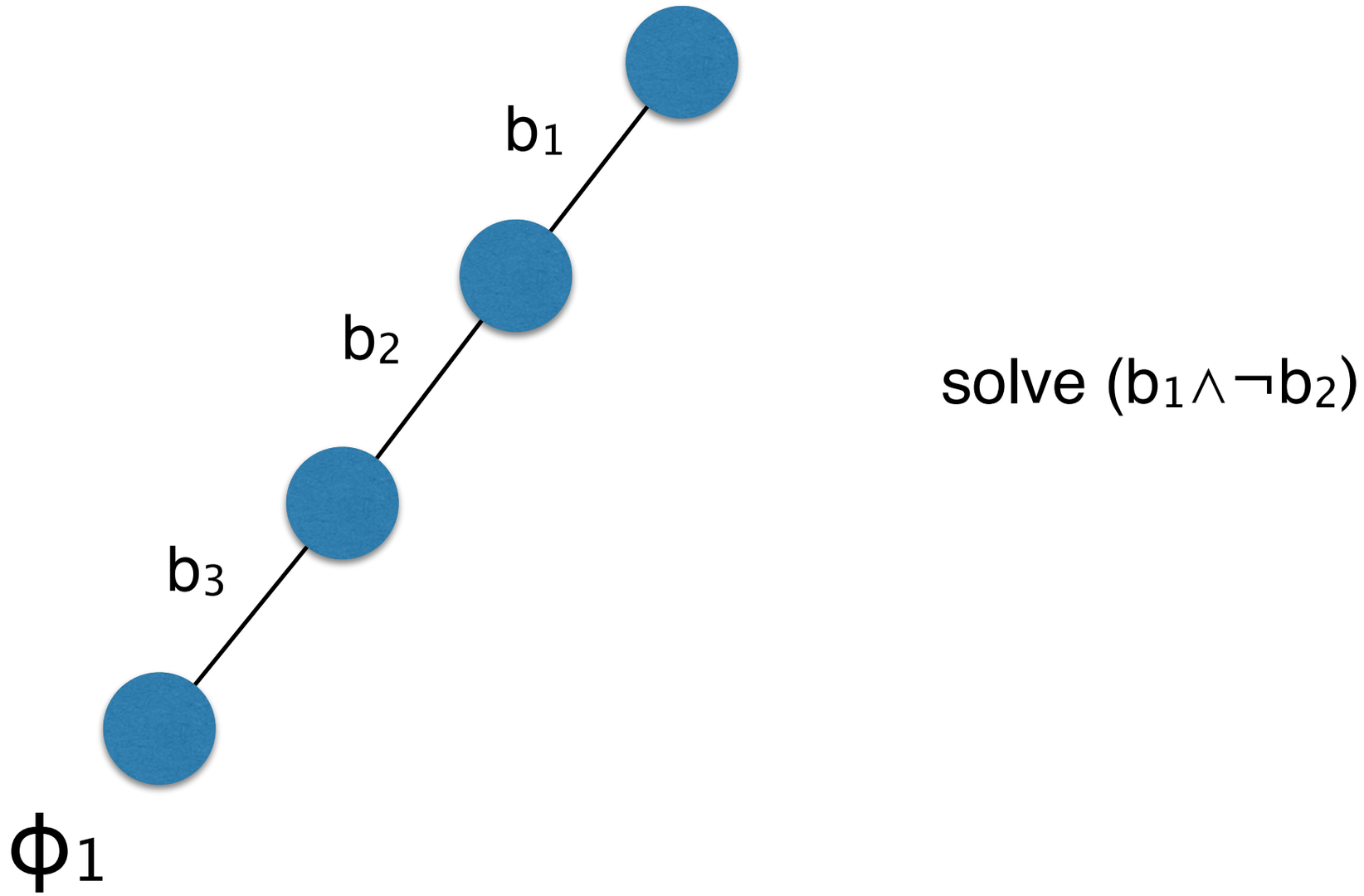
```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
  z := double (y);
  if (z==x) {
    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=30, y=15, z=30	x=α, y=β, z=2*β
	2*β = α ∧ α > β+15

error-triggering input

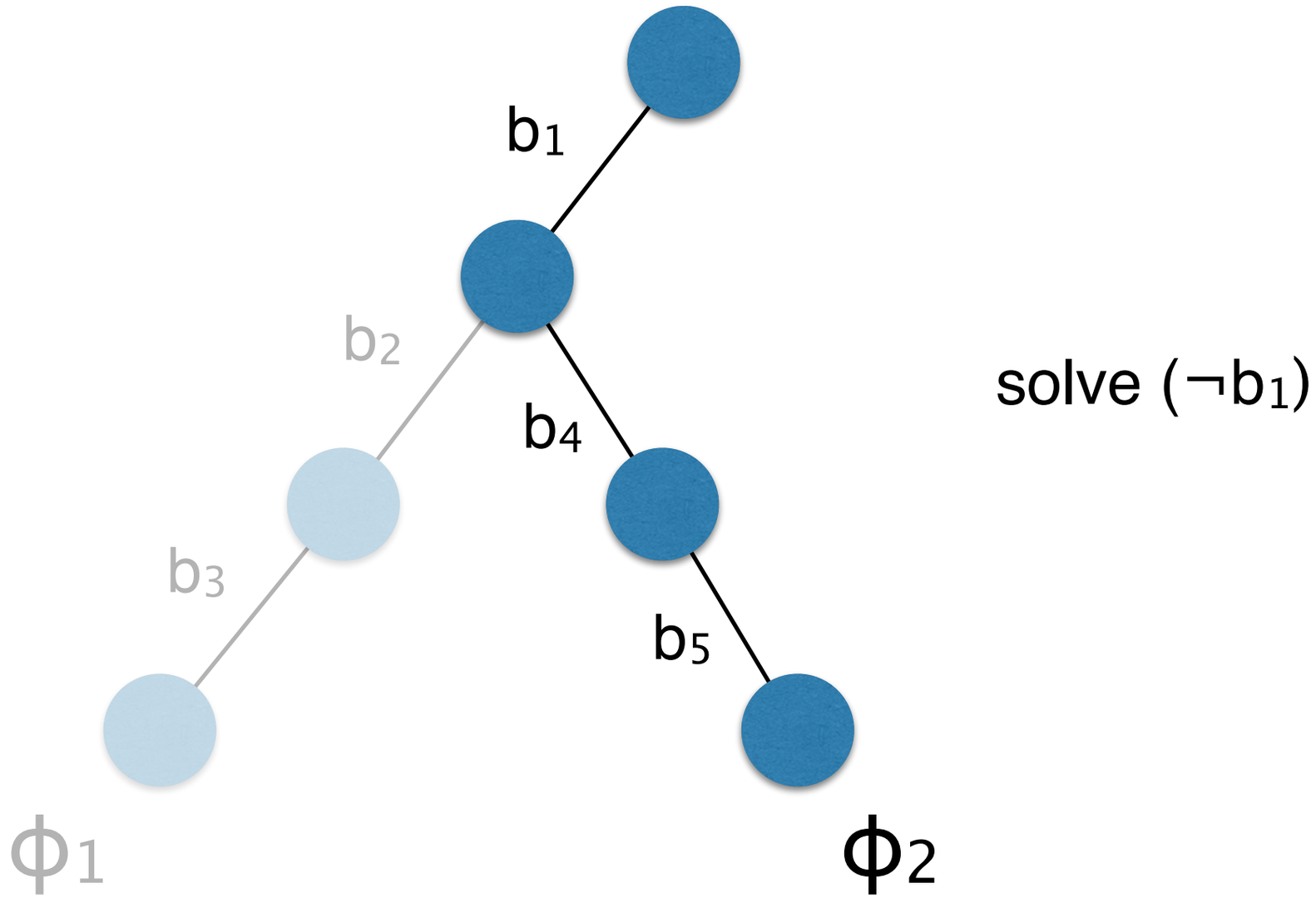
3rd iteration

Concolic Testing



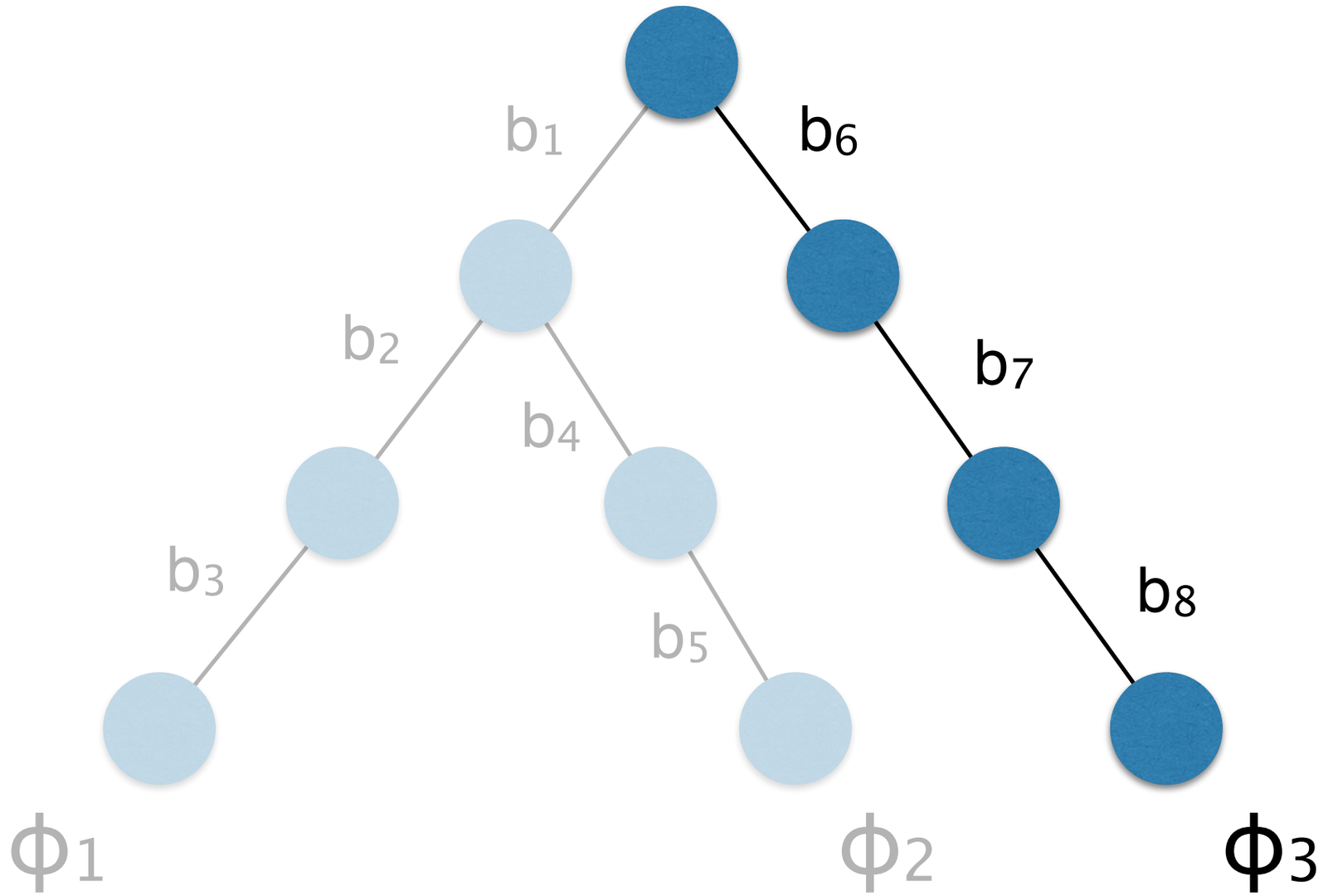
execution tree

Concolic Testing



execution tree

Concolic Testing



execution tree

Concolic Testing Algorithm

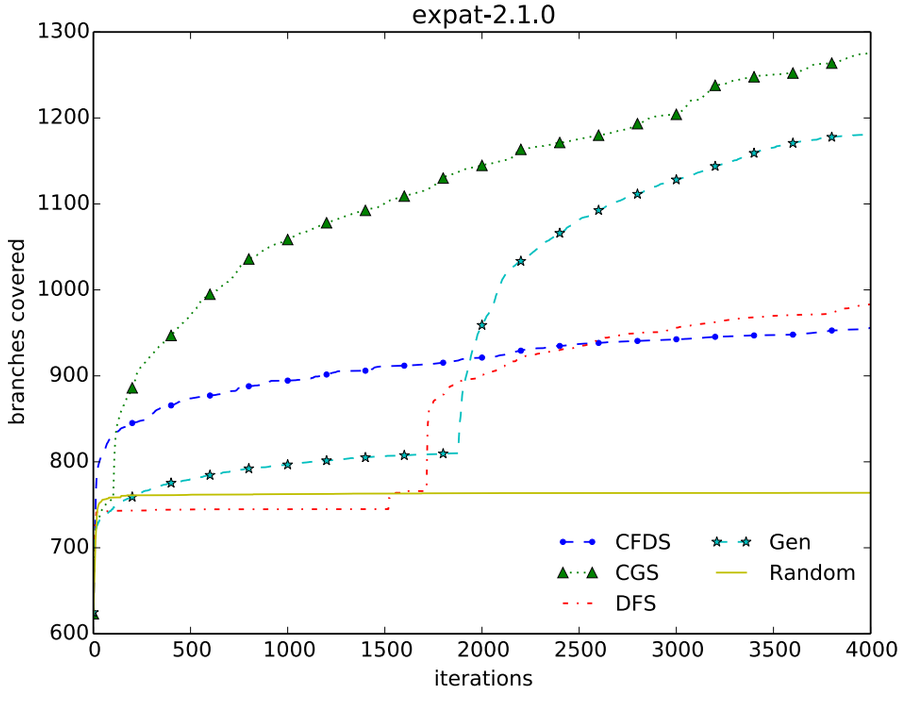
Input : Program P, initial input vector v_0, budget N
Output : The number of branches covered

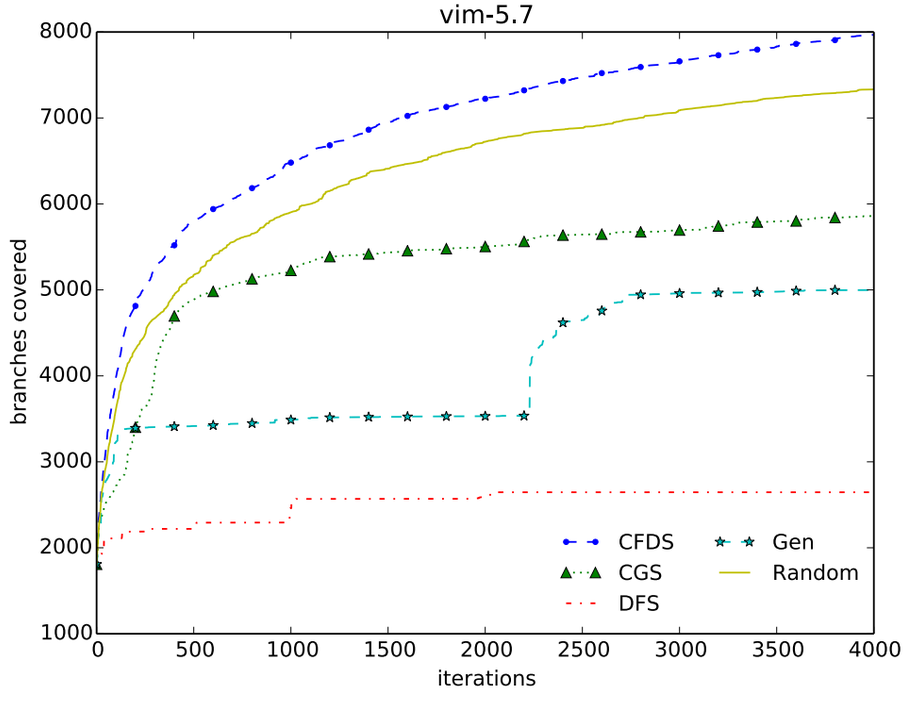
1: $T \leftarrow \langle\rangle$
2: $v \leftarrow v_0$
3: **for** $m = 1$ to N **do**
4: $\quad \Phi_m \leftarrow$ RunProgram(P
5: $\quad T \leftarrow T \cdot \Phi_m$
6: $\quad$ **repeat**
7: $\quad\quad (\Phi, \phi_i) \leftarrow$ Choose(T) $\qquad (\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$
8: $\quad$ **until** SAT($\bigwedge_{j<i} \phi_j \wedge \neg\phi_i$)
9: $\quad v \leftarrow$ model($\bigwedge_{j<i} \phi_j \wedge \neg\phi_i$)
10: **end for**
11: **return** |Branches(T)|

Search Heuristic

Existing Approaches

- Exissting search heuristics have been hand-tuned:
  - e.g., CGS [FSE'14], CarFast [FSE'12], CFDS [ASE'08], Generational [NDSS'08], DFS [PLDI'05], …



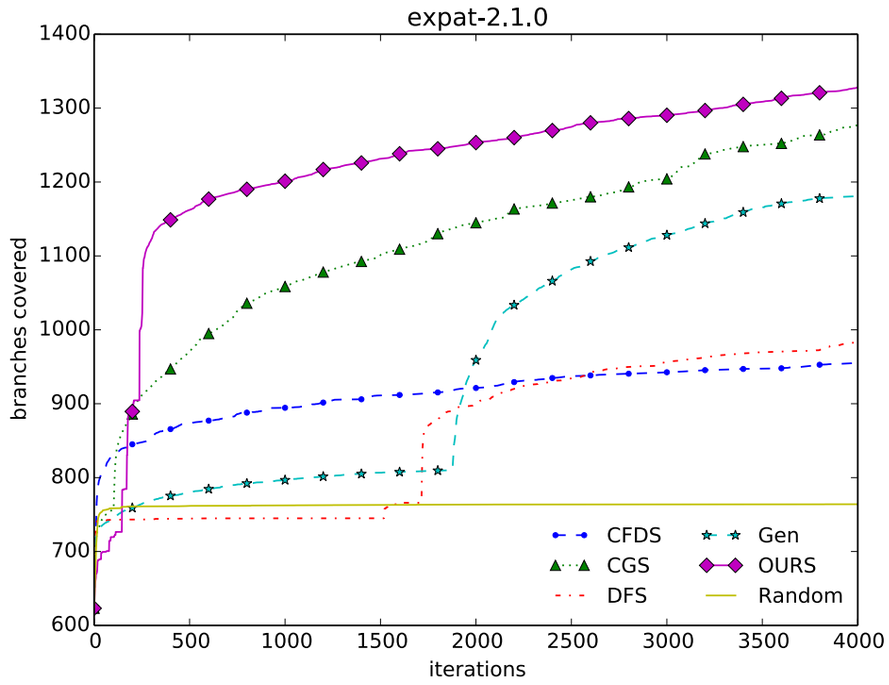


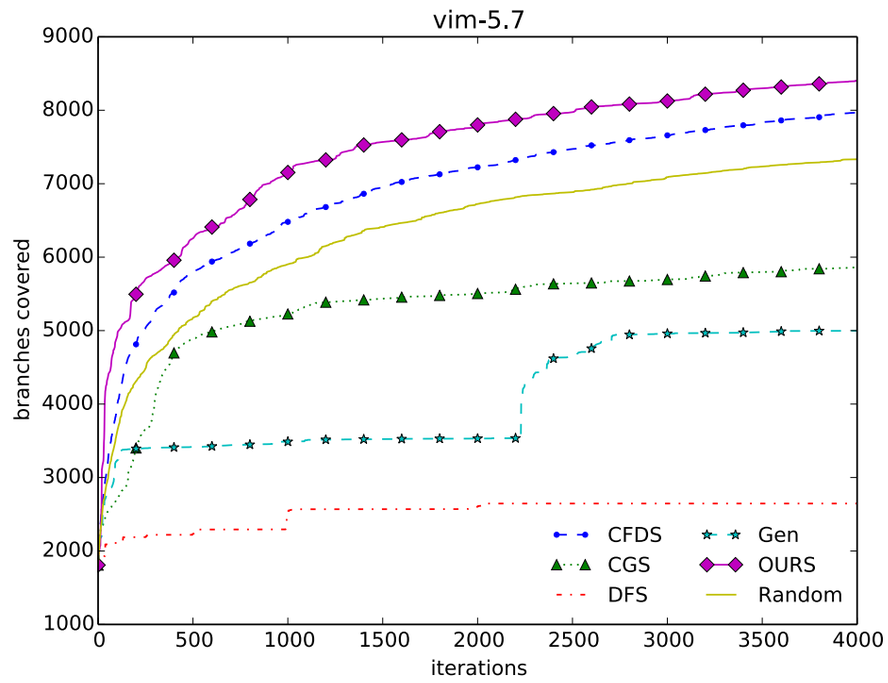
Our goal: automatically generating path-selection heuristics

ICSE'18

Data-Driven Concolic Testing

- Considerable increase in code coverage





- Dramatic increase in bug-finding capability

	OURS	CFDS	CGS	Random	Gen	DFS
gawk-3.0.3	**100/100**	0/100	0/100	0/100	0/100	0/100
grep-2.2	**47/100**	0/100	5/100	0/100	0/100	0/100

	Phenomenons	**Bug-Triggering Inputs**	**Version**
sed	Memory Exhaustion	**'H g ;D'**	4.4(latest)
sed	Infinite File Write	**'H w {- x; D'**	4.4(latest)
grep	Segmentation Fault	**'\(\)\1\+**'**	3.1(latest)
grep	Non-Terminating	**'?(^(|^+*)*\+\{8957\}'**	3.1(latest)
gawk	Memory Exhaustion	**'$6672467e2=E7'**	4.21(latest)

Learning Algorithm Overview (OOPSLA'15)

Static Analyzer

$$F(p, a) \Rightarrow n$$

number of proved assertions

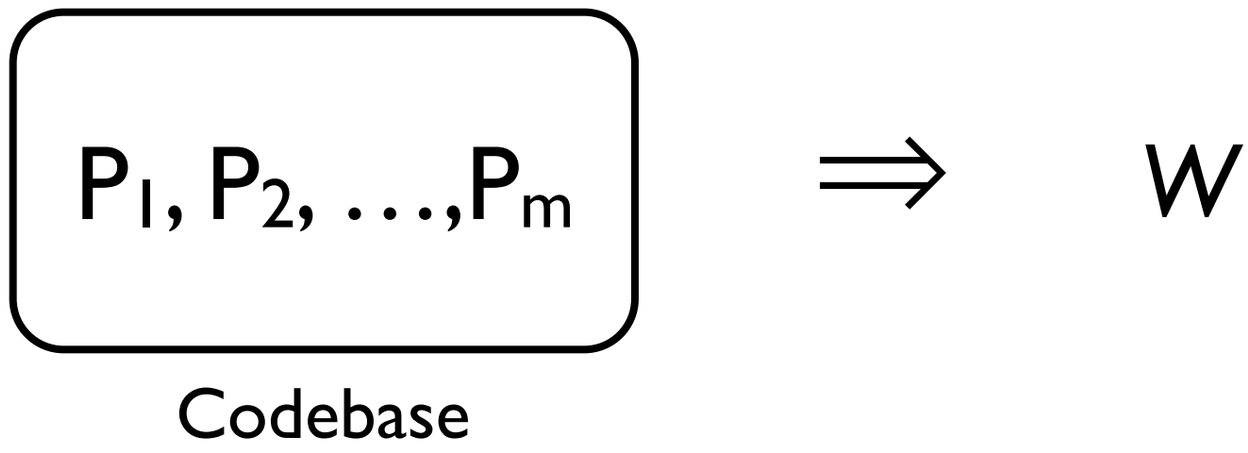
abstraction (e.g., a set of procedures)

Overall Approach

- Parameterized heuristic

$$H_w : \text{pgm} \rightarrow 2^{\text{Proc}}$$

- Learn a good parameter *W* from existing codebase



- For new program P, run static analysis with *Hw*(P)

1. Parameterized Heuristic

$$H_w : \text{pgm} \rightarrow 2^{\text{Proc}}$$

(1) Represent procedures as feature vectors.

(2) Compute the score of each procedure.

(3) Choose the top-k procedures based on the score.

(1) Features

- Predicates over procedures:

$$f = \{f_1, f_2, \ldots, f_5\} \quad (f_i : \text{Proc} \rightarrow \{0,1\})$$

- We used 25 simple syntactic features

Signature features									
#1	“java”	#3	“sun”	#5	“void”	#7	“int”	#9	“String”
#2	“lang”	#4	“()”	#6	“security”	#8	“util”	#10	“init”

Statement features					
#11	AssignStmt	#16	BreakpointStmt	#21	LookupStmt
#12	IdentityStmt	#17	EnterMonitorStmt	#22	NopStmt
#13	InvokeStmt	#18	ExitMonitorStmt	#23	RetStmt
#14	ReturnStmt	#19	GotoStmt	#24	ReturnVoidStmt
#15	ThrowStmt	#20	IfStmt	#25	TableSwitchStmt

(1) Features

- Represent each procedure as a feature vector:

$$f(x) = \langle f_1(x), f_2(x), f_3(x), f_4(x), f_5(x) \rangle$$

$$f(x) = \langle 1,0,1,0,0 \rangle$$
$$f(y) = \langle 1,0,1,0,1 \rangle$$
$$f(z) = \langle 0,0,1,1,0 \rangle$$

(2) Scoring

- The parameter **w** is a real-valued vector: e.g.,

$$\mathbf{w} = \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle$$

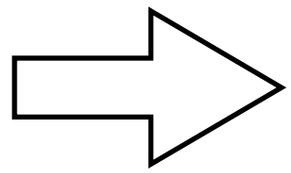
- Compute scores of procedures:

$$\text{score}(x) = \langle 1,0,1,0,0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.3$$
$$\text{score}(y) = \langle 1,0,1,0,1 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.6$$
$$\text{score}(z) = \langle 0,0,1,1,0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.1$$

(3) Choose Top-k Variables

- Choose the top-k procedures based on their scores: e.g., when k=2,

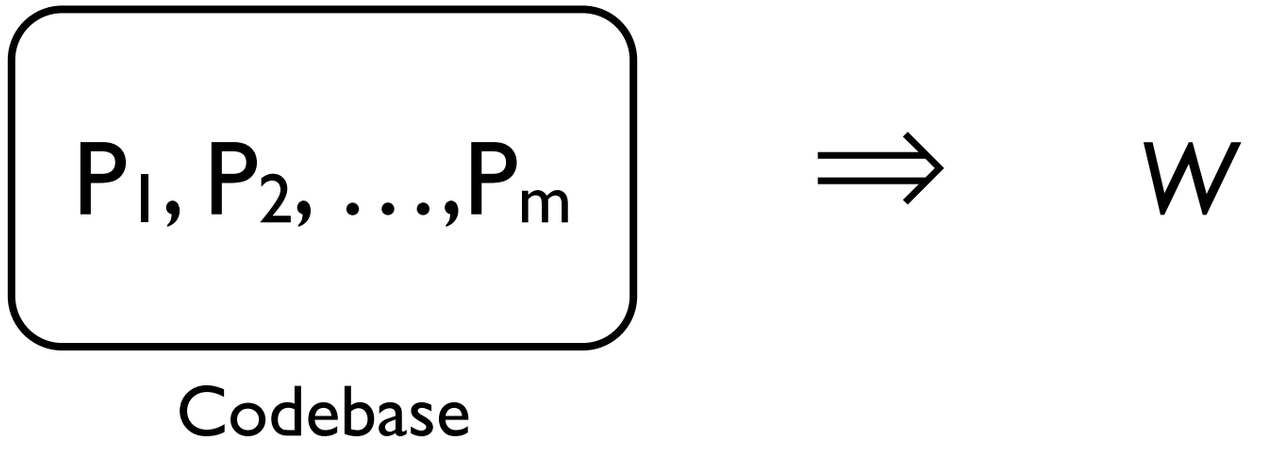
score(x) = 0.3
score(y) = 0.6
score(z) = 0.1



{x,y}

- In experiments, we choose 10% of procedures with highest scores.

2. Learn a Good Parameter



- Learning = solving the optimization problem:

$$\text{Find } \mathbf{w} \text{ that maximizes } \sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i))$$

- We solve it via Bayesian optimization (details in paper)

Limitations & Follow-ups

- Limited expressiveness due to linear combination
  - Disjunctive heuristic [OOPSLA'17a, TOPLAS'19,…]
- Semi-automatic due to manual feature engineering
  - Automated feature engineering [OOPSLA'17b]
- High learning cost due to black-box optimization
  - More efficient approaches [SAS'16, APLAS'16, ICSE'17,…]

OOPSLA'17

Learning with Disjunctive Heuristics

- The linear heuristic cannot express disjunctive properties:

$x : \{a_1, a_2\}$
$y : \{a_1\}$
$z : \{a_2\}$
$w : \emptyset$

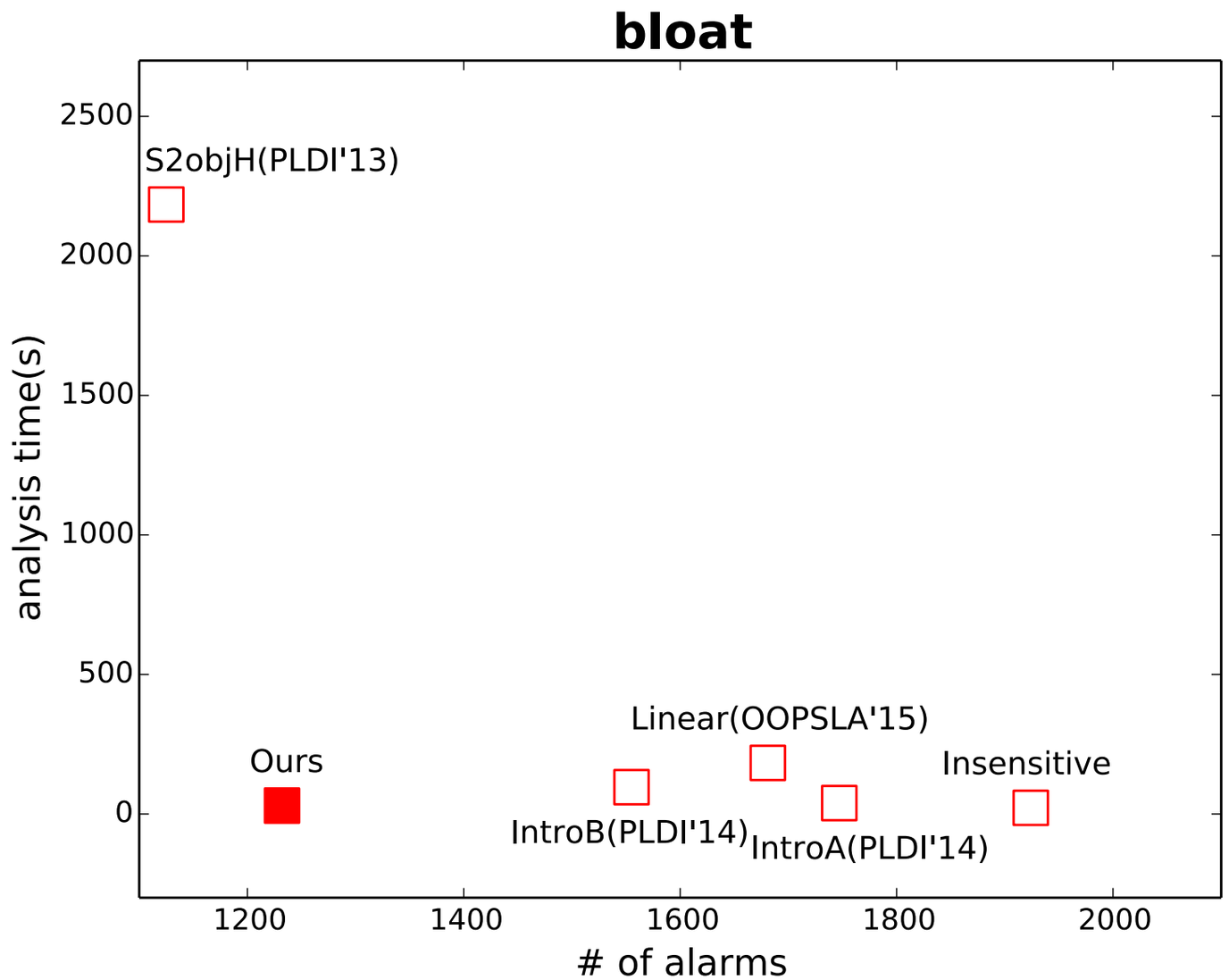
$\text{Goal: } \{x, w\}$

$(a_1 \wedge a_2) \vee (\neg a_1 \wedge \neg a_2)$

- Disjunctive heuristic + algorithm for learning boolean formulas

Performance

- Without disjunction, the learned heuristic lags behind hand-tuning becaused of limited expressiveness



Manual Feature Engineering

- The success of ML heavily depends on the "features"
- Feature engineering is nontrivial and time-consuming
- Features do not generalize to other analyses

Type	#	Features
A	1	local variable
	2	global variable
	3	structure field
	4	location created by dynamic memory allocation
	5	defined at one program point
	6	location potentially generated in library code
	7	assigned a constant expression (e.g., x = c1 + c2)
	8	compared with a constant expression (e.g., x < c)
	9	compared with an other variable (e.g., x < y)
	10	negated in a conditional expression (e.g., if (!x))
	11	directly used in malloc (e.g., malloc(x))
	12	indirectly used in malloc (e.g., y = x; malloc(y))
	13	directly used in realloc (e.g., realloc(x))
	14	indirectly used in realloc (e.g., y = x; realloc(y))
	15	directly returned from malloc (e.g., x = malloc(e))
	16	indirectly returned from malloc
	17	directly returned from realloc (e.g., x = realloc(e))
	18	indirectly returned from realloc
	19	incremented by one (e.g., x = x + 1)
	20	incremented by a constant expr. (e.g., x = x + (1+2))
	21	incremented by a variable (e.g., x = x + y)
	22	decremented by one (e.g., x = x - 1)
	23	decremented by a constant expr (e.g., x = x - (1+2))
	24	decremented by a variable (e.g., x = x - y)
	25	multiplied by a constant (e.g., x = x * 2)
	26	multiplied by a variable (e.g., x = x * y)
	27	incremented pointer (e.g., p++)
	28	used as an array index (e.g., a[x])
	29	used in an array expr. (e.g., x[e])
	30	returned from an unknown library function
	31	modified inside a recursive function
	32	modified inside a local loop
	33	read inside a local loop
B	34	$1 \wedge 8 \wedge (11 \vee 12)$
	35	$2 \wedge 8 \wedge (11 \vee 12)$
	36	$1 \wedge (11 \vee 12) \wedge (19 \vee 20)$
	37	$2 \wedge (11 \vee 12) \wedge (19 \vee 20)$
	38	$1 \wedge (11 \vee 12) \wedge (15 \vee 16)$
	39	$2 \wedge (11 \vee 12) \wedge (15 \vee 16)$
	40	$(11 \vee 12) \wedge 29$
	41	$(15 \vee 16) \wedge 29$
	42	$1 \wedge (19 \vee 20) \wedge 33$
	43	$2 \wedge (19 \vee 20) \wedge 33$
	44	$1 \wedge (19 \vee 20) \wedge \neg 33$
	45	$2 \wedge (19 \vee 20) \wedge \neg 33$

flow-sensitivity

Type	#	Features
A	1	leaf function
	2	function containing malloc
	3	function containing realloc
	4	function containing a loop
	5	function containing an if statement
	6	function containing a switch statement
	7	function using a string-related library function
	8	write to a global variable
	9	read a global variable
	10	write to a structure field
	11	read from a structure field
	12	directly return a constant expression
	13	indirectly return a constant expression
	14	directly return an allocated memory
	15	indirectly return an allocated memory
	16	directly return a reallocated memory
	17	indirectly return a reallocated memory
	18	return expression involves field access
	19	return value depends on a structure field
	20	return void
	21	directly invoked with a constant
	22	constant is passed to an argument
	23	invoked with an unknown value
	24	functions having no arguments
	25	functions having one argument
	26	functions having more than one argument
	27	functions having an integer argument
	28	functions having a pointer argument
	29	functions having a structure as an argument
B	30	$2 \wedge (21 \vee 22) \wedge (14 \vee 15)$
	31	$2 \wedge (21 \vee 22) \wedge \neg(14 \vee 15)$
	32	$2 \wedge 23 \wedge (14 \vee 15)$
	33	$2 \wedge 23 \wedge \neg(14 \vee 15)$
	34	$2 \wedge (21 \vee 22) \wedge (16 \vee 17)$
	35	$2 \wedge (21 \vee 22) \wedge \neg(16 \vee 17)$
	36	$2 \wedge 23 \wedge (16 \vee 17)$
	37	$2 \wedge 23 \wedge \neg(16 \vee 17)$
	38	$(21 \vee 22) \wedge \neg 23$

context-sensitivity

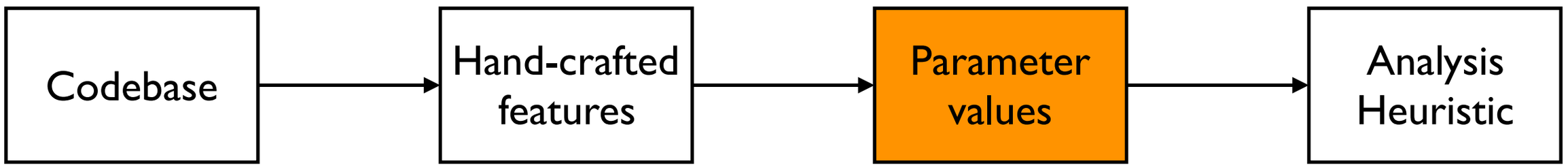
Type	#	Features
A	1	used in array declarations (e.g., a[c])
	2	used in memory allocation (e.g., malloc(c))
	3	used in the righthand-side of an assignment (e.g., x = c)
	4	used with the less-than operator (e.g, x < c)
	5	used with the greater-than operator (e.g., x > c)
	6	used with $\leq$ (e.g., x $\leq$ c)
	7	used with $\geq$ (e.g., x $\geq$ c)
	8	used with the equality operator (e.g., x == c)
	9	used with the not-equality operator (e.g., x ! = c)
	10	used within other conditional expressions (e.g., x < c+y)
	11	used inside loops
	12	used in return statements (e.g., return c)
	13	constant zero
B	14	$(1 \vee 2) \wedge 3$
	15	$(1 \vee 2) \wedge (4 \vee 5 \vee 6 \vee 7)$
	16	$(1 \vee 2) \wedge (8 \vee 9)$
	17	$(1 \vee 2) \wedge 11$
	18	$(1 \vee 2) \wedge 12$
	19	$13 \wedge 3$
	20	$13 \wedge (4 \vee 5 \vee 6 \vee 7)$
	21	$13 \wedge (8 \vee 9)$
	22	$13 \wedge 11$
	23	$13 \wedge 12$

widening thresholds

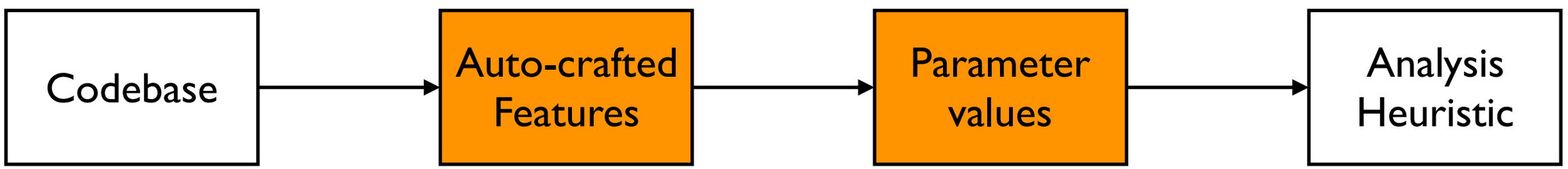
OOPSLA'17

Automating Feature Engineering

Before (OOPSLA'15)



New method (OOPSLA'17)



Thank You!

- Research areas: programming languages, software engineering, software security
  - program analysis and testing
  - program synthesis and repair
- Publication: top-venues in PL, SE, Security, and AI:
  - PLDI('12,'14), OOPSLA('15,'17,'17,'18,'18), TOPLAS('14,'16,'17,'18,'19), ICSE('17,'18,'19), FSE'18, ASE'18, S&P'17, IJCAI('17,'18), etc

http://prl.korea.ac.kr