# Data-Driven Program Analysis

**Hakjoo Oh**

**Korea University**

29 September 2017 @SNU

(co-work with Sooyoung Cha, Kwonsoo Chae, Kihong Heo, Minseok Jeon, Sehun Jeong, Hongseok Yang, Kwangkeun Yi)

# PL Research in Korea Univ.

- We research on technology for safe and reliable software.
- Research areas: programming languages, software engineering, software security
  - program analysis and testing
  - program synthesis and repair
- Publication: top-venues in PL, SE, and Security
  - PLDI('12,'14), ICSE'17, OOPSLA('15,'17,'17), Oakland'17, etc

http://prl.korea.ac.kr

# Heuristics in Static Analysis

Sparrow The Early Bird  WALA T. J. WATSON LIBRARIES FOR ANALYSIS  Chord

Astrée  DOOP  TAJS  SAFE

- Practical static analyzers involve many heuristics
  - Which procedures should be analyzed context-sensitively?
  - Which relationships between variables should be tracked?
  - When to split and merge in trace partitioning?
  - Which program parts to analyze unsoundly or soundly?, etc
- Designing a good heuristic is an art
  - Usually done by trials and error: nontrivial and suboptimal

Goal

# Automatically Generating Heuristics from Data

- Automate the process: use data to make heuristic decisions in static analysis

GitHub

machine learning

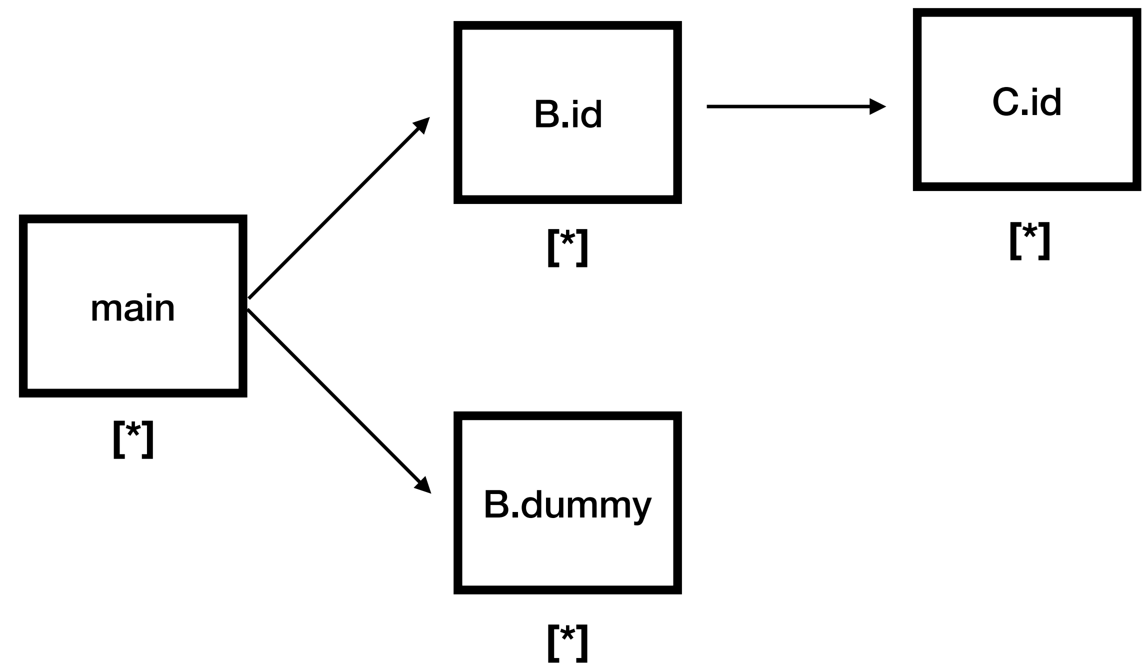
context-sensitivity heuristics
flow-sensitivity heuristics
unsoundness heuristics
…

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

# Context-Sensitivity

```
class D{} class E{}

class C{
Object id(Object v){return v;}}

class B{
void dummy(){}
Object id(Object v){
C c = new C();//C1
return c.id(v);}}

class A{
public static void main(String[] args){
B b1 = new B();//B1
B b2 = new B();//B2
D d = (D) b1.id1(new D());//query1
E e = (E) b2.id1(new E());//query2
b1.dummy();
b2.dummy();}}
```

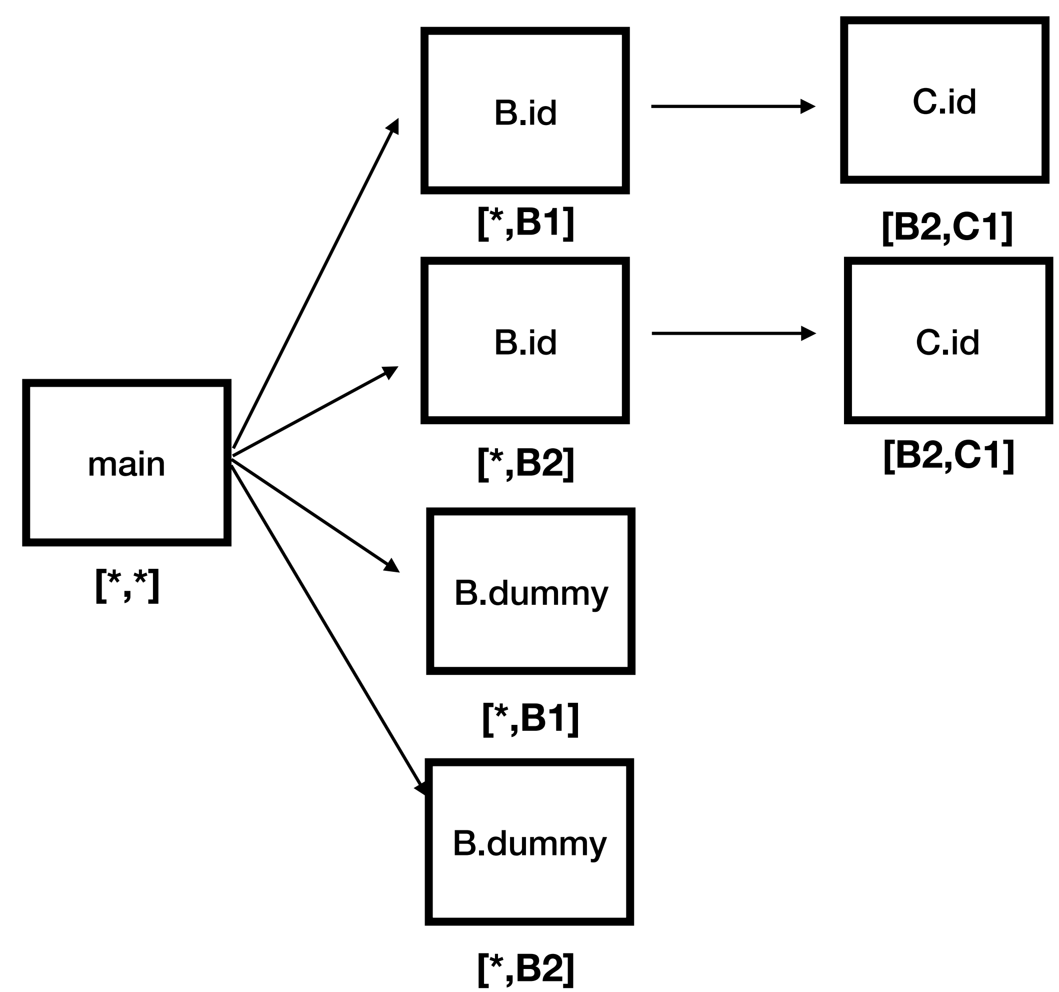
Without context-sensitivity, analysis fails to prove queries



# Context-Sensitivity

```
class D{} class E{}
class C{
Object id(Object v){return v;}}
class B{
void dummy(){}
Object id(Object v){
C c = new C();//C1
return c.id(v);}}
class A{
public static void main(String[] args){
B b1 = new B();//B1
B b2 = new B();//B2
D d = (D) b1.id1(new D());//query1
E e = (E) b2.id1(new E());//query2
b1.dummy();
b2.dummy();}}
```

## 2-object-sensitivity succeeds but does not scale



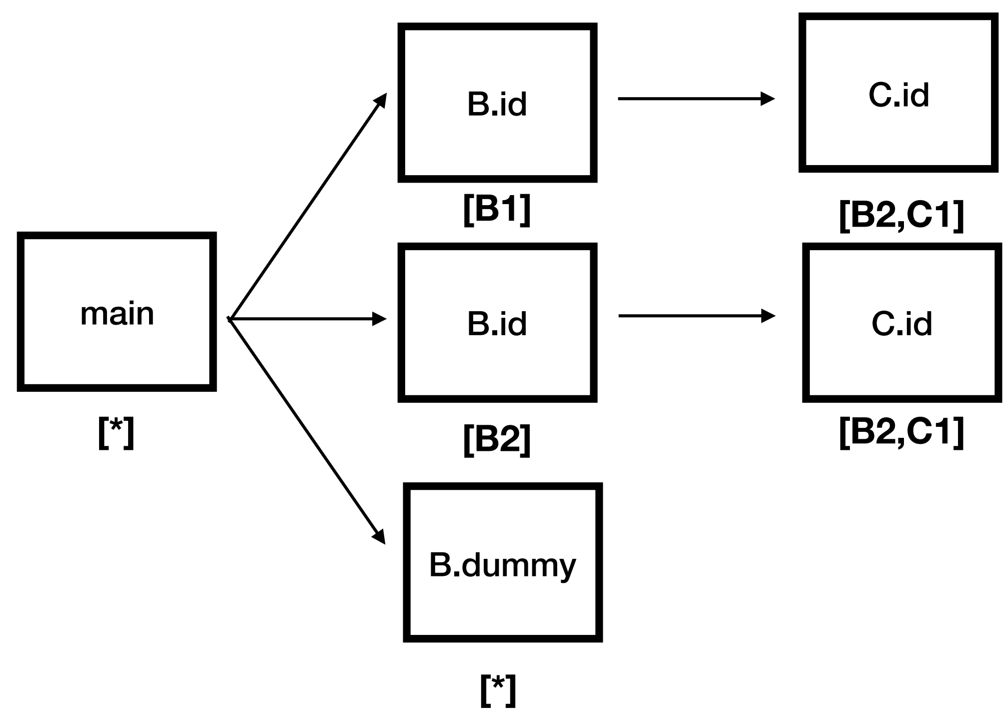
# Selective Context-Sensitivity

```
class D{} class E{}
class C{
Object id(Object v){return v;}}
class B{
void dummy(){}
Object id(Object v){
C c = new C();//C1
return c.id(v);}}
class A{
public static void main(String[] args){
B b1 = new B();//B1
B b2 = new B();//B2
D d = (D) b1.id1(new D());//query1
E e = (E) b2.id1(new E());//query2
b1.dummy();
b2.dummy();}}
```

Apply 2-obj-sens: {C.id}

Apply 1-obj-sens: {B.id}

Apply insens: {B.m}

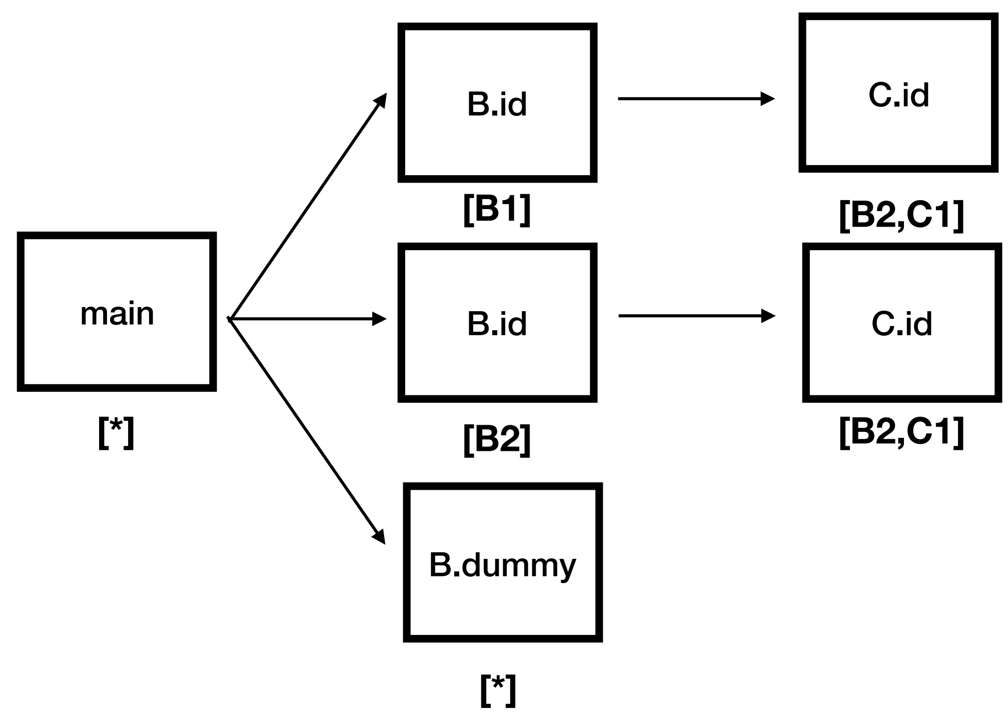


# Selective Context-Sensitivity

Challenge: How to decide? Data-driven approach

```
class D{} class E{}

class C{
Object id(Object v){return v;}}

class B{
void dummy(){}
Object id(Object v){
C c = new C();//C1
return c.id(v);}}

class A{
public static void main(String[] args){
B b1 = new B();//B1
B b2 = new B();//B2
D d = (D) b1.id1(new D());//query1
E e = (E) b2.id1(new E());//query2
b1.dummy();
b2.dummy();}}
```

Apply 2-obj-sens: {C.id}
Apply 1-obj-sens: {B.id}
Apply insens: {B.m}



OOPSLA'17

# Data-Driven Ctx-Sensitivity

Parametric static analyzer

Training data (programs)

Atomic features (a1,a2,…,a25)

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

Our DD Framework

OOPSLA'17

# Data-Driven Ctx-Sensitivity

Parametric static analyzer

Training data (programs)

Atomic features (a1,a2,…,a25)

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

**Our DD Framework**

Heuristic for applying (hybrid) object-sensitivity:

f2: Methods that require 2-object-sensitivity

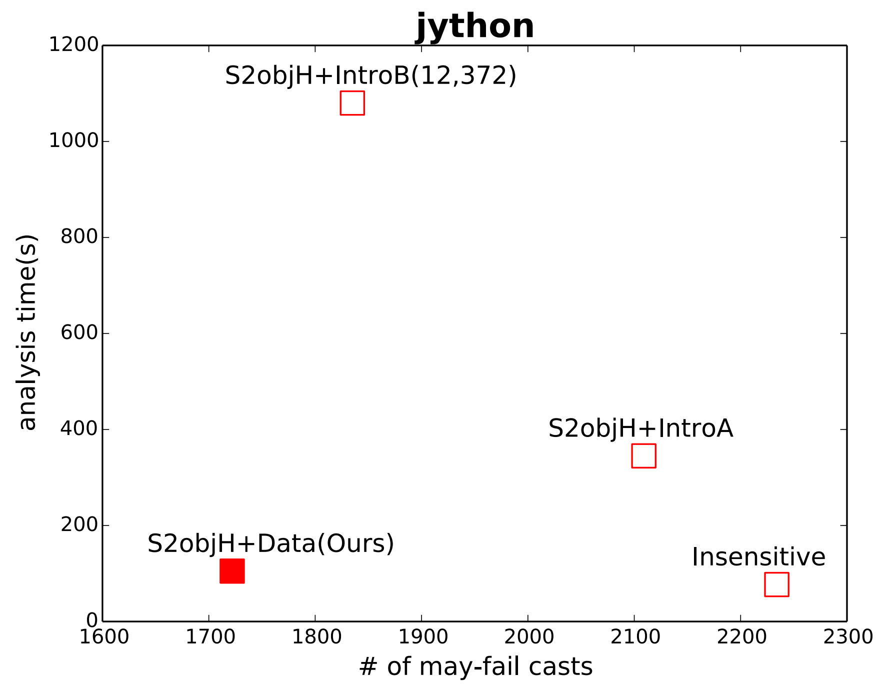
$$1 \land \neg 3 \land \neg 6 \land 8 \land \neg 9 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25$$

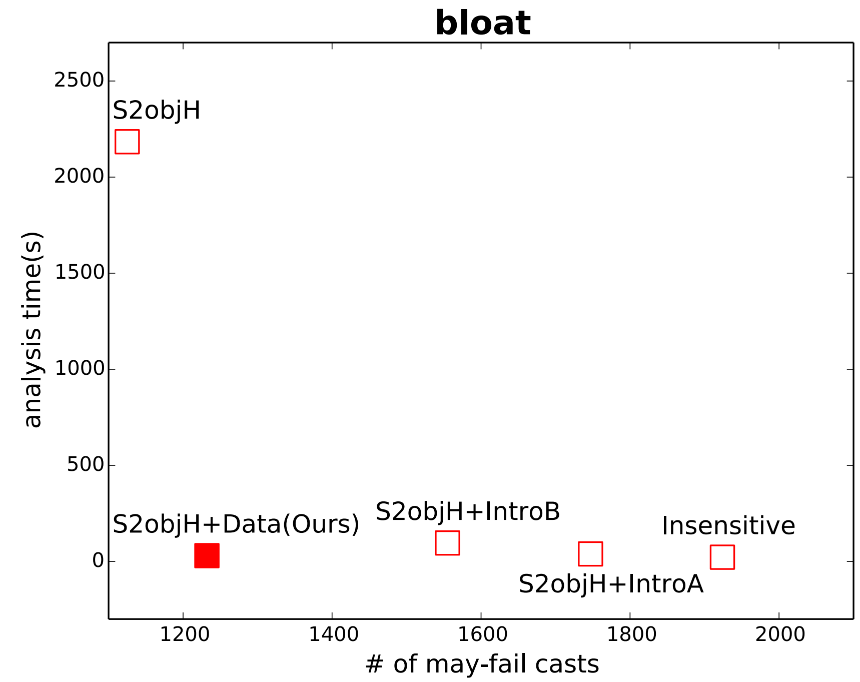
f1: Methods that require 1-object-sensitivity

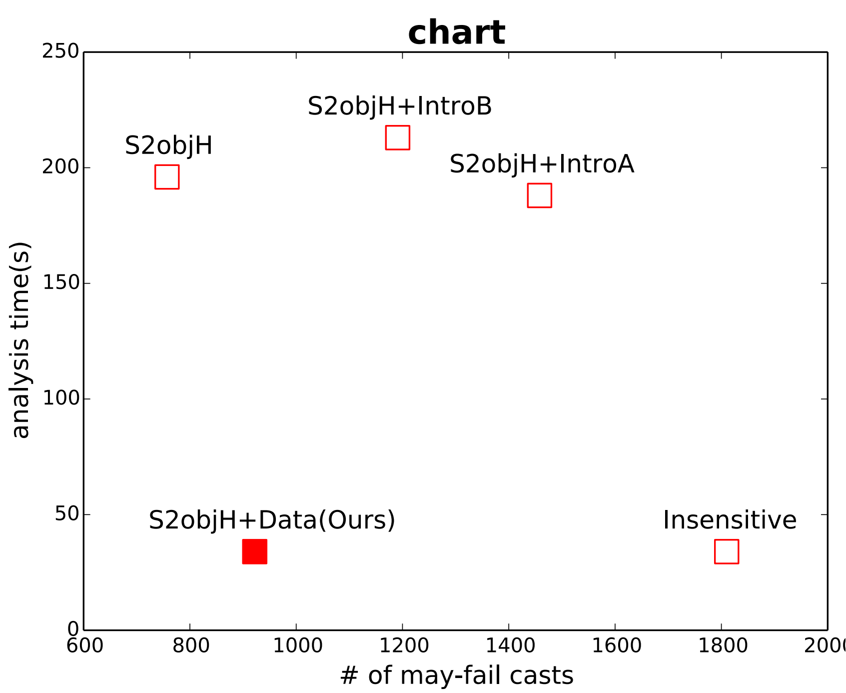
$$(1 \land \neg 3 \land \neg 4 \land \neg 7 \land \neg 8 \land 6 \land \neg 9 \land \neg 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25) \lor$$
$$(\neg 3 \land \neg 4 \land \neg 7 \land \neg 8 \land \neg 9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

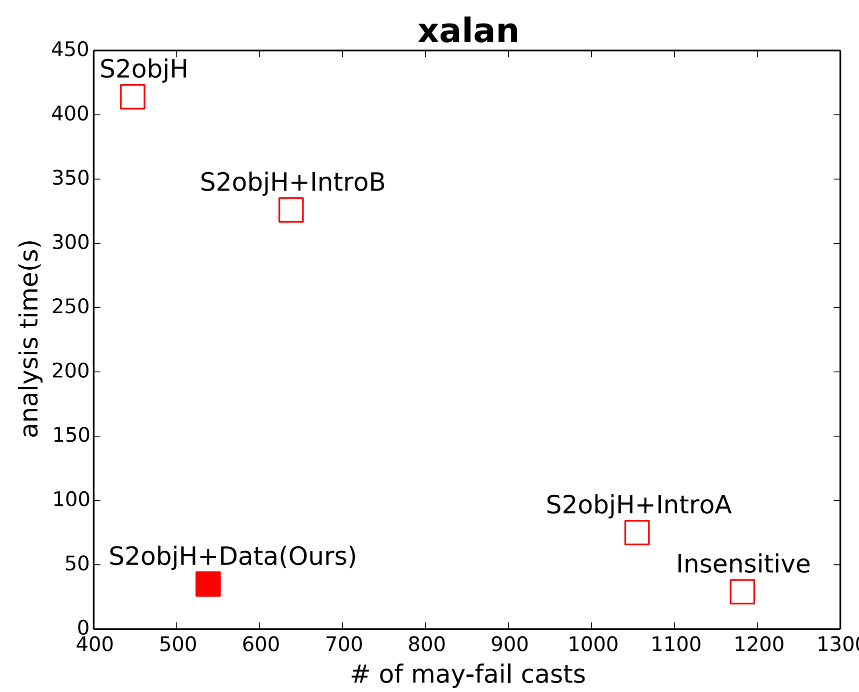
# Performance

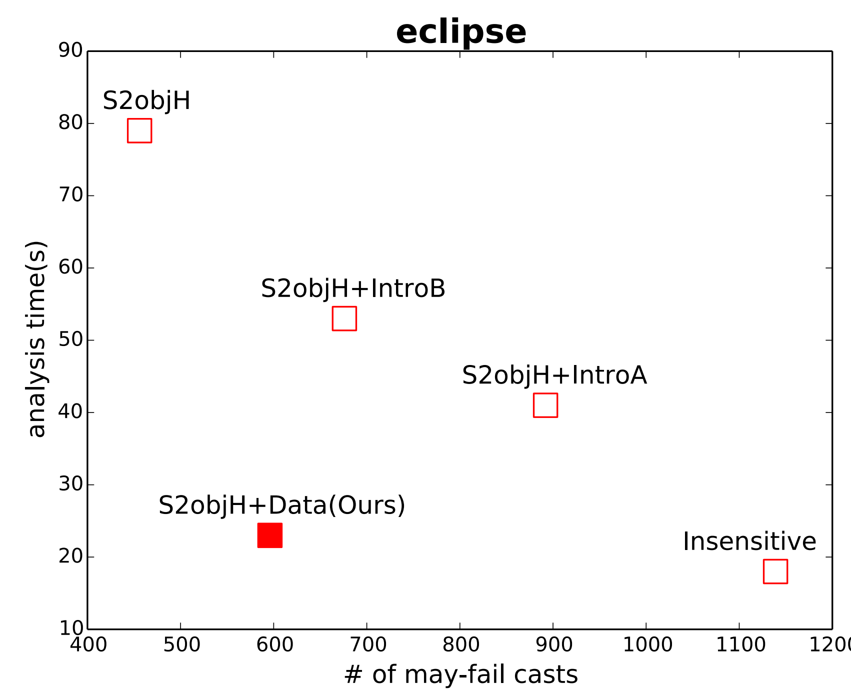
- Training with 4 small programs from DaCapo, and applied to 6 large programs (1 for validation)



# Other Context-Sensitivities

- Plain (not hybrid) Object-sensitivity:
  - Depth-2 formula ($f_2$):

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

  - Depth-1 formula ($f_1$):

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

- Call-site-sensitivity:
  - Depth-2 formula ($f_2$):

$$1 \land \neg 6 \land \neg 7 \land 11 \land 12 \land 13 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25$$

  - Depth-1 formula ($f_1$):

$$(1 \land 2 \land \neg 7 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25)$$

- Type-sensitivity:
  - Depth-2 formula ($f_2$):

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

  - Depth-1 formula ($f_1$):

$$1 \land 2 \land \neg 3 \land \neg 6 \land \neg 7 \land \neg 8 \land \neg 9 \land \neg 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25$$

# Obj-Sens vs. Type-Sens

- In theory, obj-sens is more precise than type-sens
- The set of methods that benefit from obj-sens is a superset of the methods that benefit from type-sens
- Interestingly, our algorithm automatically discovered this rule from data:

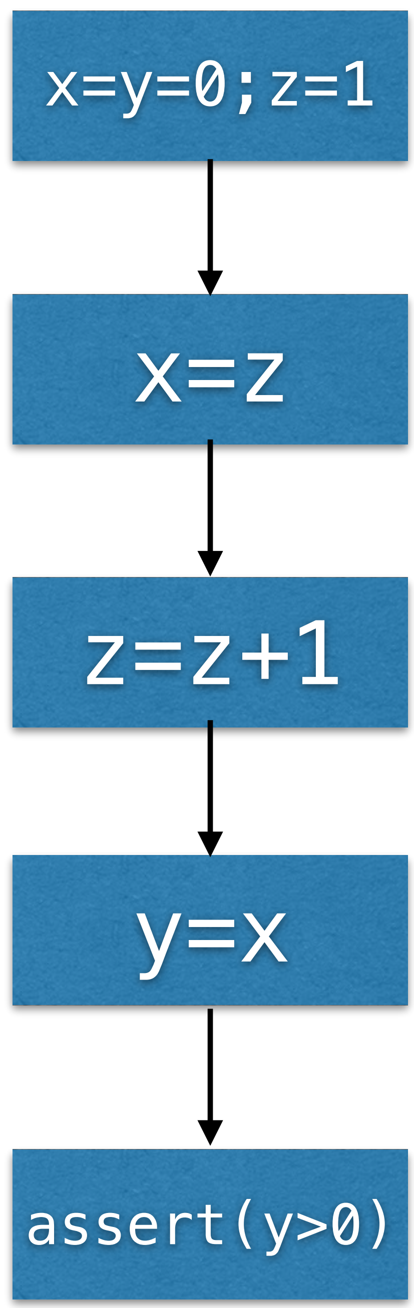
$$
\begin{array}{rcl}
 & & (\mathbf{1 \land 2 \land \neg 3 \land \neg 6 \land \neg 7 \land \neg 8 \land \neg 9 \land \neg 16 \land \cdots \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25})\lor \\
f_1 \text{ for } \mathit{2objH{+}Data} & : & (\neg 1 \land \neg 2 \land 8 \land 5 \land \neg 9 \land 11 \land 12 \land \cdots \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25)\lor \\
 & & (\neg 3 \land \neg 4 \land \neg 7 \land \neg 8 \land \neg 9 \land 10 \land 11 \land \cdots \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25) \\
\hline
f_1 \text{ for } \mathit{2typeH{+}Data} & : & \mathbf{1 \land 2 \land \neg 3 \land \neg 6 \land \neg 7 \land \neg 8 \land \neg 9 \land \neg 15 \land \neg 16 \land \cdots \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25}
\end{array}
$$

# Data-Driven Static Analysis

- Techniques
  - Learning via black-box optimization [OOPSLA'15]
  - Learning with disjunctive model [OOPSLA'17]
  - Learning with automatically generated features [OOPSLA'17]
  - Learning with supervison [ICSE'17,SAS'16,APLAS'16]
- Applications
  - context-sensitivity, flow-sensitivity, variable clustering, widening thresholds, unsoundness, search strategy in concolic testing, etc

# Learing via
# Black-Box Optimization
# (OOPSLA'15)

# Selective Flow-Sensitivity




FS : {x,y}

| | |
|---|---|
| x | [0,0] |
| y | [0,0] |

| | |
|---|---|
| x | [1,+∞] |
| y | [0,0] |

| | |
|---|---|
| x | [1,+∞] |
| y | [0,0] |

| | |
|---|---|
| x | [1,+∞] |
| y | [1,+∞] |

FI : {z}

| | |
|---|---|
| z | [1,+∞] |

# Static Analyzer

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

number of
proved assertions

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

# Overall Approach

- Parameterized heuristic

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

# Overall Approach

- Parameterized heuristic

$$H_W : \text{pgm} \rightarrow 2^{\text{Var}}$$

- Learn a good parameter *W* from existing codebase

$$\boxed{P_1, P_2, \ldots, P_m} \quad \Rightarrow \quad W$$

Codebase

# Overall Approach

- Parameterized heuristic

$$H_W : \text{pgm} \rightarrow 2^{\text{Var}}$$

- Learn a good parameter *W* from existing codebase

$$\boxed{P_1, P_2, \ldots, P_m} \quad \Rightarrow \quad W$$

Codebase

- For new program P, run static analysis with $H_W(\text{P})$

# 1. Parameterized Heuristic

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

(1) Represent program variables as feature vectors.

(2) Compute the score of each variable.

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

# (1) Features

- Predicates over variables:

$$f = \{f_1, f_2, \ldots, f_5\} \qquad (f_i : Var \rightarrow \{0,1\})$$

- We used 45 simple syntactic features for variables
  - e.g., local / global variable, passed to / returned from malloc, incremented by constants, etc

# (1) Features

- Represent each variable 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 variables:

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

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

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

# (3) Choose Top-k Variables

- Choose the top-k variables 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 variables with highest scores.

# 2. Learn a Good Parameter

$P_1, P_2, \ldots, P_m \Rightarrow w$

Codebase

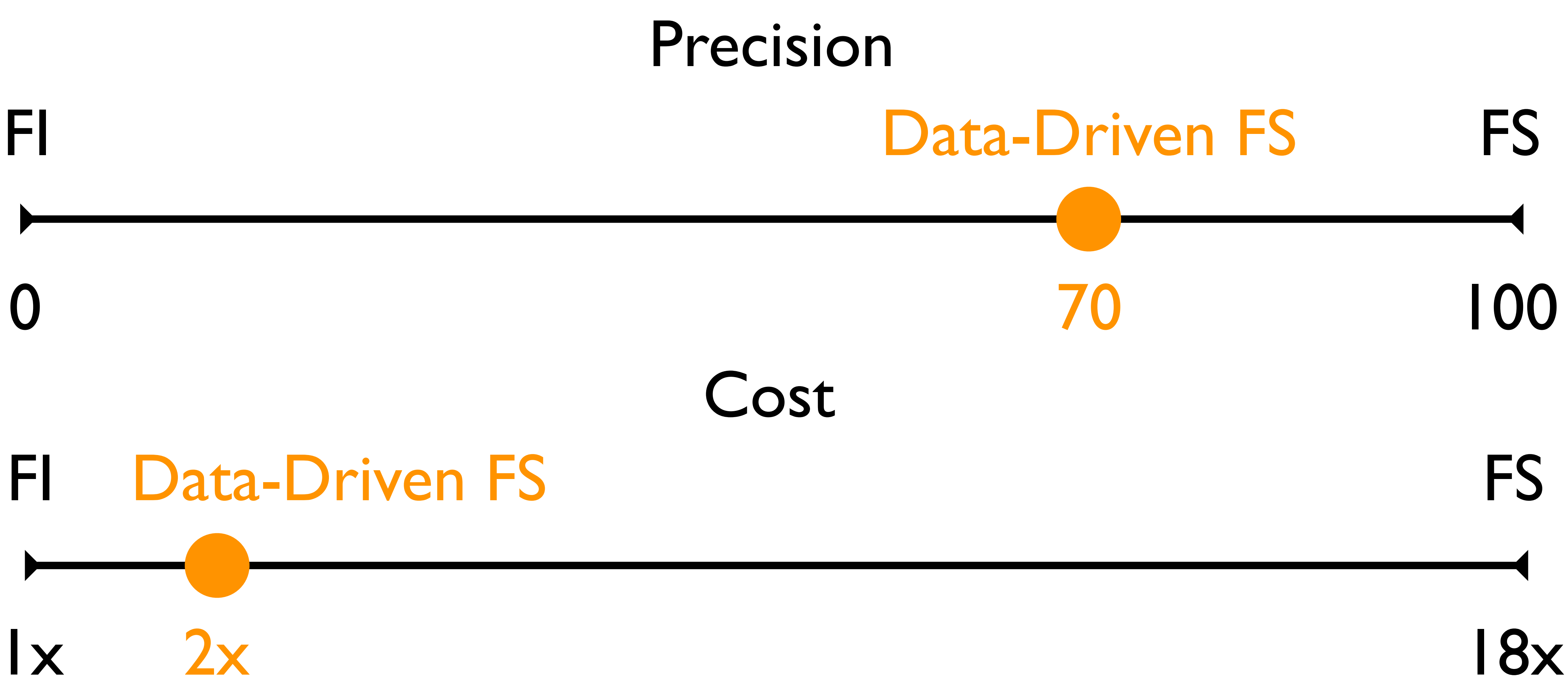
- Formulated as the optimization problem:

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

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

# Effectiveness on Sparrow

- Implemented in Sparrow, an interval analyzer for C
- Evaluated on 30 open-source programs
  - Training with 20 programs (12 hours)
  - Evaluation with the remaining 10 programs



# Limitations & Follow-ups

- Limited expressiveness due to linear heuristic
  - Disjunctive heuristic [OOPSLA'17]
- Semi-automatic due to manual feature engineering
  - Automated feature engineering [OOPSLA'17]
- High learning cost due to black-box approach
  - Supervised approaches [SAS'16,APLAS'16,ICSE'17]

OOPSLA'17

# Learning with Disjunctive Heuristics

- The linear heuristic cannot express disjunctive properties:

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

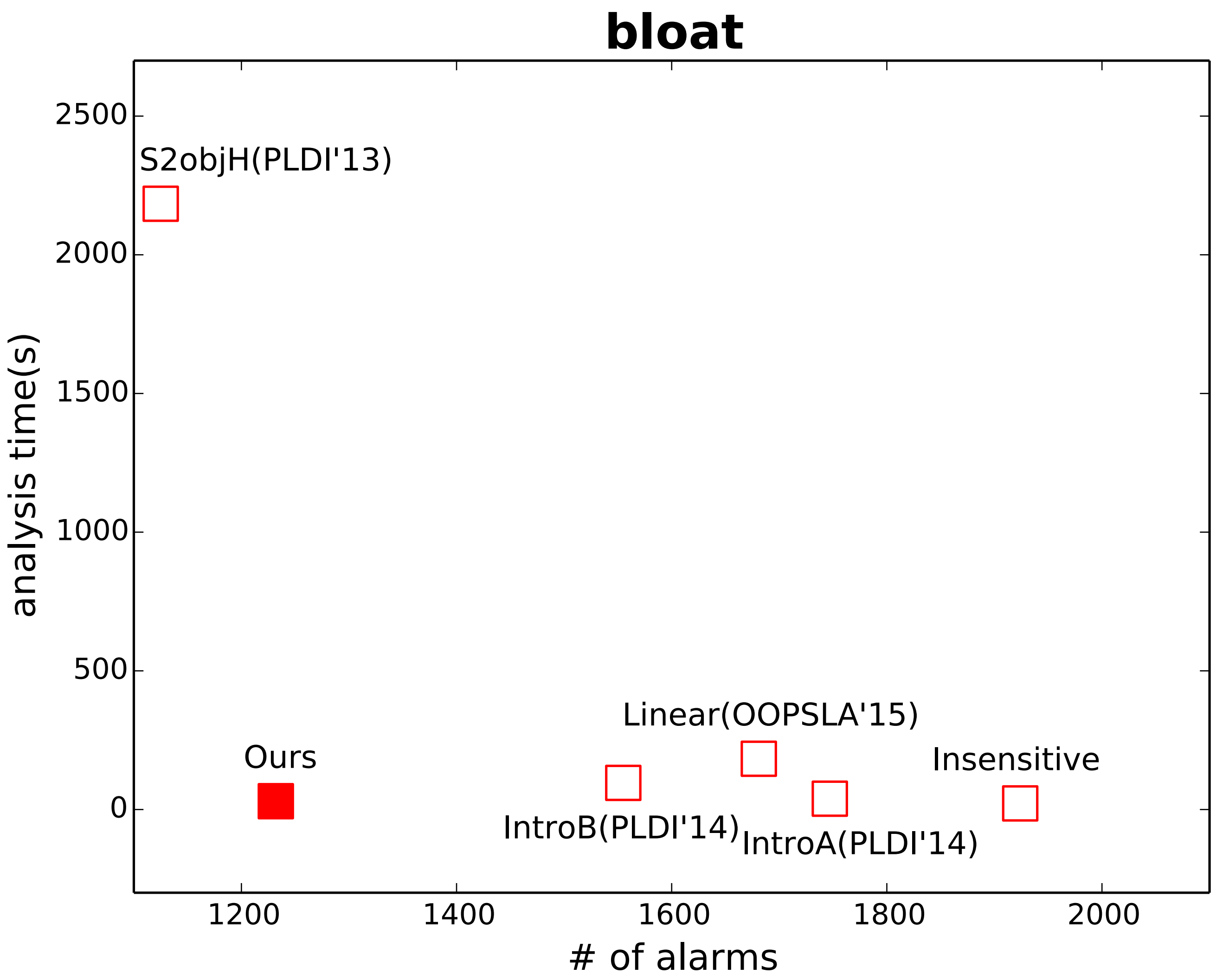
Goal: $\{x, w\}$

$$(a_1 \land a_2) \lor (\neg a_1 \land \neg a_2)$$

- Disjunctive heuristic + algorithm for learning boolean formulas

# Performance

- Applied to context-sensitive points-to analysis for Java
- Without disjunction, the learned heuristic lags behind hand-tuning because 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

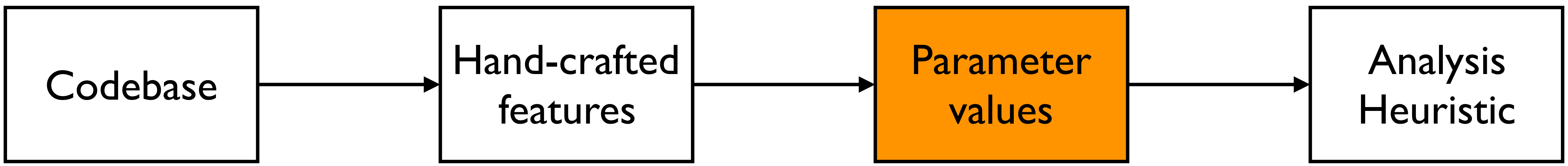
| Type | # | Features |
|---|---|---|
| A | 1 | used in array declarations (e.g., a[c]) |
| | 2 | used in memory allocation (e.g., malloc(c)) |
| | 3 | used in the righthand-side of an assignment (e.g., x = c) |
| | 4 | used with the less-than operator (e.g, x < c) |
| | 5 | used with the greater-than 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

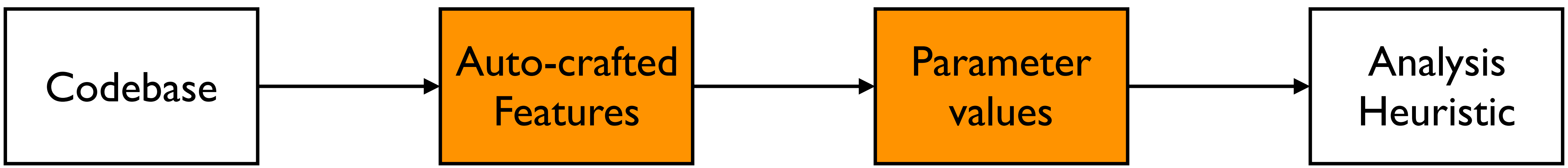
OOPSLA'17

# Automating Feature Engineering

Before (OOPSLA'15)

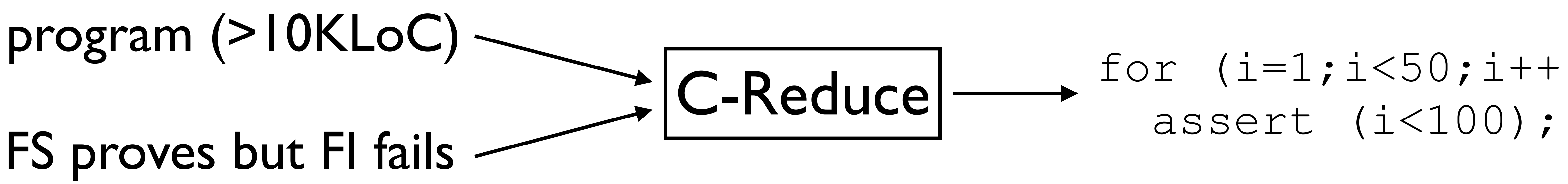



New method (OOPSLA'17)



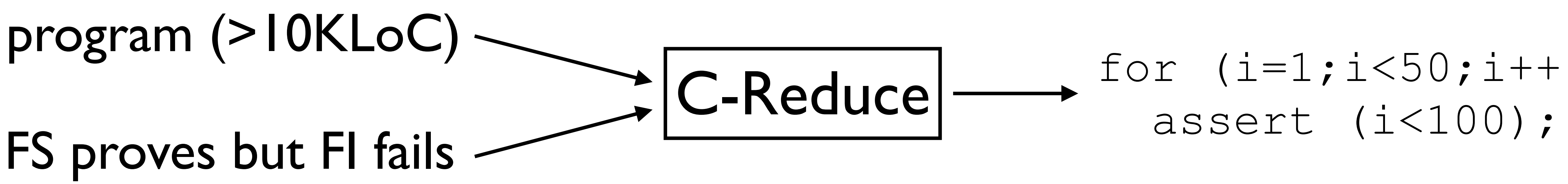
# Key Ideas

- Use program reducer to capture the key reason why FS succeeds but FI fails.

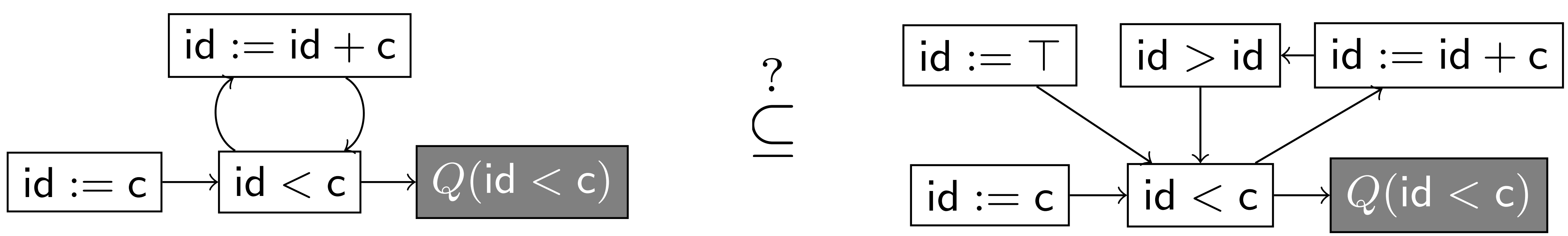


# Key Ideas

- Use program reducer to capture the key reason why FS succeeds but FI fails.




- Generalize the programs by abstract data flow graphs and check graph-inclusion




feature program

target program

# Data-Driven Concolic Testing
# (In submission)

# Concolic Testing

- 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? $(0 \leq x,y \leq 100)$

# 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%**

# 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? ($0 \le x,y \le 100$)

**< 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 |
|---|---|
| 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=22, y=7, z=14 | x=α, y=β, z=2*β |
| | 2*β ≠ α |

Solve: 2*β = α

Solution: α=2, β=1

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 |
|---|---|
| x=2, y=1, z=2 | x=α, y=β,z=2*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 |
|---|---|
| x=2, y=1, z=2 | $x=\alpha, y=\beta, z=2*\beta$ |
| | $2*\beta = \alpha$ |

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 | $x=\alpha, y=\beta, z=2*\beta$ |
| | $2*\beta = \alpha \wedge \alpha \leq \beta+10$ |

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=2, y=1, z=2 | x=α, y=β, z=2*β |
| | $2*\beta = \alpha \wedge \alpha \leq \beta+10$ |

Solve: $2*\beta = \alpha \wedge \alpha > \beta+10$

Solution: α=30, β=15

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

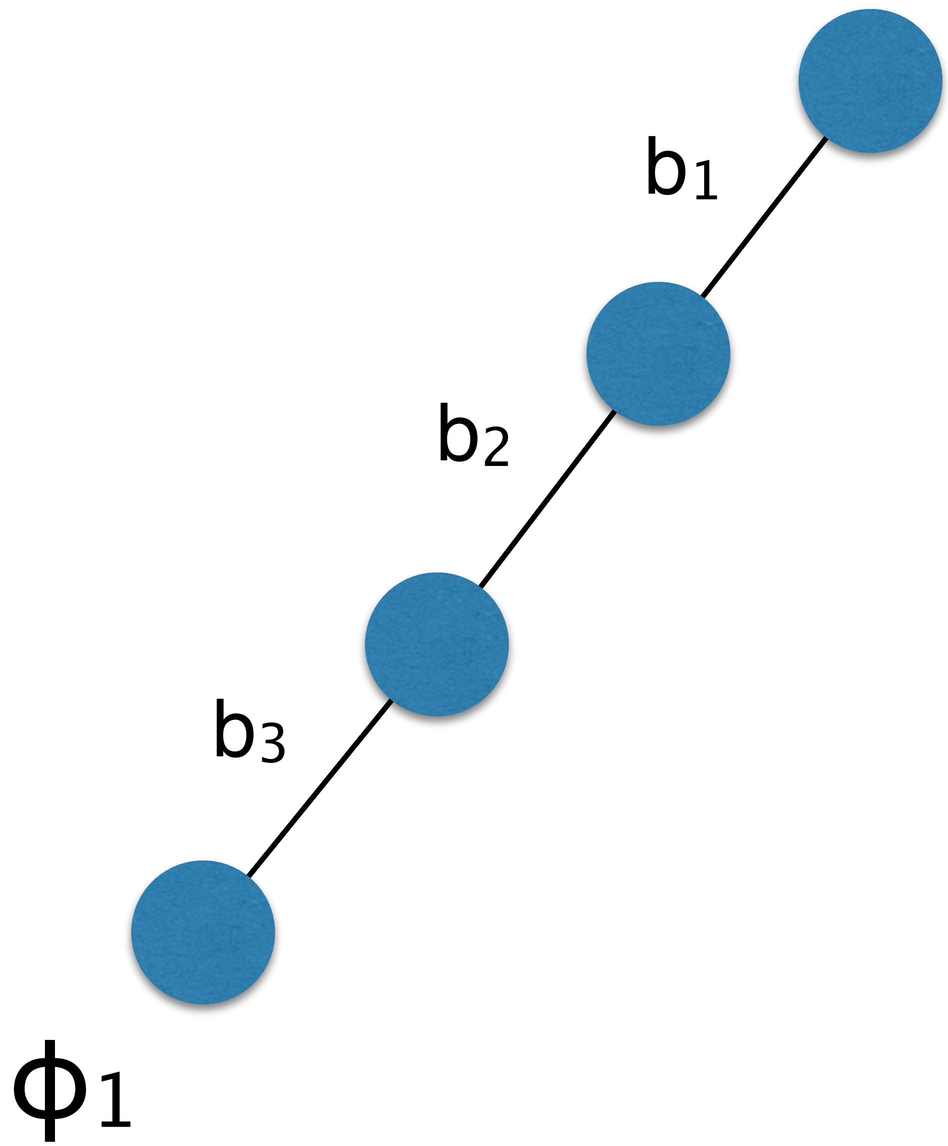
```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

error-triggering input

| Concrete State | Symbolic State |
|---|---|
| x=30, y=15, z=30 | x=α, y=β, z=2*β |
| | $2*\beta = \alpha \wedge$ $\alpha > \beta+15$ |

3rd iteration

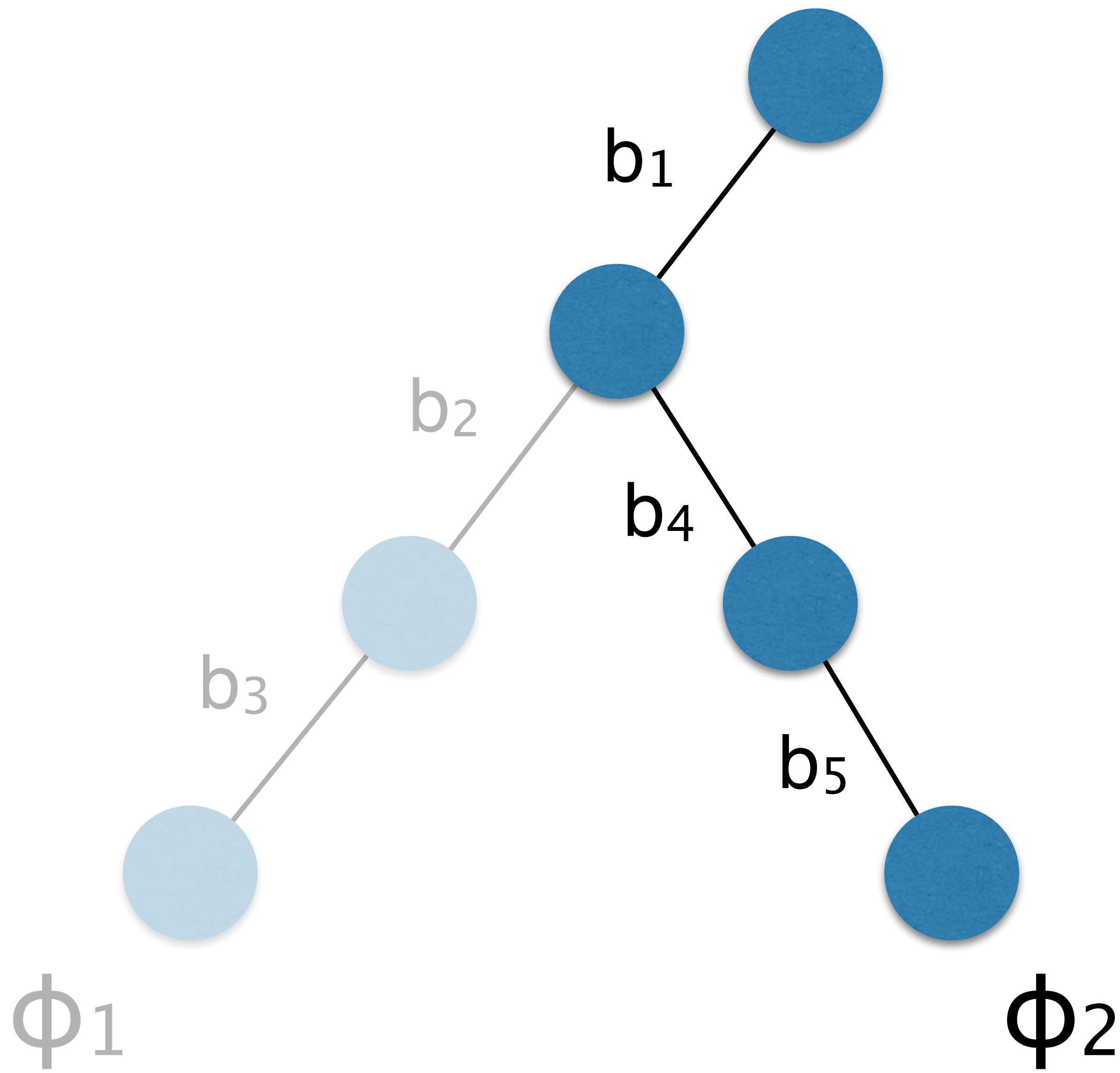
# Concolic Testing




solve $(b_1 \wedge \neg b_2)$

execution tree

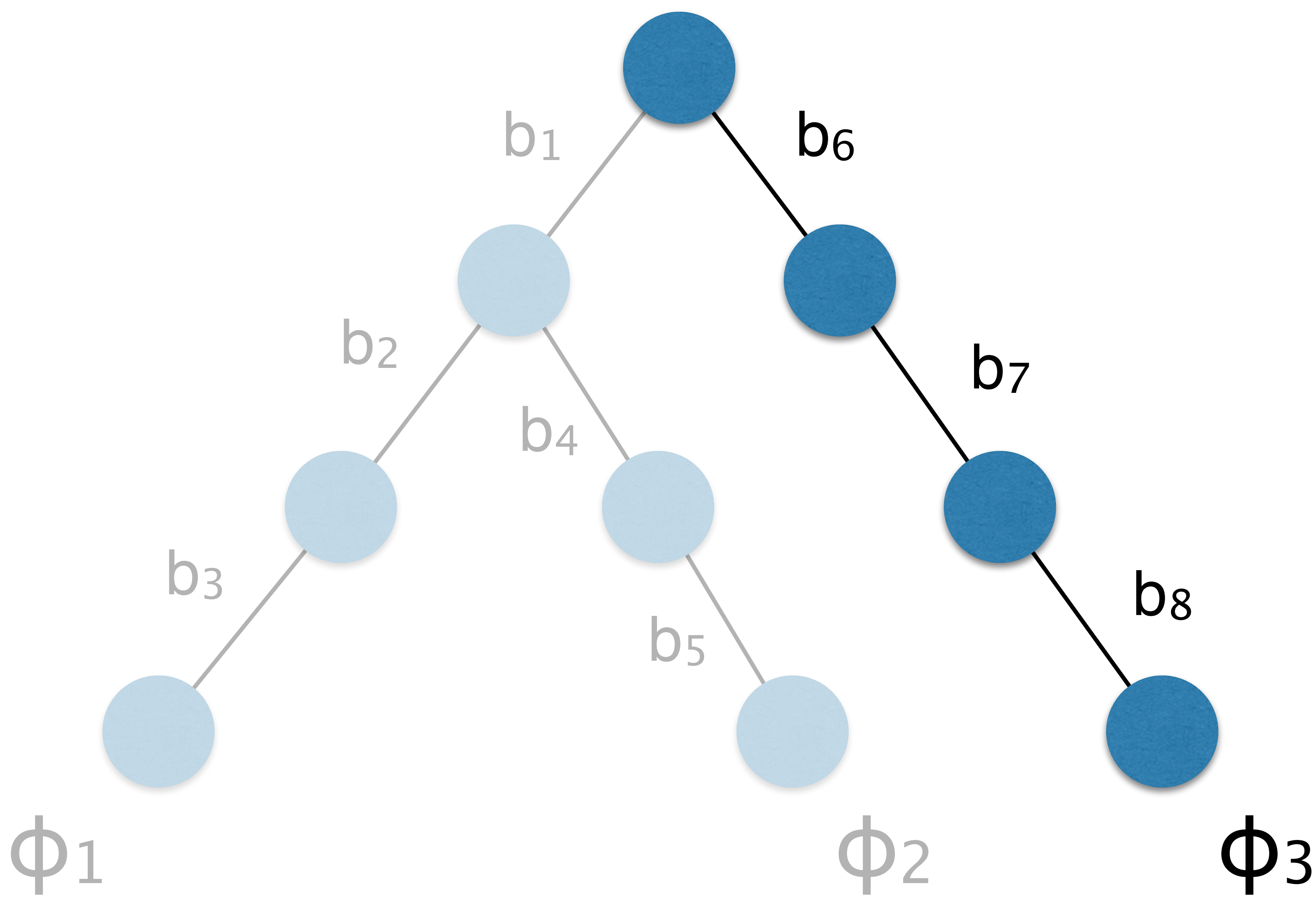
# Concolic Testing




solve ($\neg b_1$)

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

$T \leftarrow \langle\rangle$
$v \leftarrow v_0$
**for** $m = 1$ to $N$ **do**
  $\Phi_m \leftarrow \text{RunProgram}(P, v)$
  $T \leftarrow T \cdot \Phi_m$
  **repeat**
    $(\Phi, \phi_i) \leftarrow \text{Choose}(T)$    $(\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$
  **until** $\text{SAT}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
  $v \leftarrow \text{model}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
**end for**
**return** $|\text{Branches}(T)|$

# Concolic Testing Algorithm

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

$T \leftarrow \langle\rangle$
$v \leftarrow v_0$
**for** $m = 1$ to $N$ **do**
    $\Phi_m \leftarrow \text{RunProgram}(P$
    $T \leftarrow T \cdot \Phi_m$
    **repeat**
        $(\Phi, \phi_i) \leftarrow \text{Choose}(T)$    $(\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$
    **until** $\text{SAT}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
    $v \leftarrow \text{model}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
**end for**
**return** $|\text{Branches}(T)|$

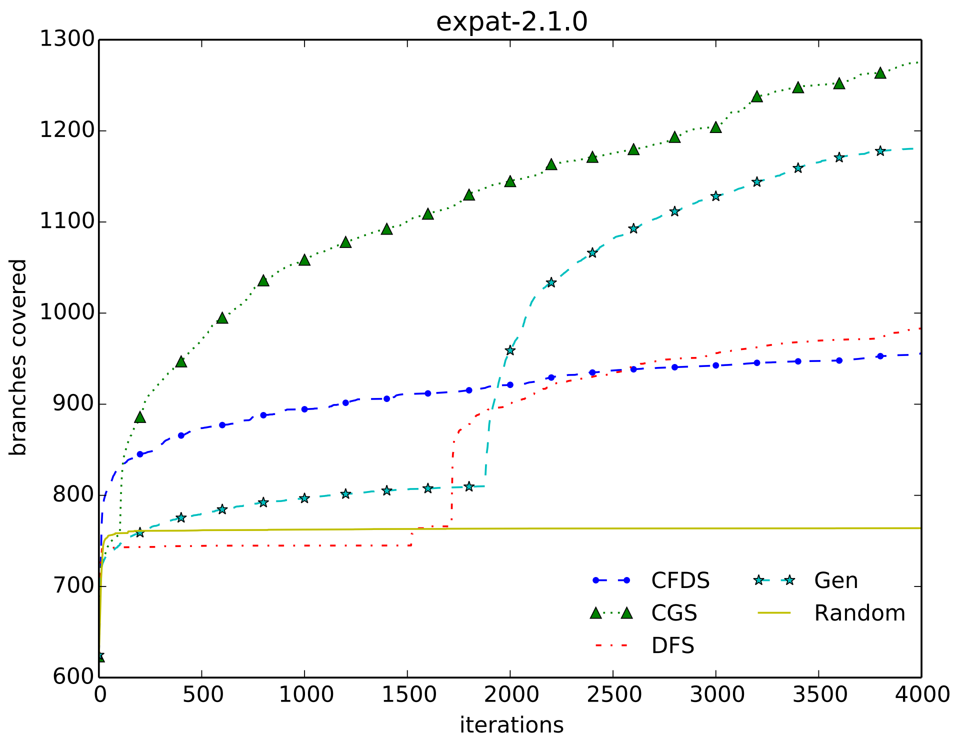
Search Heuristic

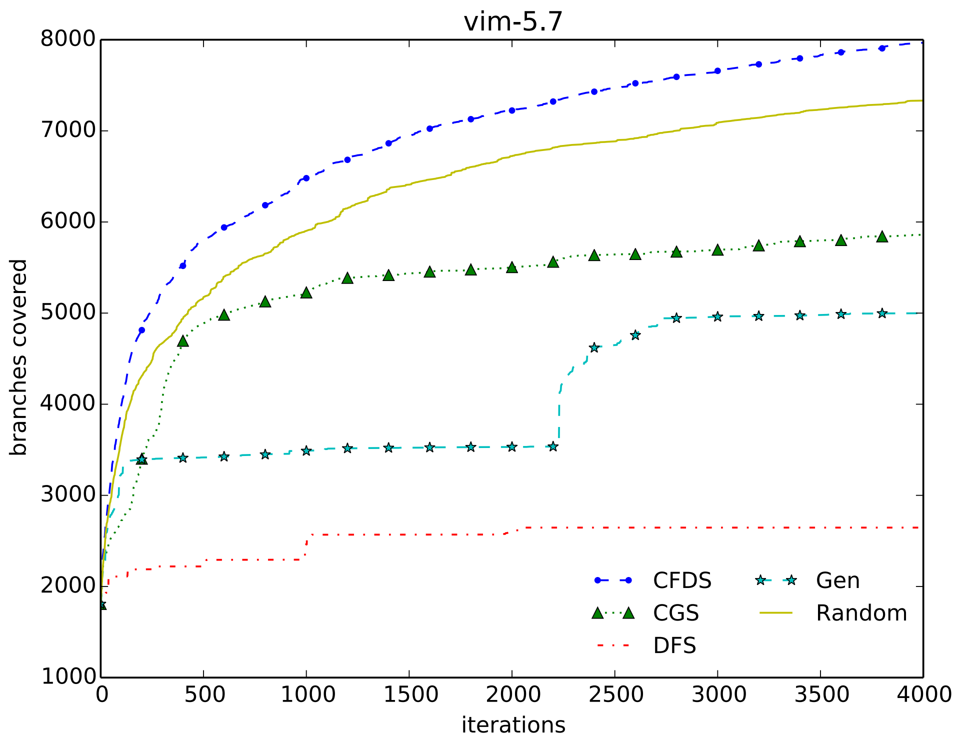
# Search Heuristics

- Concolic testing relies on search heuristics to maximize code coverage in a limited time budget.
- Key but the most manual and ad-hoc component of concolic testing
- Numerous heuristics have been proposed:
  - DFS [PLDI'05], BFS, Random, CFDS [ASE'08], Generational [NDSS'08], CarFast[FSE'12], CGS [FSE'14], …

# Limitations of Existing Search Heuristics

- No existing heuristics perform well in practice



# Limitations of Existing Search Heuristics

- Developing a heuristic requires a huge amount of engineering effort and expertise.


Heuristics for Scalable Dynamic Test Generation

ASE’08


CarFast: Achieving Higher Statement Coverage Faster

FSE’12


How We Get There: A Context-Guided Search Strategy in Concolic Testing

FSE’14


Automatically Generating Search Heuristics for Concolic Testing

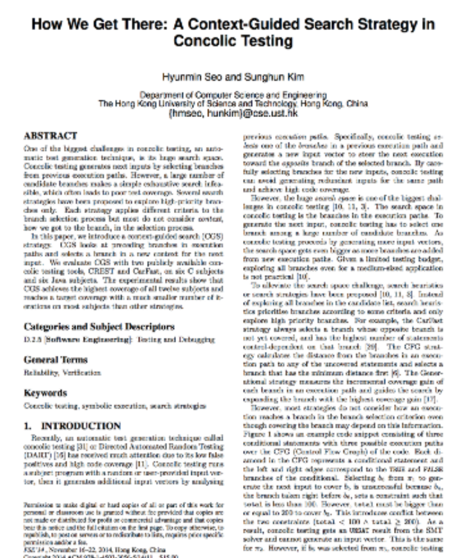
?

# Limitations of Existing Search Heuristics

- Developing a heuristic requires a huge amount of engineering effort and expertise.


ASE'08


FSE'12



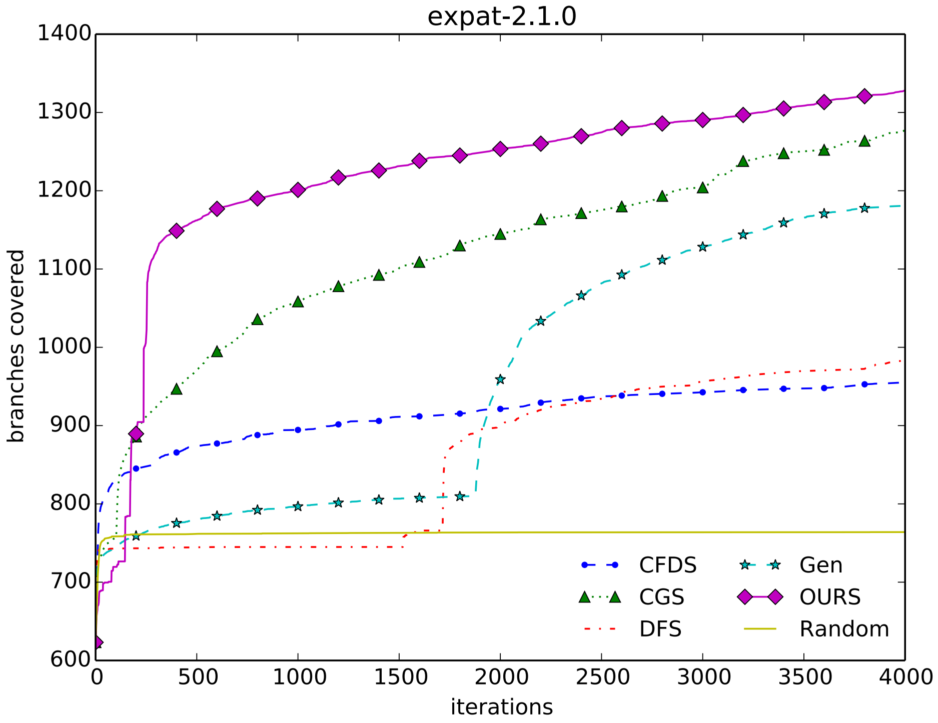

FSE'14

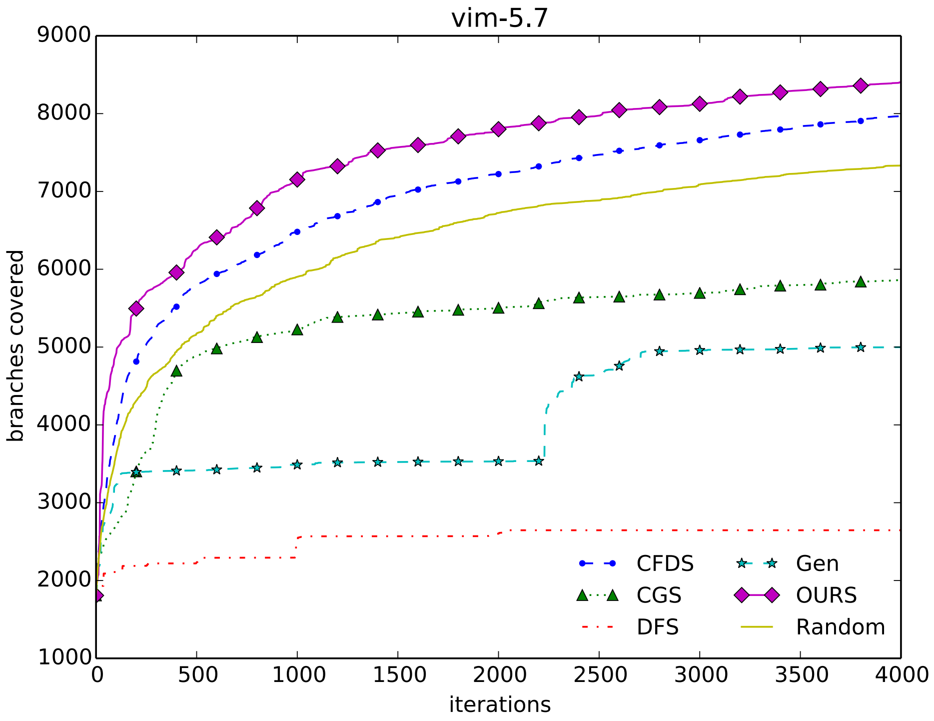

?

**Our goal: automatically generating search heuristics**

# Effectiveness

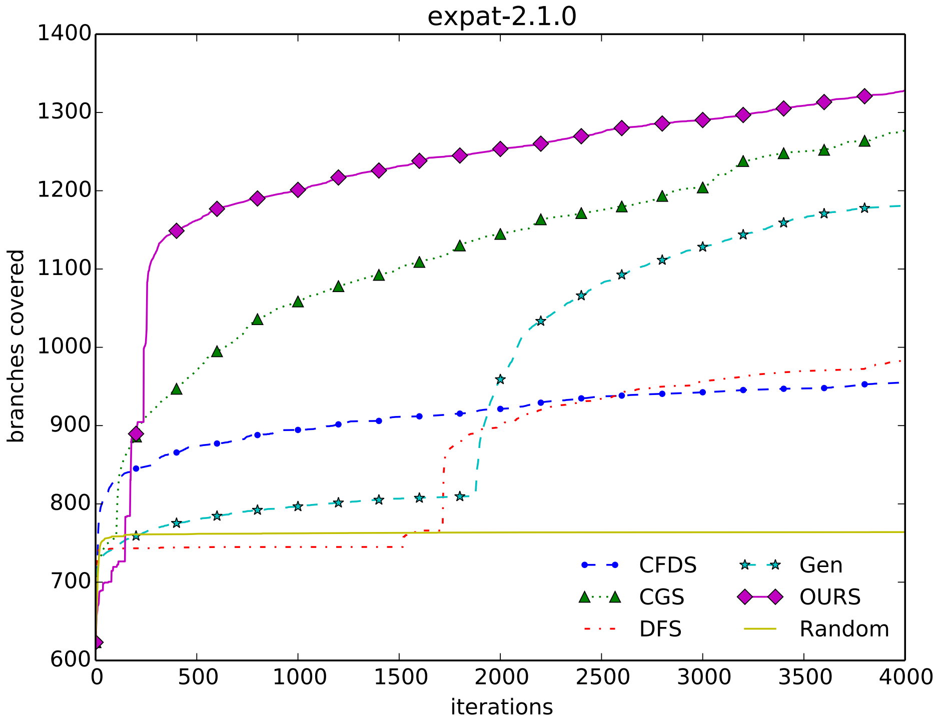
- Considerable increase in branch coverage

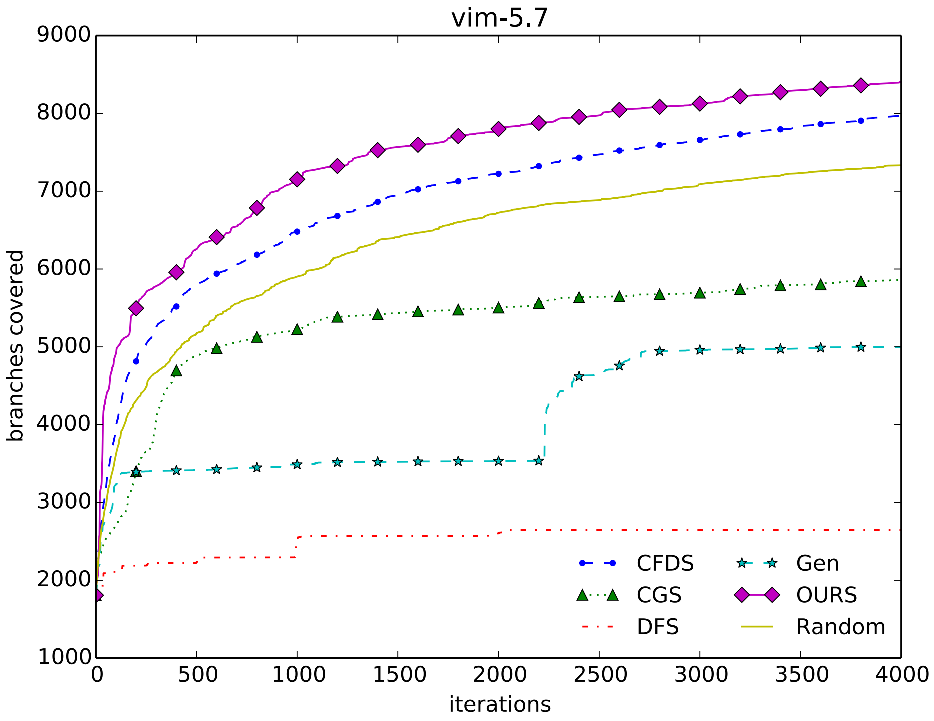


# Effectiveness

- Considerable increase in branch coverage



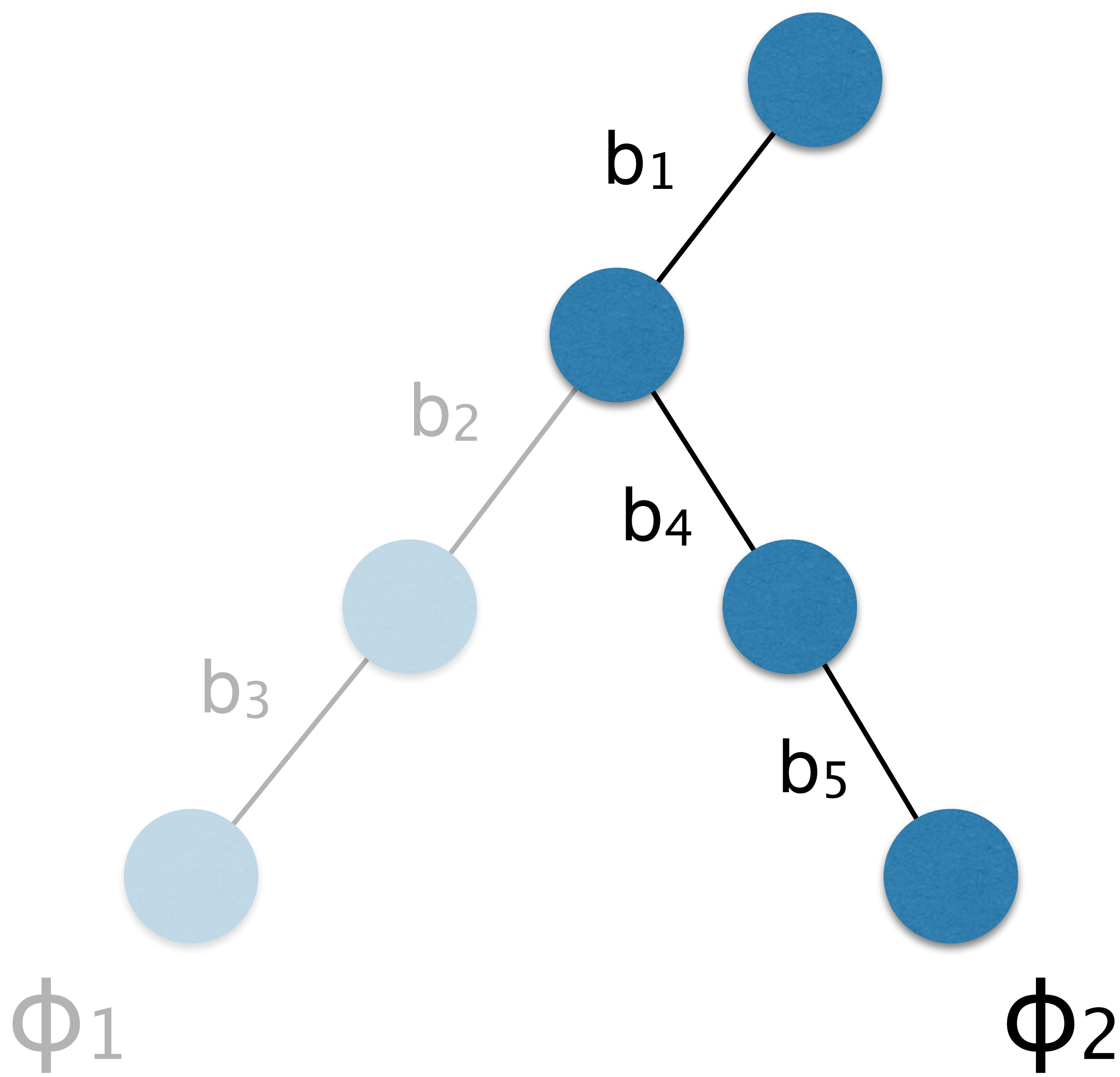




- Dramatic increase in bug-finding

| | **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 |

# Parameterized Search Heuristic

$$\text{Choose}_\theta(\langle \Phi_1 \cdots \Phi_m \rangle) = (\Phi_m, \operatorname*{argmax}_{\phi_j \in \Phi_m} \mathit{score}_\theta(\phi_j))$$




$score_\theta(b_1) = 1.3$

$score_\theta(b_4) = 0.0$

$score_\theta(b_5) = 0.7$

# (1) Feature Extraction

- A feature is a predicate on branches:

$$\pi_i : Branch \rightarrow \{0, 1\}$$

e.g., whether the branch is located in a loop

- Represent a branch by a feature vector

$$\pi(\phi) = \langle \pi_1(\phi), \pi_2(\phi), \ldots, \pi_k(\phi) \rangle$$

- Example

$\pi(b_1) = \langle 1,0,1,1,0 \rangle$

$\pi(b_4) = \langle 0,1,1,1,0 \rangle$

$\pi(b_5) = \langle 1,0,0,0,1 \rangle$

# Branch Features

- 12 static features
  - extracted without execution
- 28 dynamic features
  - extracted at runtime

| # | Description |
|---|---|
| 1 | branch in the main function |
| 2 | true branch of a loop |
| 3 | false branch of a loop |
| 4 | nested branch |
| 5 | branch containing external function calls |
| 6 | branch containing integer expressions |
| 7 | branch containing constant strings |
| 8 | branch containing pointer expressions |
| 9 | branch containing local variables |
| 10 | branch inside a loop body |
| 11 | true branch of a case statement |
| 12 | false branch of a case statement |
| 13 | first 10% branches of a path |
| 14 | last 10% branches of a path |
| 15 | branch appearing most frequently in a path |
| 16 | branch appearing least frequently in a path |
| 17 | branch newly covered in the previous execution |
| 18 | branch located right after the just-negated branch |
| 19 | branch whose context ($k = 1$) is already visited |
| 20 | branch whose context ($k = 2$) is already visited |
| 21 | branch whose context ($k = 3$) is already visited |
| 22 | branch whose context ($k = 4$) is already visited |
| 23 | branch whose context ($k = 5$) is already visited |
| 24 | branch negated more than 10 times |
| 25 | branch negated more than 20 times |
| 26 | branch negated more than 30 times |
| 27 | branch near the just-negated branch |
| 28 | branch failed to be negated more than 10 times |
| 29 | the opposite branch failed to be negated more than 10 times |
| 30 | the opposite branch is uncovered (depth 0) |
| 31 | the opposite branch is uncovered (depth 1) |
| 32 | branch negated in the last 10 executions |
| 33 | branch negated in the last 20 executions |

# (2) Scoring

- The parameter is a k-length vector of real numbers

$$\theta = \langle 0.8,-0.5,0.3,0.2,-0.7 \rangle$$

- Compute score by linear combination of feature vector and parameter

$$score_\theta(\phi) = \pi(\phi) \cdot \theta$$

$$score_\theta(b_1) = \langle 1,0,1,1,0 \rangle \cdot \langle 0.8,-0.5,0.3,0.2,-0.7 \rangle = 1.3$$

$$score_\theta(b_4) = \langle 0,1,1,1,0 \rangle \cdot \langle 0.8,-0.5,0.3,0.2,-0.7 \rangle = 0.0$$

$$score_\theta(b_5) = \langle 1,0,0,0,1 \rangle \cdot \langle 0.8,-0.5,0.3,0.2,-0.7 \rangle = 0.1$$

# Optimization Algorithm

- Finding a good search heuristic reduces to solving the optimization problem:

$$\operatorname*{argmax}_{\theta \in \mathbb{R}^k} C(P, \mathrm{Choose}_\theta)$$

where

$$C : \mathit{Program} \times \mathit{SearchHeuristic} \rightarrow \mathbb{N}$$

# Naive Algorithm

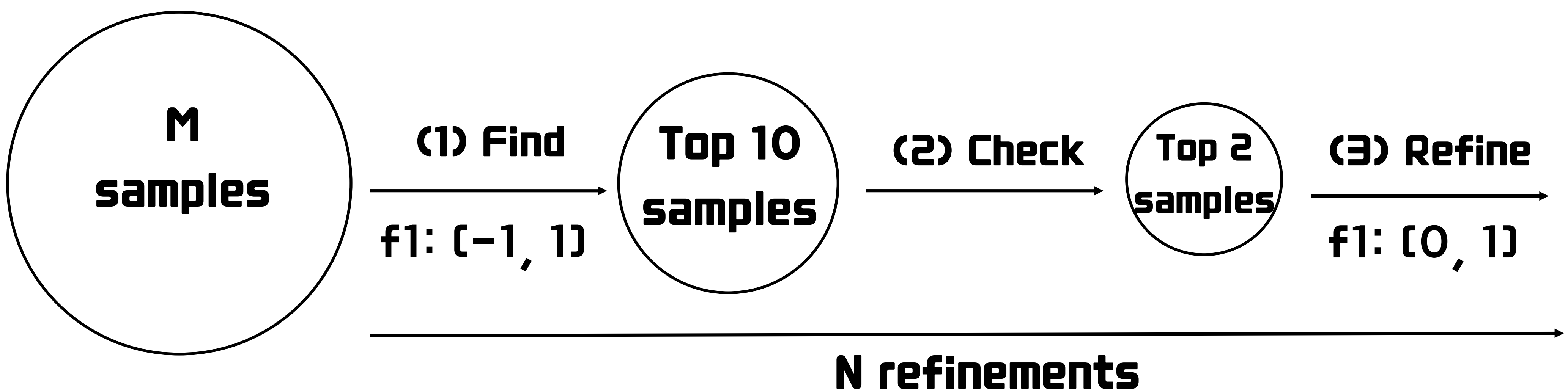
- Naive algorithm based on random sampling

```
repeat
    θ ← sample from ℝ^k
    B ← C(P, Choose_θ)
until timeout
return best θ found
```

- Failed to find good parameters
  - Search space is intractably large
  - Inherent performance variation in concolic testing

# Our Algorithm

- Iteratively refine the sample space based on the feedback from previous runs of concolic testing



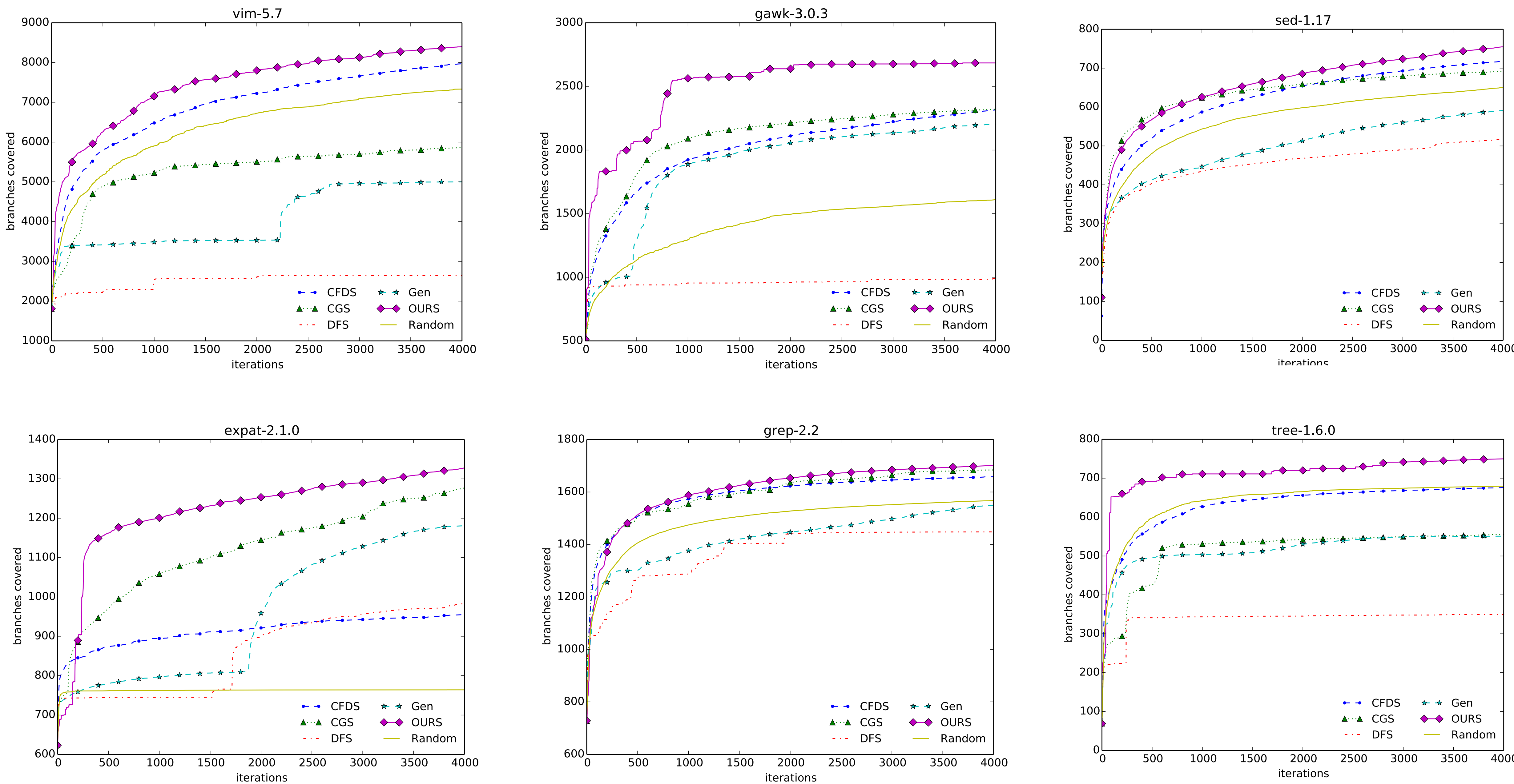
# Experiments

- Implemented in CREST
- Compared with five existing heuristics
  - CGS, CFDS, Random, DFS, Generational
- 10 open-source programs

| Program | # Total branches | LOC |
|---|---|---|
| vim-5.7 | 35,464 | 165K |
| gawk-3.0.3 | 8,038 | 30K |
| expat-2.1.0 | 8,500 | 49K |
| grep-2.2 | 3,836 | 15K |
| sed-1.17 | 2,565 | 9K |
| tree-1.6.0 | 1,438 | 4K |
| cdaudio | 358 | 3K |
| floppy | 268 | 2K |
| kbfiltr | 204 | 1K |
| replace | 196 | 0.5K |

# Effectiveness

- Average branch coverage (on large programs)



# Effectiveness

- Maximum branch coverage

| | **OURS** | CFDS | CGS | Random | Gen | DFS |
|---|---|---|---|---|---|---|
| vim | **8,744** | 8,322 | 6,150 | 7,645 | 5,092 | 2,646 |
| expat | **1,422** | 1,060 | 1,337 | 965 | 1,348 | 1,027 |
| gawk | **2,684** | 2,532 | 2,449 | 2,035 | 2,443 | 1,025 |
| grep | **1,807** | 1,726 | 1,751 | 1,598 | 1,640 | 1,456 |
| sed | **830** | 780 | 781 | 690 | 698 | 568 |
| tree | **797** | 702 | 599 | 704 | 600 | 360 |

- On small benchmarks

| | **OURS** | CFDS | **CGS** | Random | Gen | DFS |
|---|---|---|---|---|---|---|
| cdaudio | **250** | **250** | **250** | 242 | 236 | **250** |
| floppy | **205** | **205** | **205** | 170 | 168 | **205** |
| replace | **181** | 177 | **181** | 174 | 171 | 176 |
| kbfiltr | **149** | **149** | **149** | **149** | 134 | **149** |

# Effectiveness

- Higher branch coverage leads to much more effective finding of real bugs

| | **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 |

- Our heuristics are much better than others in exercising diverse program paths

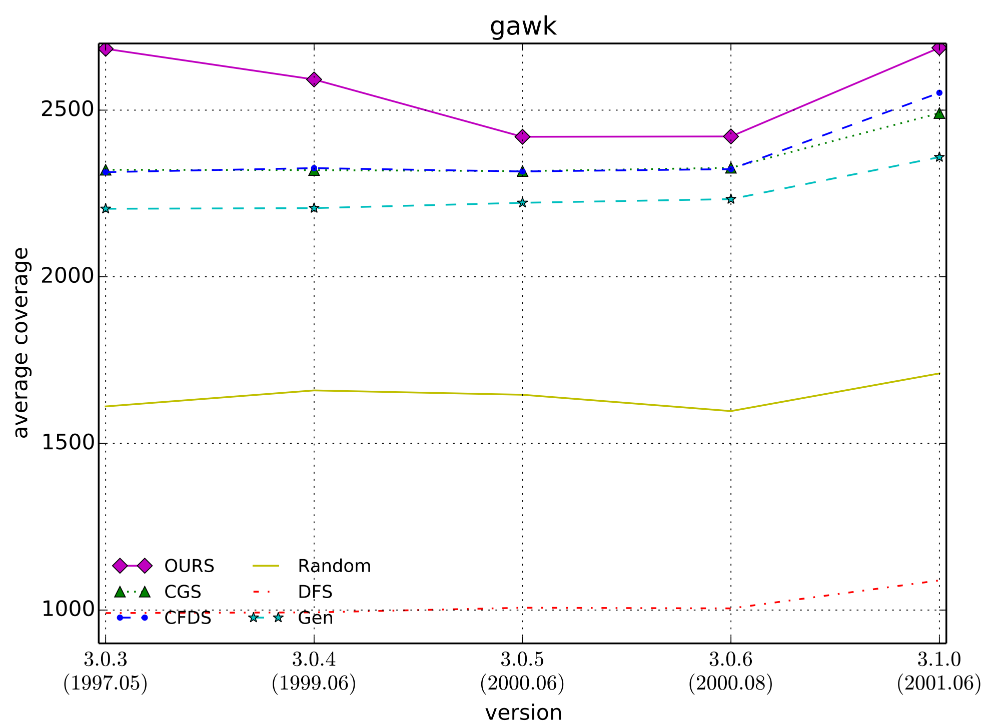
# Training Overhead

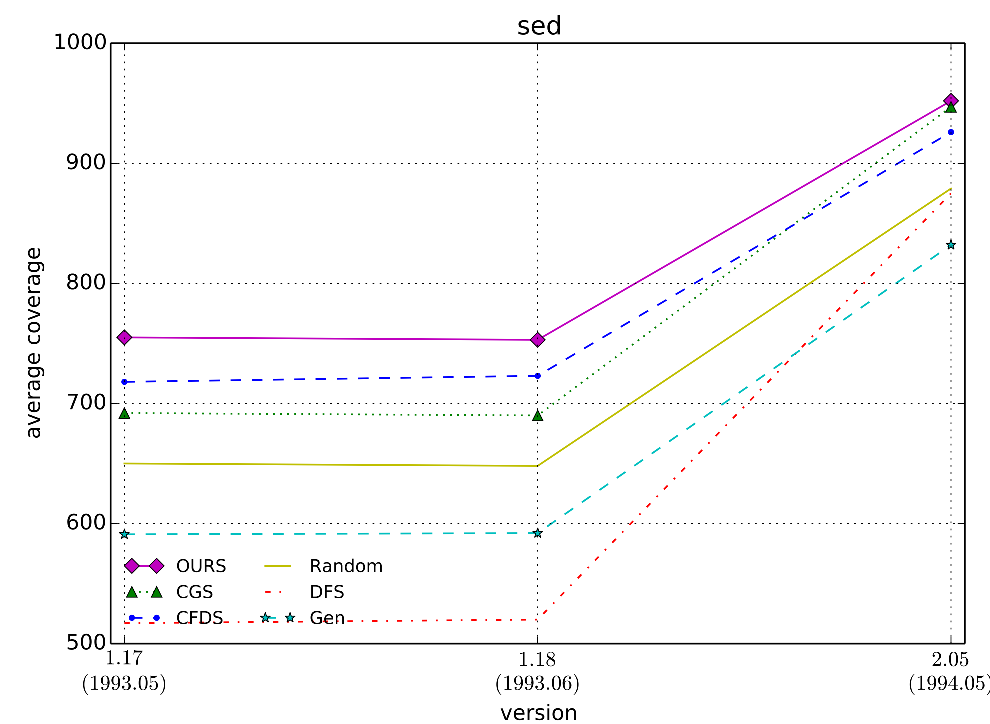
- Time for obtaining the heuristics (with 20 cores)

| Benchmarks | # Sample | # Iteration | Total times |
|---|---:|---:|---:|
| vim-5.7 | 300 | 5 | 24h 18min |
| expat-2.1.0 | 1,000 | 6 | 10h 25min |
| gawk-3.0.3 | 1,000 | 4 | 6h 30min |
| grep-2.2 | 1,000 | 5 | 5h 24min |
| sed-1.17 | 1,000 | 4 | 8h 54min |
| tree-1.6.0 | 1,000 | 4 | 3h 18min |

# Still useful

- Reusable as programs evolve




- Concolic testing is run in the training phase

| | **OURS** | CFDS | CGS | Random | Gen | DFS |
|---|---|---|---|---|---|---|
| vim | **14,003** | 13,706 | 7,934 | 13,835 | 7,290 | 7,934 |
| expat | **2,455** | 2,339 | 2,157 | 1,325 | 2,116 | 2,036 |
| gawk | **3,473** | 3,382 | 3,261 | 3,367 | 3,302 | 1,905 |
| grep | **2,167** | 2,024 | 2,016 | 2,066 | 1,965 | 1,478 |
| sed | 1,019 | 1,041 | **1,042** | 1,007 | 979 | 937 |
| tree | **808** | 800 | 737 | 796 | 730 | 665 |

# Summary

- **Problem**: Heuristic decisions in program analysis
- **Approach**: Use data to make heuristic decisions
- **Finding**: Machine-tuning outperforms hand-tuning
- Still on-going: …

# Summary

- **Problem**: Heuristic decisions in program analysis
- **Approach**: Use data to make heuristic decisions
- **Finding**: Machine-tuning outperforms hand-tuning
- Still on-going: …

Thank you