

# 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



**WALA**  
T. J. WATSON LIBRARIES FOR ANALYSIS



**Astrée**

**DOOP**

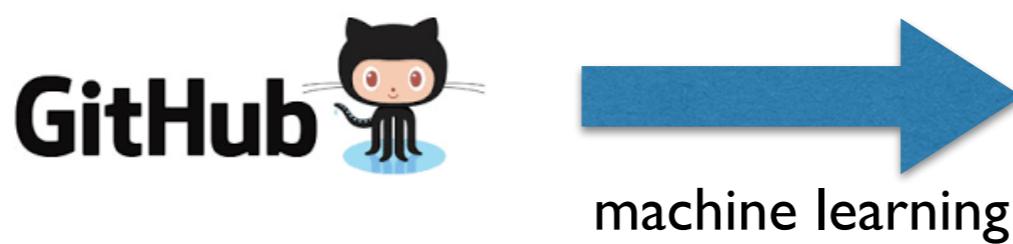
**TAJS**

**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

# Automatically Generating Heuristics from Data

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



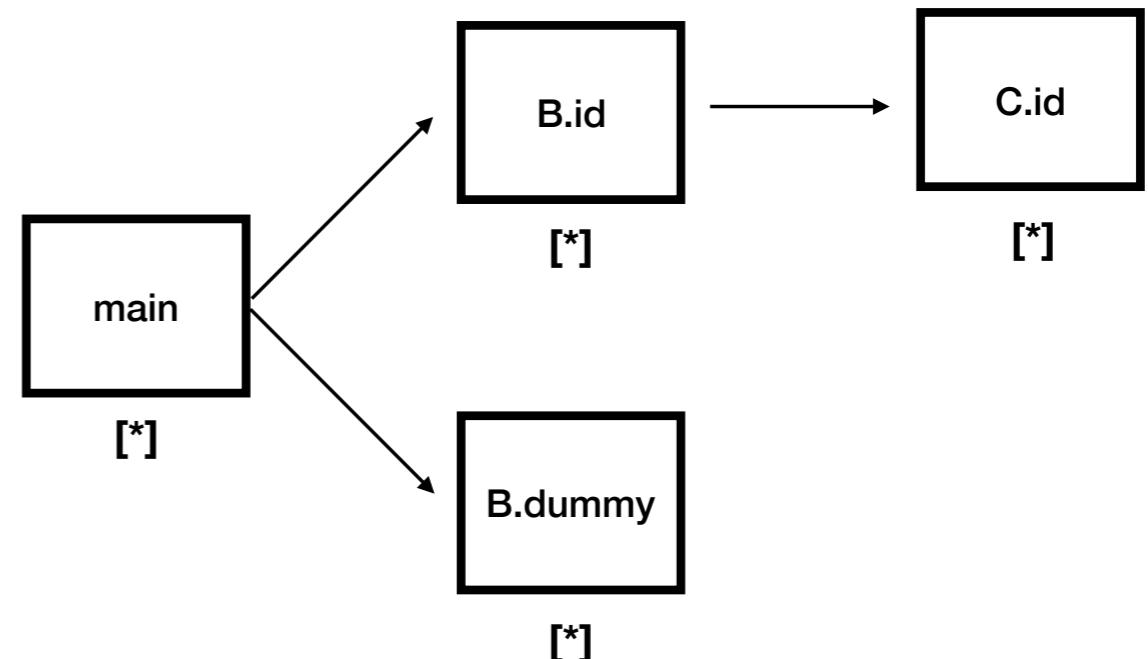
context-sensitivity heuristics  
flow-sensitivity heuristics  
unsoundness heuristics  
...

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

# Context-Sensitivity

```
1: class D{} class E{}
2:
3: class C{
4: Object id(Object v){return v;}}
5:
6: class B{
7: void dummy(){}
8: Object id(Object v){
9: C c = new C(); //C1
10: return c.id(v);}}
11:
12: class A{
13: public static void main(String[] args){
14: B b1 = new B(); //B1
15: B b2 = new B(); //B2
16: D d = (D) b1.id1(new D()); //query1
17: E e = (E) b2.id1(new E()); //query2
18: b1.dummy();
19: b2.dummy();}}
```

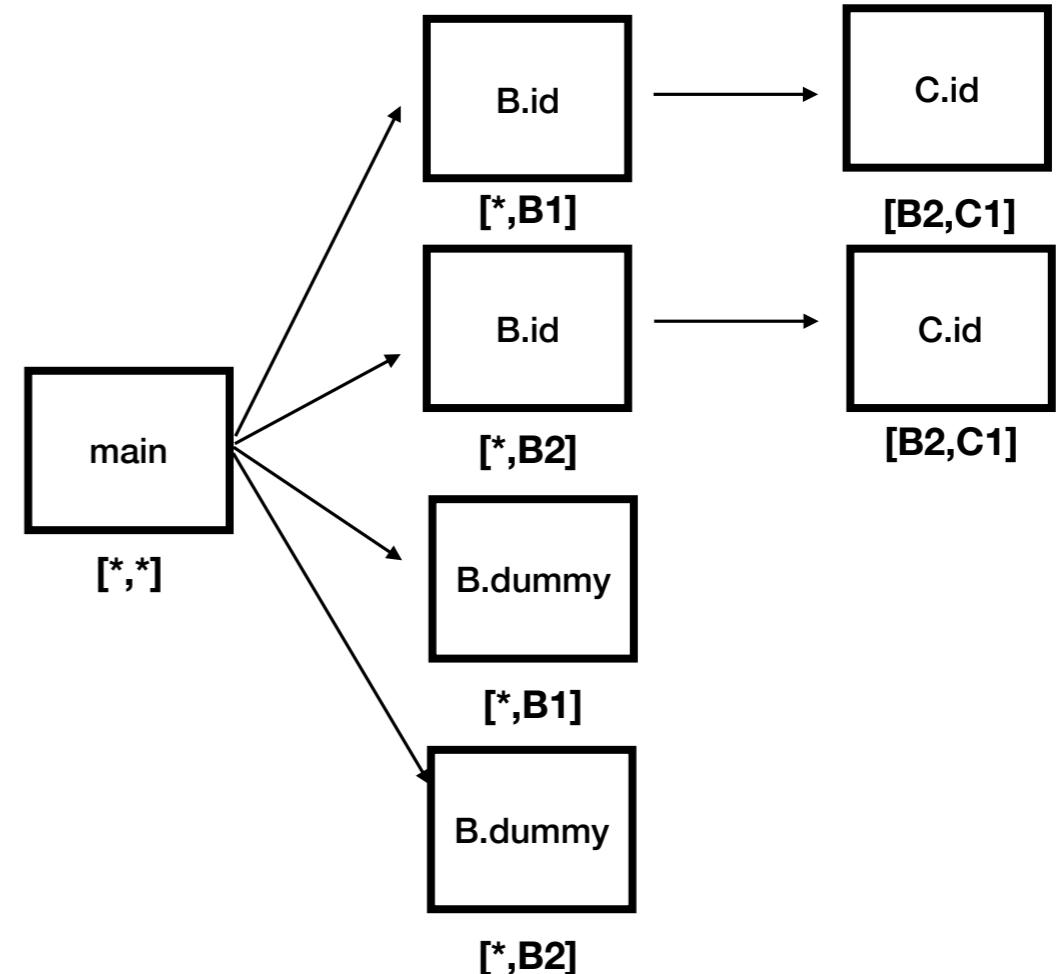
Without context-sensitivity,  
analysis fails to prove queries



# Context-Sensitivity

```
1: class D{} class E{}
2:
3: class C{
4: Object id(Object v){return v;}}
5:
6: class B{
7: void dummy(){}
8: Object id(Object v){
9: C c = new C(); //C1
10: return c.id(v);}}
11:
12: class A{
13: public static void main(String[] args){
14: B b1 = new B(); //B1
15: B b2 = new B(); //B2
16: D d = (D) b1.id1(new D()); //query1
17: E e = (E) b2.id1(new E()); //query2
18: b1.dummy();
19: b2.dummy();}}
```

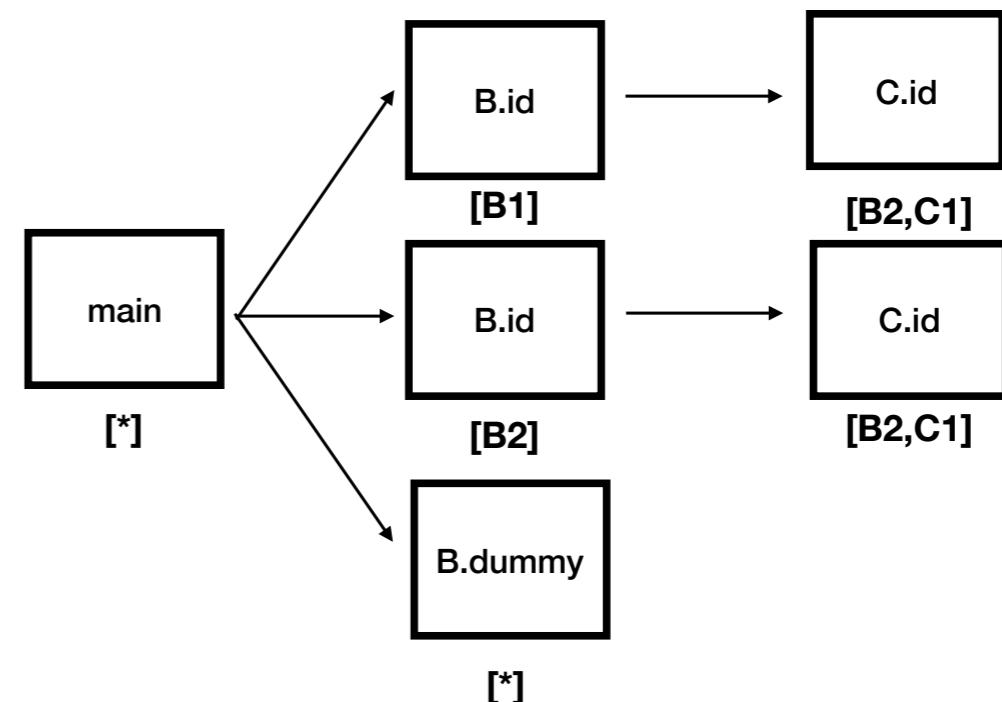
2-object-sensitivity succeeds  
but does not scale



# Selective Context-Sensitivity

```
1: class D{} class E{}
2:
3: class C{
4: Object id(Object v){return v;}}
5:
6: class B{
7: void dummy(){}
8: Object id(Object v){
9: C c = new C(); //C1
10: return c.id(v);}}
11:
12: class A{
13: public static void main(String[] args){
14: B b1 = new B(); //B1
15: B b2 = new B(); //B2
16: D d = (D) b1.id1(new D()); //query1
17: E e = (E) b2.id1(new E()); //query2
18: b1.dummy();
19: b2.dummy();}}
```

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

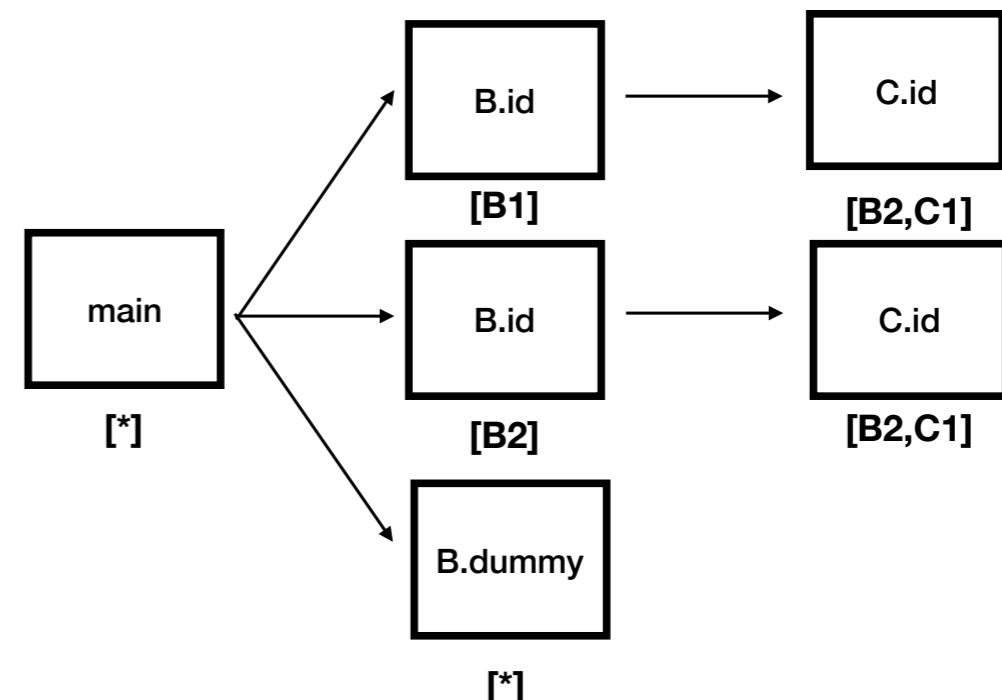


# Selective Context-Sensitivity

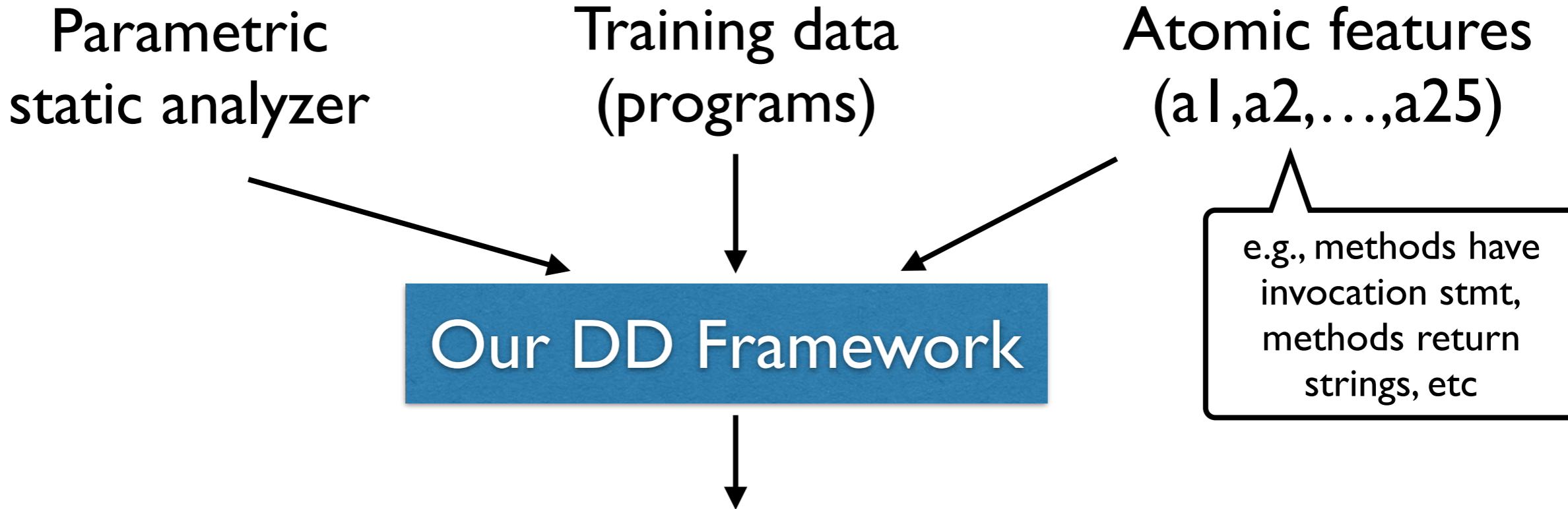
Challenge: How to decide? **Data-driven approach**

```
1: class D{} class E{}
2:
3: class C{
4: Object id(Object v){return v;}}
5:
6: class B{
7: void dummy(){}
8: Object id(Object v){
9: C c = new C(); //C1
10: return c.id(v);}}
11:
12: class A{
13: public static void main(String[] args){
14: B b1 = new B(); //B1
15: B b2 = new B(); //B2
16: D d = (D) b1.id1(new D()); //query1
17: E e = (E) b2.id1(new E()); //query2
18: b1.dummy();
19: b2.dummy();}}
```

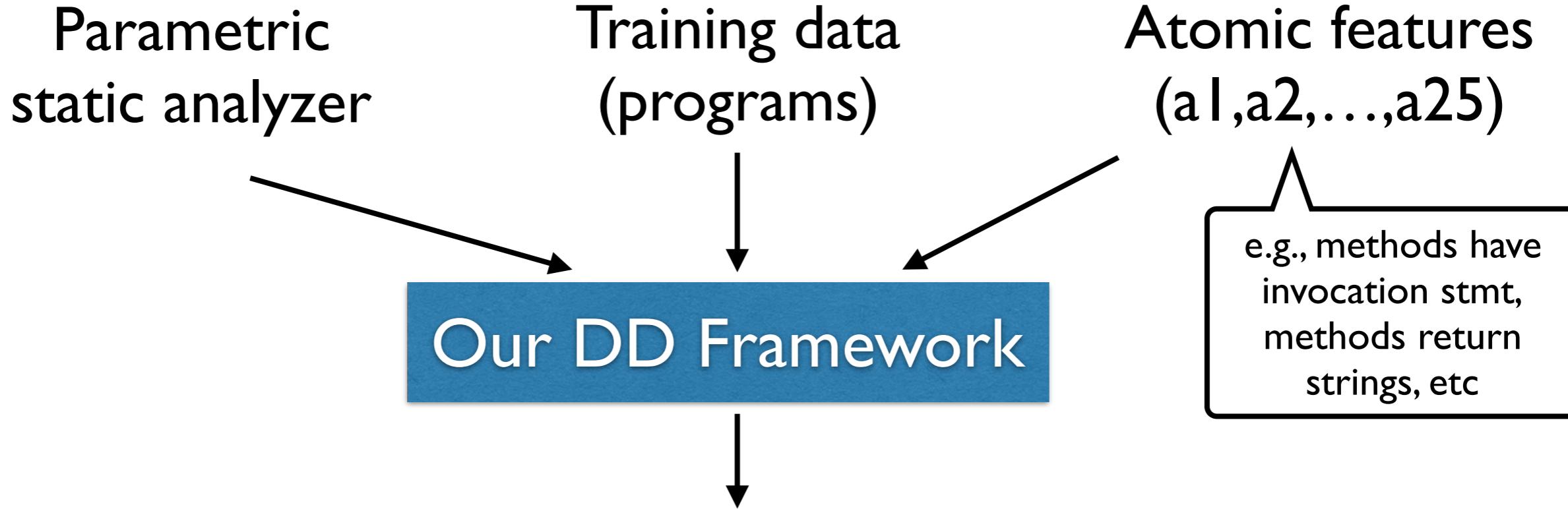
Apply 2-obj-sens: {C.id}  
Apply 1-obj-sens: {B.id}  
Apply insens: {B.m}



# Data-Driven Ctx-Sensitivity



# Data-Driven Ctx-Sensitivity



**Heuristic for applying (hybrid) object-sensitivity:**

f2: Methods that require 2-object-sensitivity

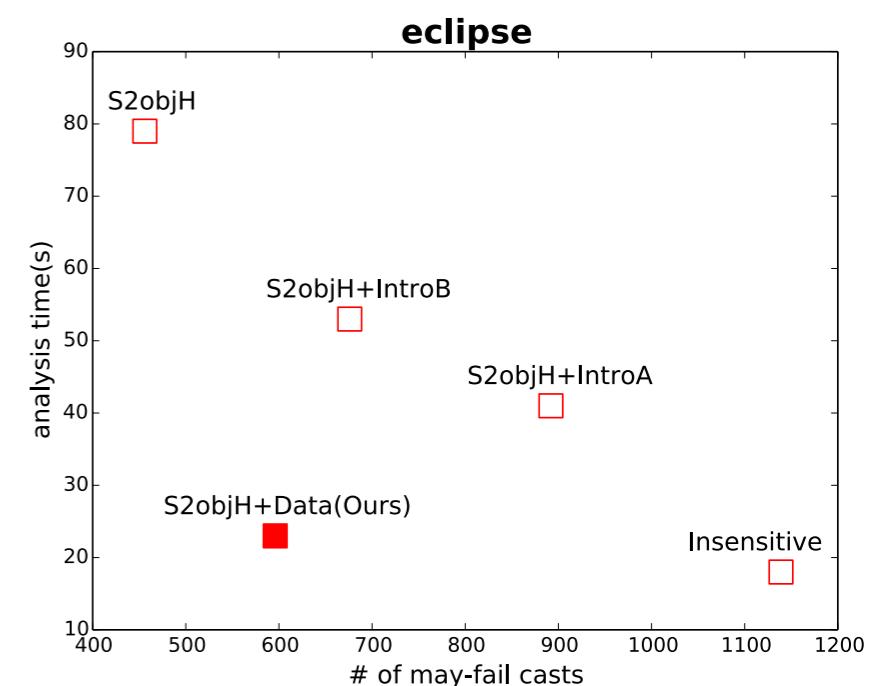
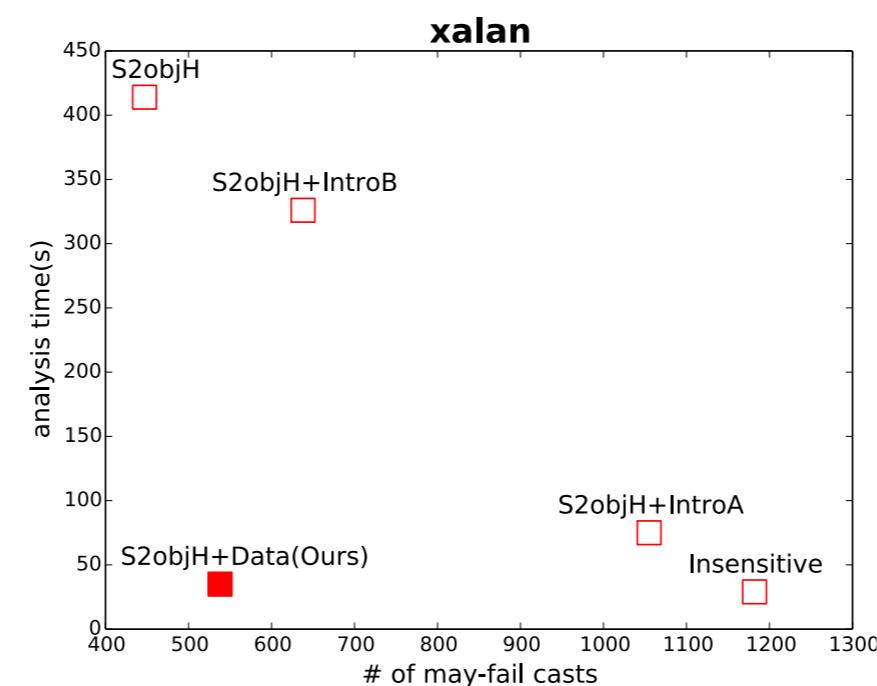
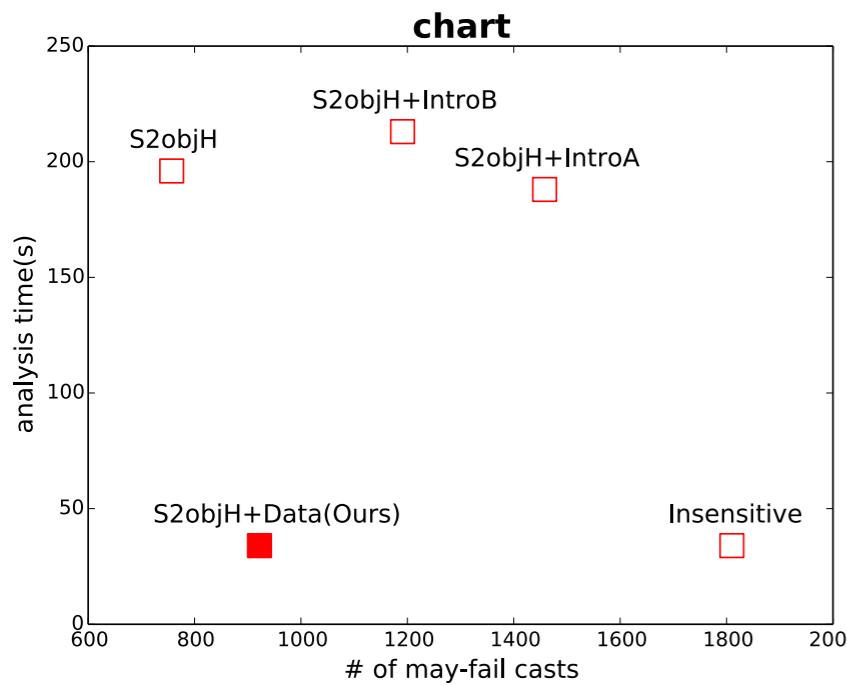
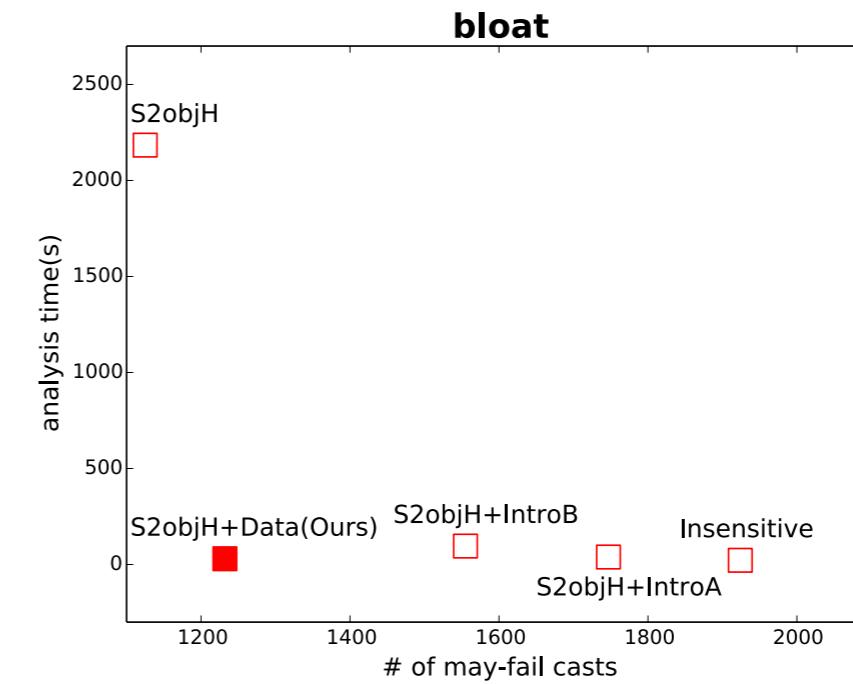
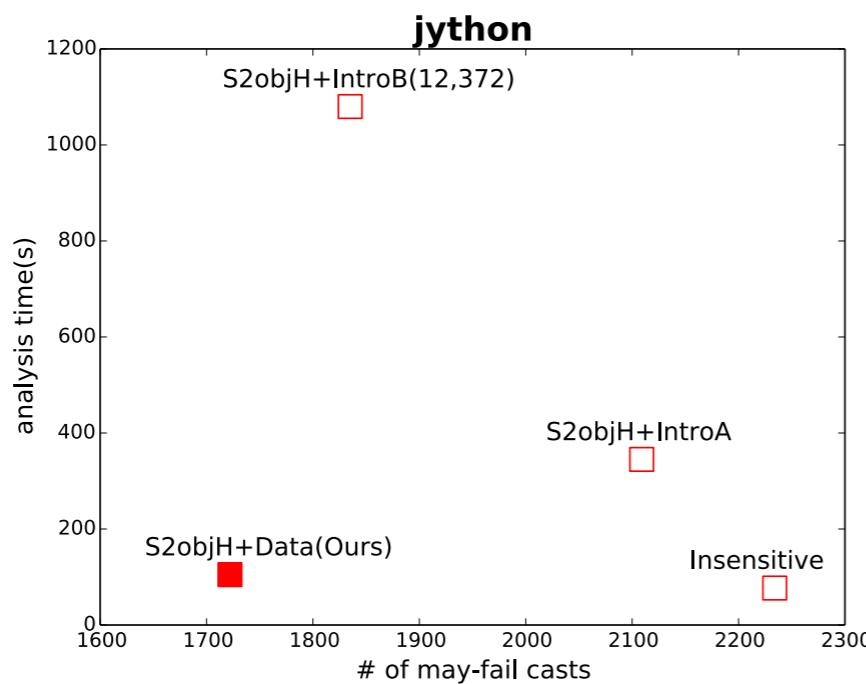
$$1 \wedge \neg 3 \wedge \neg 6 \wedge 8 \wedge \neg 9 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25$$

f1: Methods that require 1-object-sensitivity

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

# 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 ( $f_2$ ):

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

- Depth-1 formula ( $f_1$ ):

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

- Call-site-sensitivity:

- Depth-2 formula ( $f_2$ ):

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

- Depth-1 formula ( $f_1$ ):

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

- Type-sensitivity:

- Depth-2 formula ( $f_2$ ):

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

- Depth-1 formula ( $f_1$ ):

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

# 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:

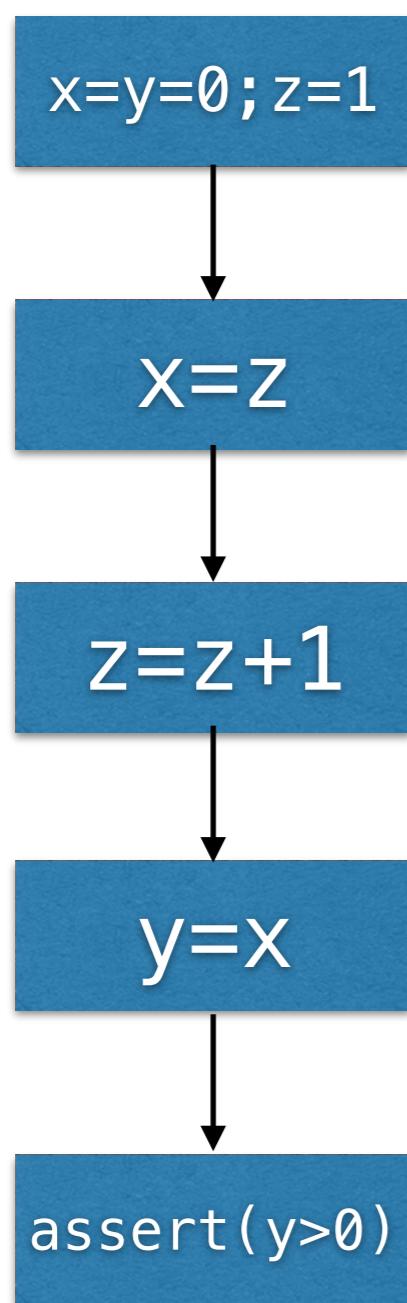
$$\begin{array}{lcl} f_1 \text{ for } 2objH+Data & : & (1 \wedge 2 \wedge \neg 3 \wedge \neg 6 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge \neg 16 \wedge \dots \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee \\ & & (\neg 1 \wedge \neg 2 \wedge 8 \wedge 5 \wedge \neg 9 \wedge 11 \wedge 12 \wedge \dots \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee \\ & & (\neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge 10 \wedge 11 \wedge \dots \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \\ \hline f_1 \text{ for } 2typeH+Data & : & 1 \wedge 2 \wedge \neg 3 \wedge \neg 6 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge \neg 15 \wedge \neg 16 \wedge \dots \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25 \end{array}$$

# 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 supervision [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	[l, +∞]
y	[0,0]

x	[l, +∞]
y	[0,0]

x	[l, +∞]
y	[l, +∞]

FI : {z}

z	[l, +∞]
---	---------

# Static Analyzer

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

number of  
proved assertions

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

# Overall Approach

- Parameterized heuristic

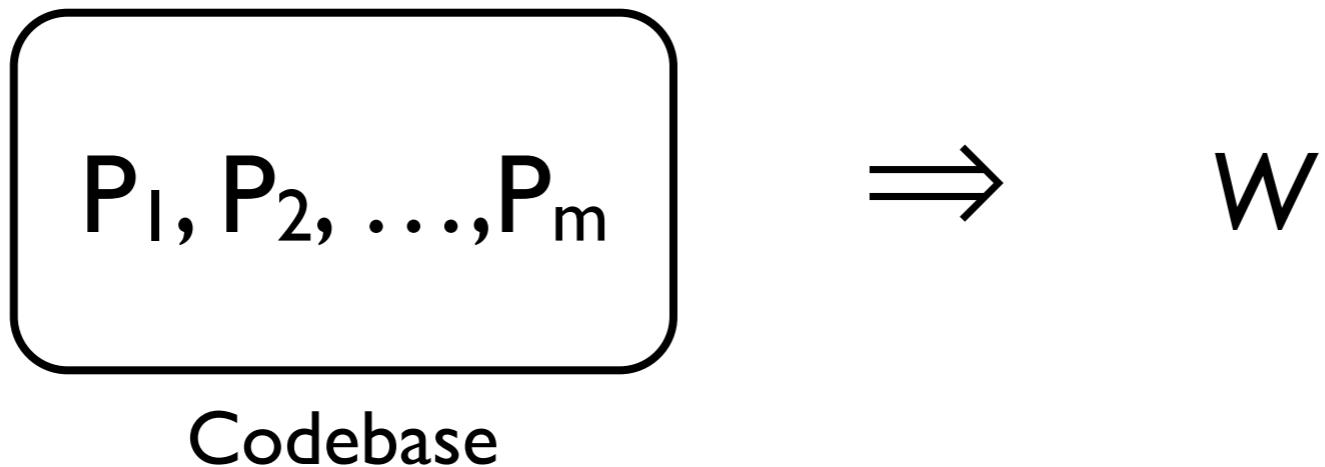
$$H_w : \text{pgm} \rightarrow 2^{\text{Var}}$$

# Overall Approach

- Parameterized heuristic

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

- Learn a good parameter  $W$  from existing codebase

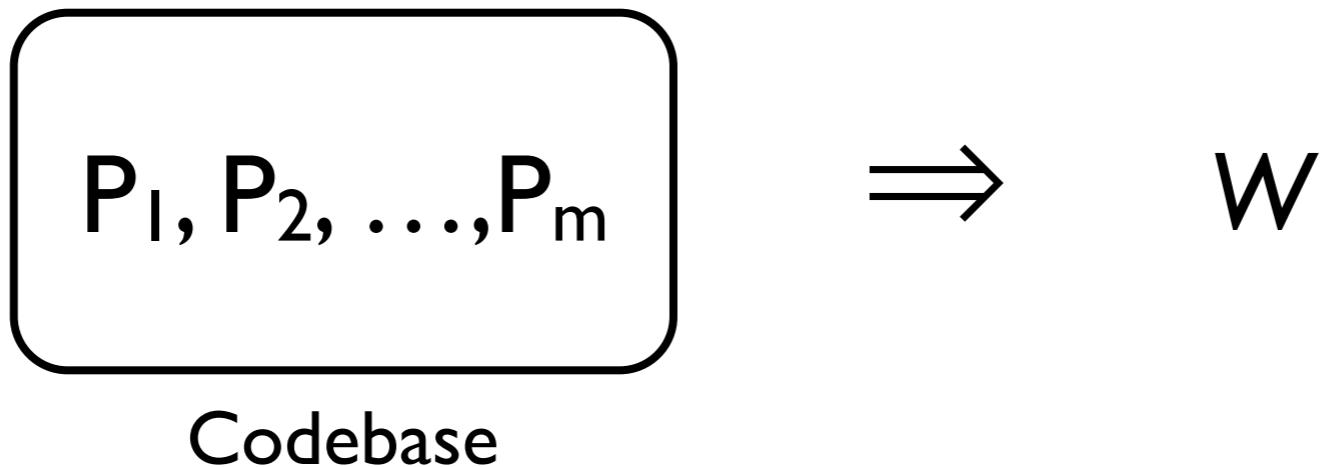


# Overall Approach

- Parameterized heuristic

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

- Learn a good parameter  $W$  from existing codebase



- For new program  $P$ , run static analysis with  $H_w(P)$

# I. 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.

# (I) Features

- Predicates over variables:

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

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

# (I) 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  $\mathbf{w}$  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:

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

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

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

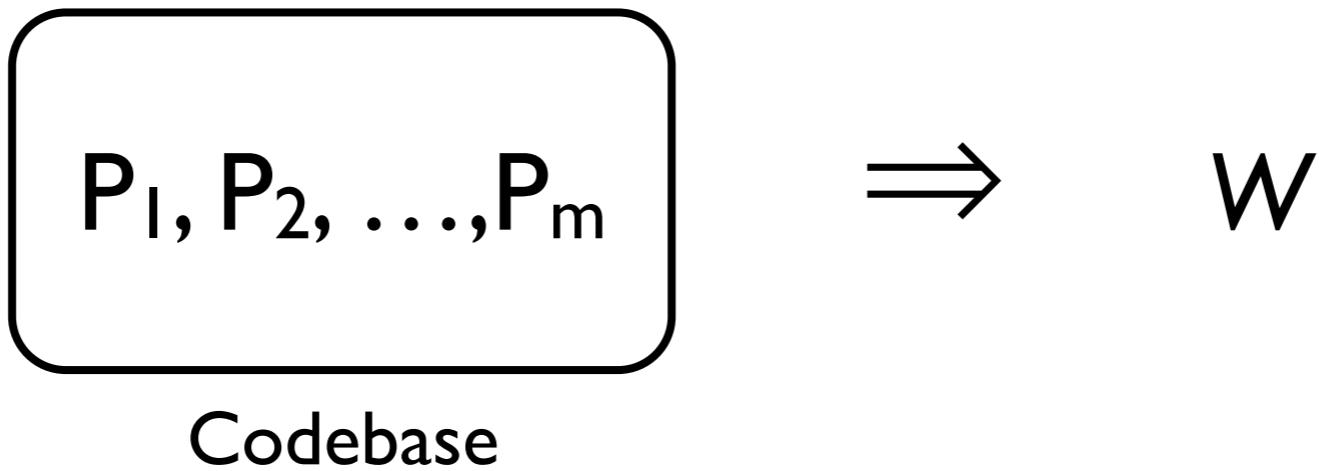
# (3) Choose Top-k Variables

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

$$\begin{array}{l} \text{score}(x) = 0.3 \\ \text{score}(y) = 0.6 \\ \text{score}(z) = 0.1 \end{array} \quad \rightarrow \quad \{x, y\}$$

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

## 2. Learn a Good Parameter



- Formulated as the optimization problem:

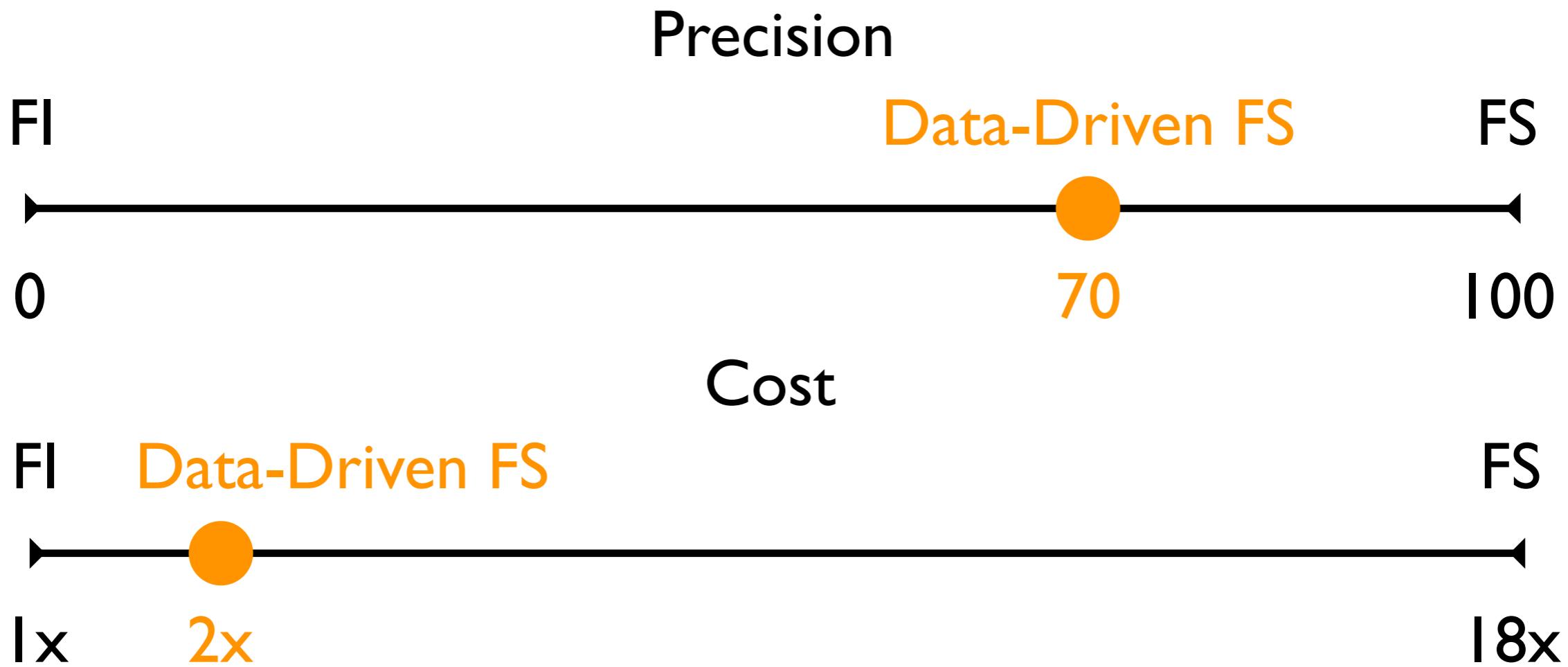
Find  $\mathbf{w}$  that maximizes  $\sum_{P_i} F(P_i, S_{\mathbf{w}}(P_i))$

- 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]

# Learning with Disjunctive Heuristics

- The linear heuristic cannot express disjunctive properties:

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

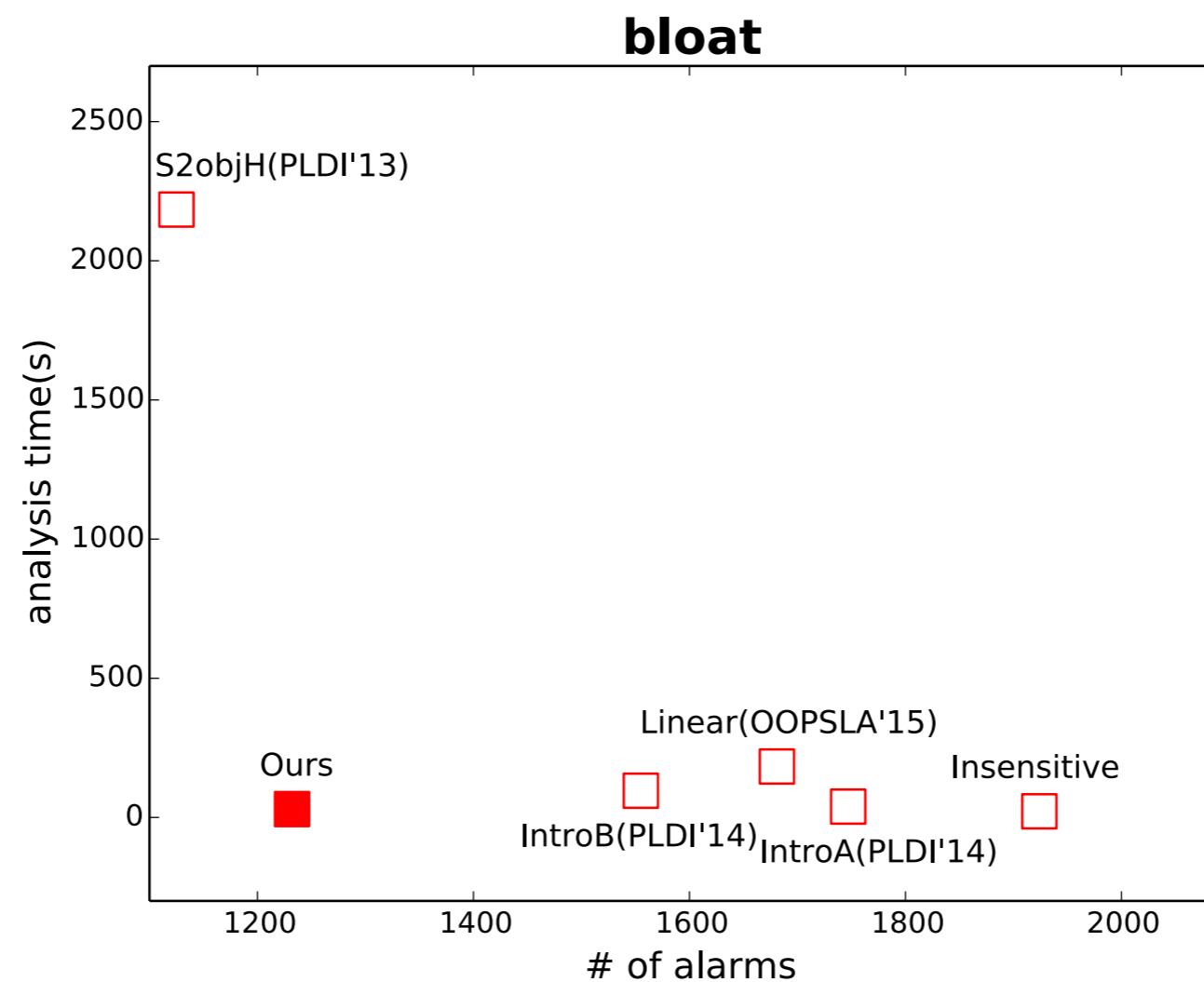
Goal:  $\{x, w\}$

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

- 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., $\text{if}(!x)$ )
	11	directly used in malloc (e.g., $\text{malloc}(x)$ )
	12	indirectly used in malloc (e.g., $y = x; \text{malloc}(y)$ )
	13	directly used in realloc (e.g., $\text{realloc}(x)$ )
	14	indirectly used in realloc (e.g., $y = x; \text{realloc}(y)$ )
	15	directly returned from malloc (e.g., $x = \text{malloc}(e)$ )
	16	indirectly returned from malloc
	17	directly returned from realloc (e.g., $x = \text{realloc}(e)$ )
	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	$(21 \vee 22) \wedge \neg 23$

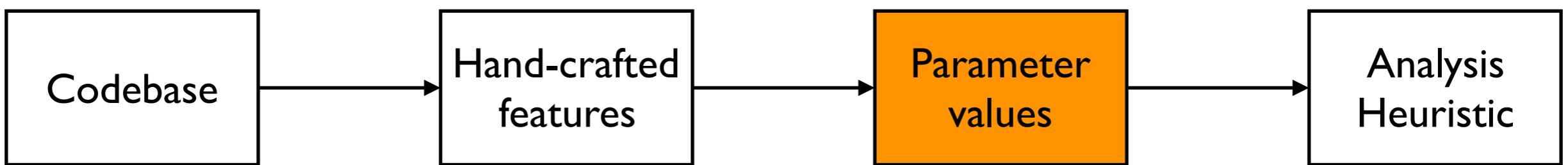
context-sensitivity

Type	#	Features
A	1	used in array declarations (e.g., $a[c]$ )
	2	used in memory allocation (e.g., $\text{malloc}(c)$ )
	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., $x < c \vee y$ )
	11	used inside loops
	12	used in return statements (e.g., $\text{return } c$ )
	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	$13 \wedge 12$

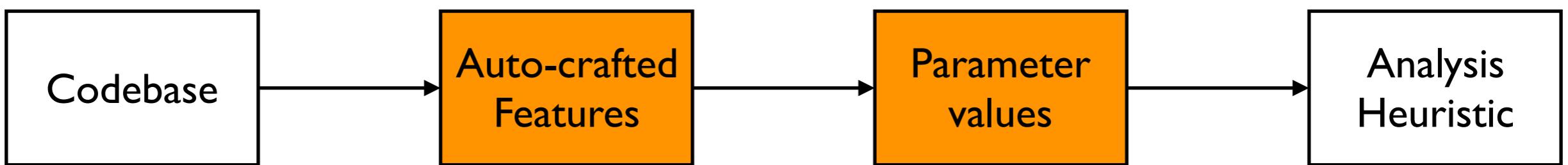
widening thresholds

# Automating Feature Engineering

Before (OOPSLA'15)

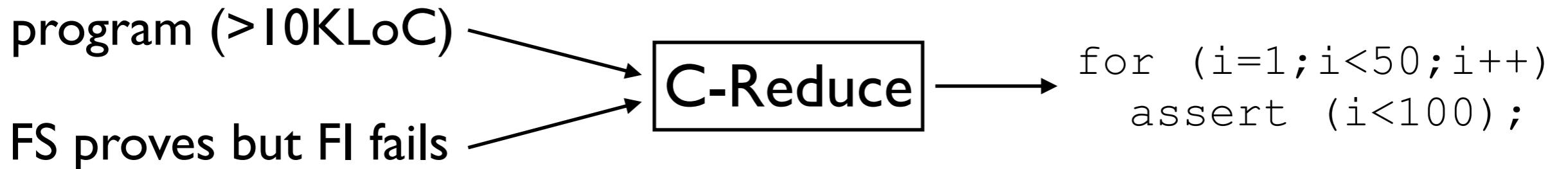


New method (OOPSLA'17)



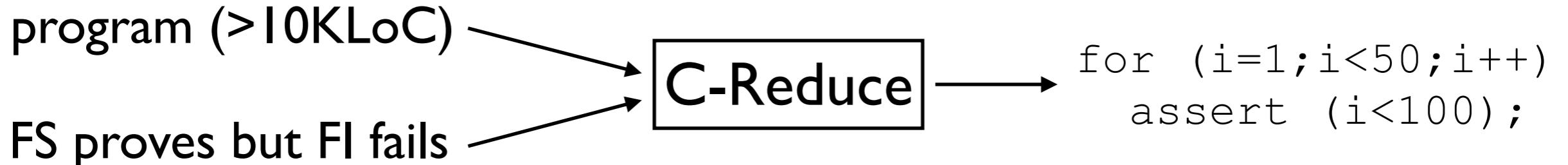
# Key Ideas

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

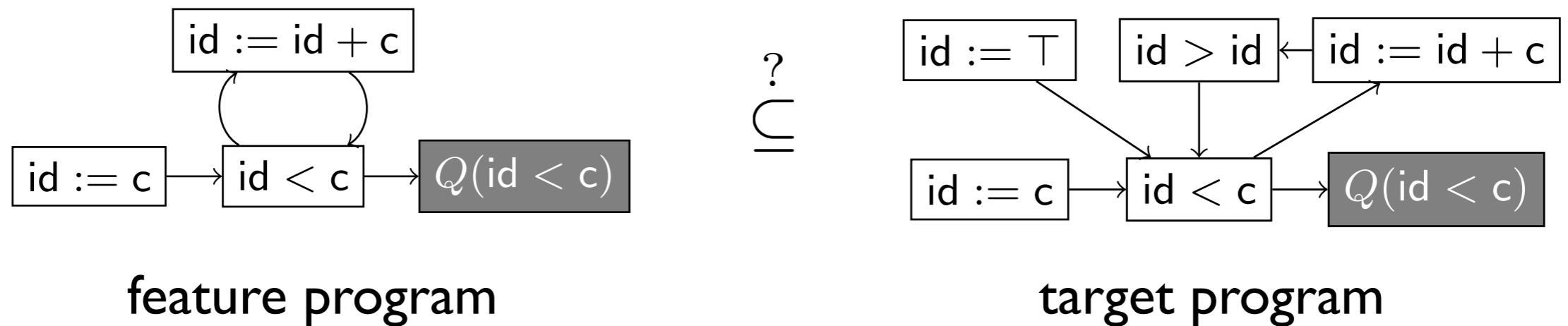


# Key Ideas

- 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



# **Data-Driven Concolic Testing**

## **(In submission)**

# Concolic Testing

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



S<sup>2</sup>E



T R I L O N

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

# Limitation of Random Testing

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

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

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

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

< 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

x=22, y=7

Symbolic  
State

x=a, 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

x=22, y=7,  
z=14

Symbolic  
State

x=a, 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

x=22, y=7,  
z=14

Symbolic  
State

x=a, y=β, z=2\*β  
2\*β ≠ a

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;  
        }  
    }  
}
```

1st iteration

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

# 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

x=2, y=1

Symbolic  
State

x=a, 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

x=2, y=1,  
z=2

Symbolic  
State

x=a, y= $\beta$ , z= $2^*\beta$   
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

x=2, y=1,  
z=2

Symbolic  
State

x=a, y=β, z=2\*β  
2\*β = a

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

x=2, y=1,  
z=2

Symbolic  
State

$x=a, y=\beta, z=2^*\beta$   
 $2^*\beta = a \wedge$   
 $a \leq \beta+10$

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

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

x=30, y=15

Symbolic  
State

x=a, 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

x=30, y=15,  
z=30

Symbolic  
State

x=a, 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

x=30, y=15,  
z=30

Symbolic  
State

x=a, y=β, z=2\*β  
2\*β = a

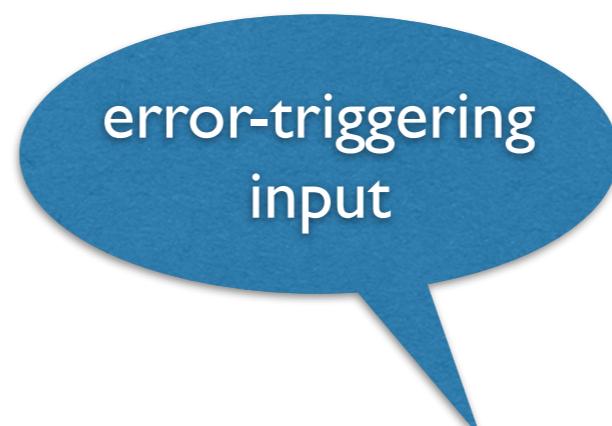
3rd iteration

# 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



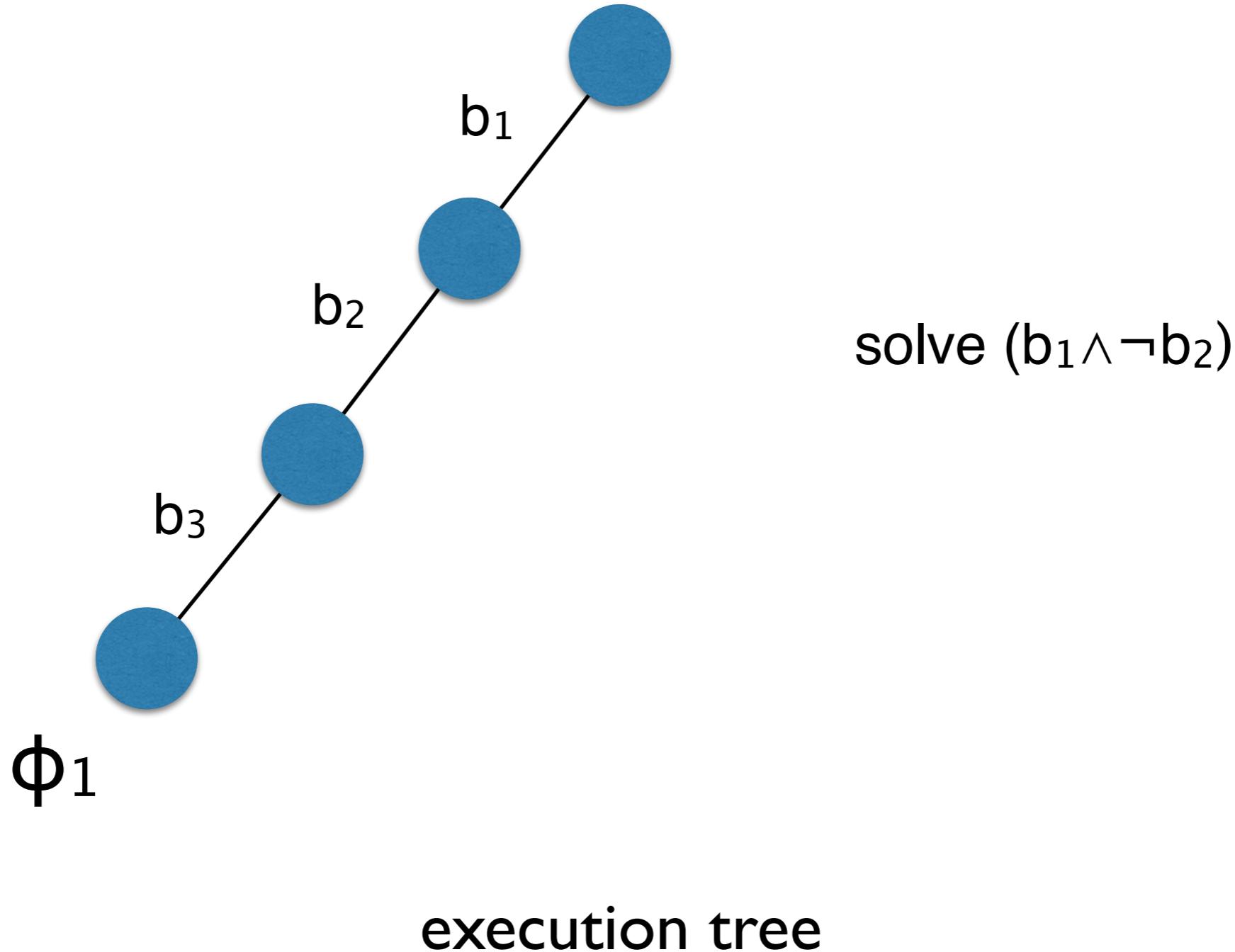
x=30, y=15,  
z=30

Symbolic  
State

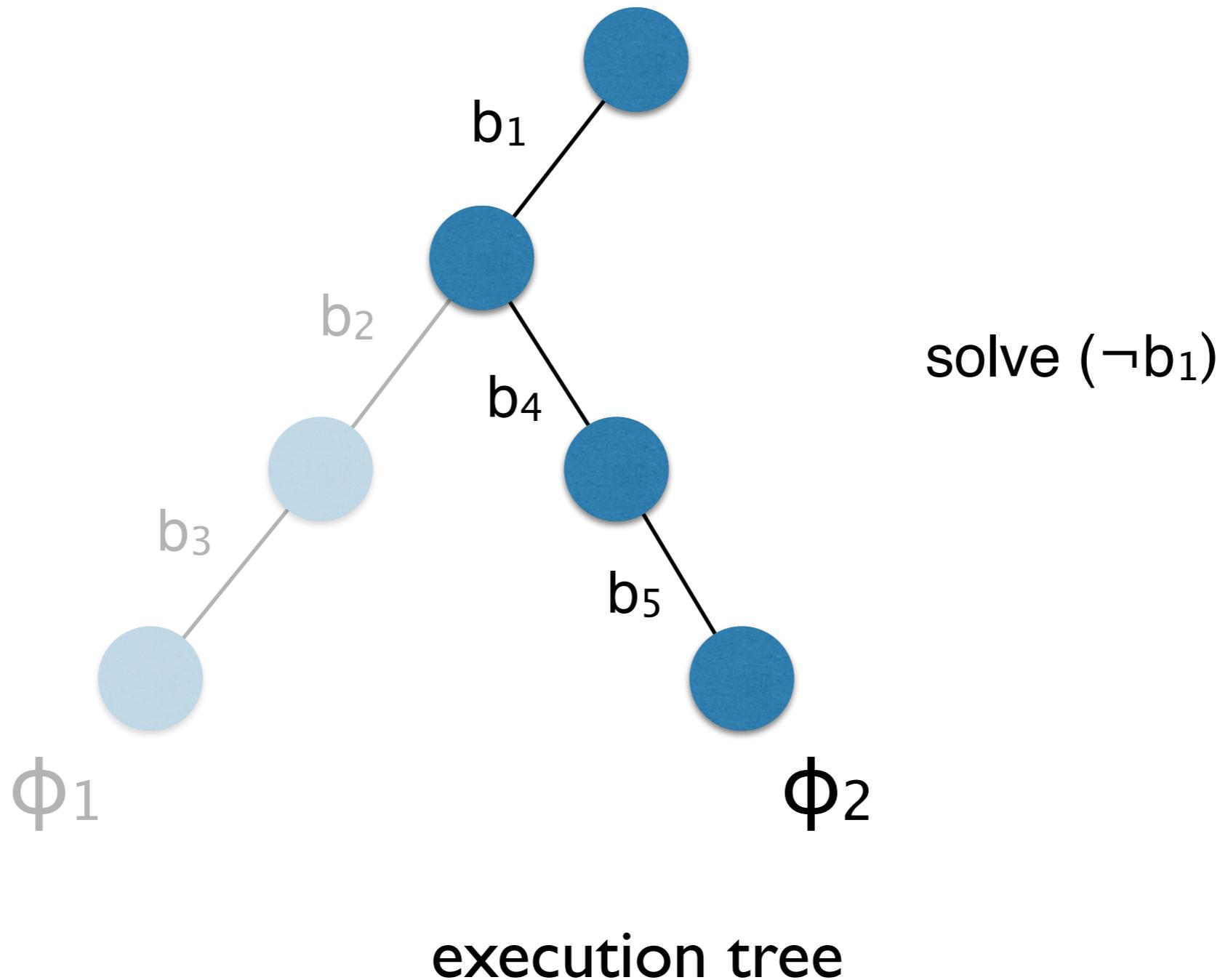
x=a, y=β, z=2\*β  
 $2^*\beta = a \wedge$   
 $a > \beta + 15$

3rd iteration

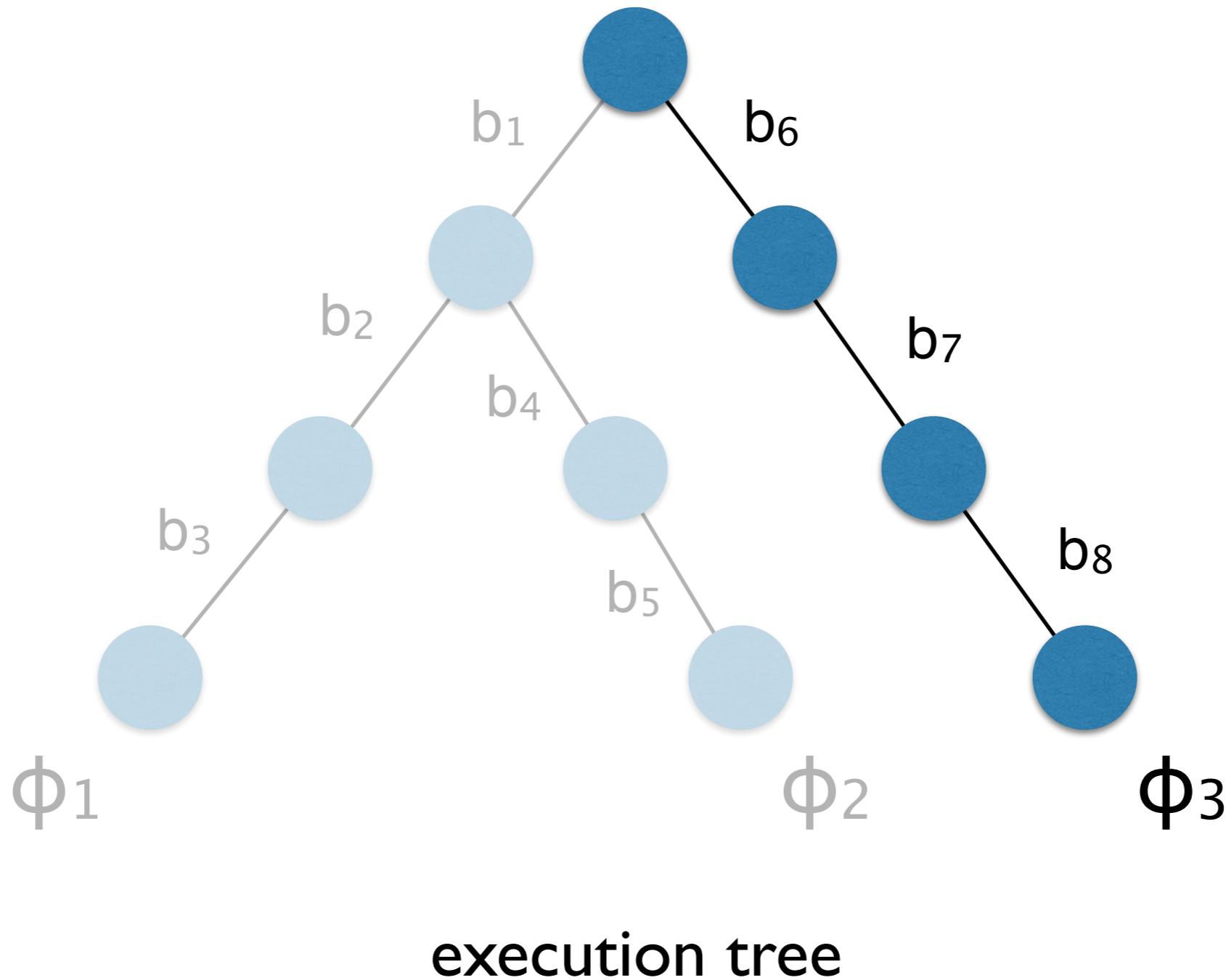
# Concolic Testing



# Concolic Testing



# Concolic Testing



# 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:    $\Phi_m \leftarrow \text{RunProgram}(P, v)$ 
5:    $T \leftarrow T \cdot \Phi_m$ 
6:   repeat
7:      $(\Phi, \phi_i) \leftarrow \text{Choose}(T)$       ( $\Phi = \phi_1 \wedge \dots \wedge \phi_n$ )
8:     until  $\text{SAT}(\bigwedge_{j < i} \phi_j \wedge \neg \phi_i)$ 
9:      $v \leftarrow \text{model}(\bigwedge_{j < i} \phi_j \wedge \neg \phi_i)$ 
10: end for
11: return |Branches( $T$ )|
```

# 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:    $\Phi_m \leftarrow \text{RunProgram}(P)$ 
5:    $T \leftarrow T \cdot \Phi_m$ 
6:   repeat
7:      $(\Phi, \phi_i) \leftarrow \text{Choose}(T) \quad (\Phi = \phi_1 \wedge \dots \wedge \phi_n)$ 
8:   until  $\text{SAT}(\bigwedge_{j < i} \phi_j \wedge \neg \phi_i)$ 
9:    $v \leftarrow \text{model}(\bigwedge_{j < i} \phi_j \wedge \neg \phi_i)$ 
10: end for
11: return |Branches( $T$ )|
```

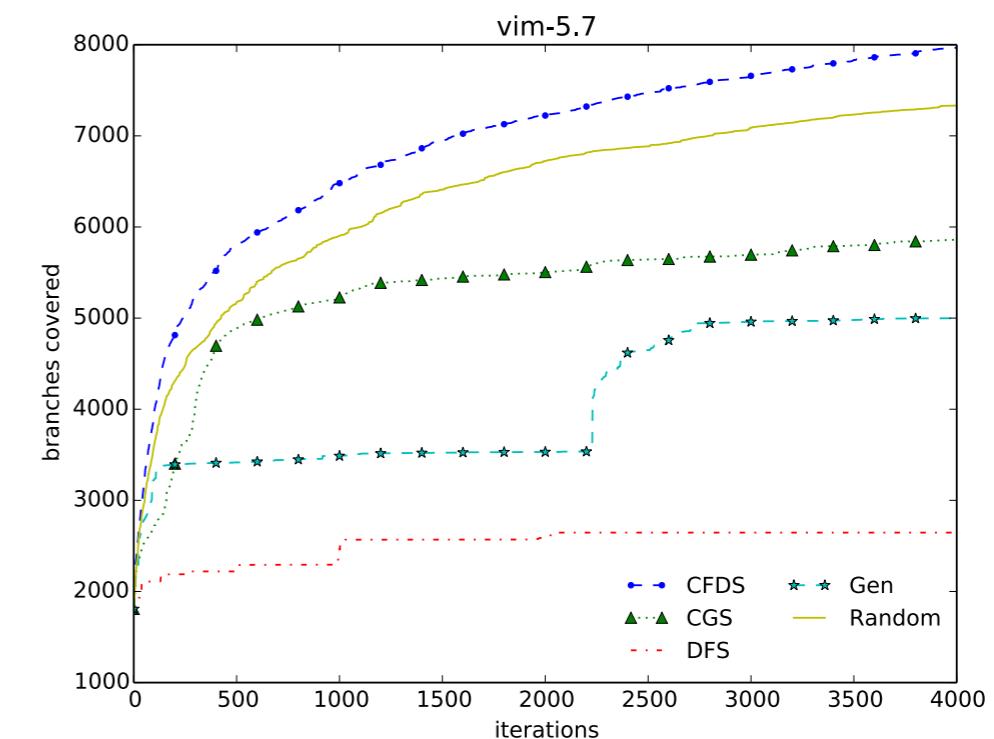
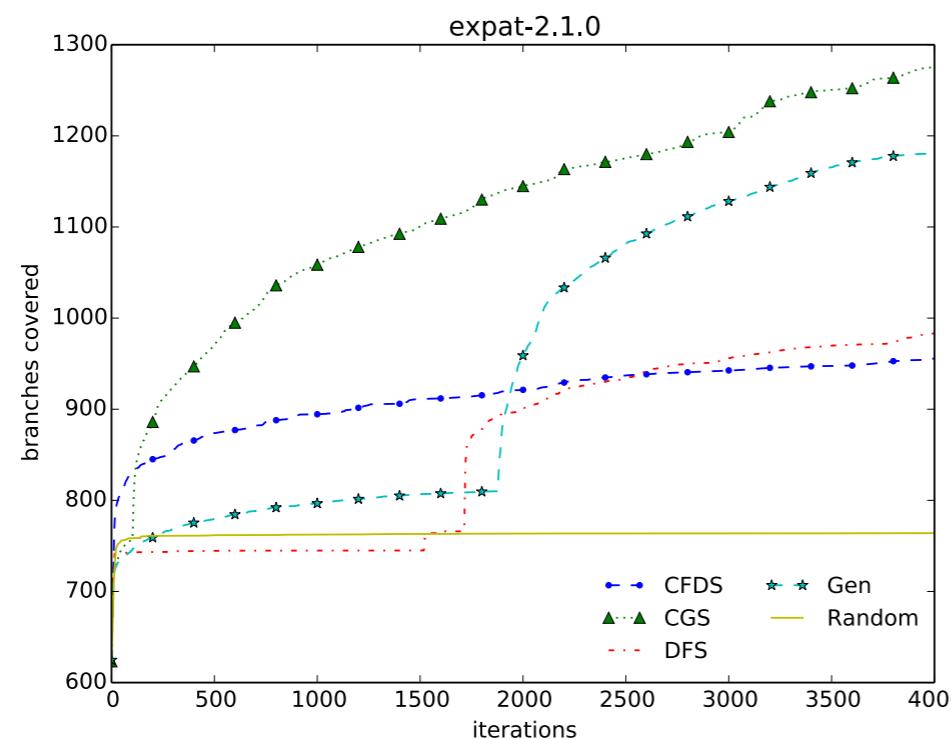


# Search Heuristics

- 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

Jacob Burnim  
EECS, UC Berkeley  
Email: jburnim@berkeley.edu

Koushik Sen  
EECS, UC Berkeley  
Email: koushik@berkeley.edu

**Abstract** Recently there has been great success in using symbolic execution to automatically generate test inputs for small to larger programs in the combinatorial explosion of the path space. It is likely that sophisticated strategies for searching the path space will be required to scale up to larger programs. We propose two search strategies for generating test inputs for large programs (by e.g., achieving significant branch coverage). The first strategy is based on a search heuristic that explores the search space guided by the control flow graph of the program under test. The second strategy is based on a search heuristic that explores open source code snippets, e.g., from GitHub. On these benchmarks, our first strategy achieves significantly greater branch coverage than the second strategy, while our second strategy is able to generate test inputs faster. Our results also show that our first strategy achieves significantly greater branch coverage than the default depth-first search strategy of symbolic testing.

We further propose two random search strategies. While in theory random testing is a program is run on random inputs, these two strategies test a program along random execution paths. The second strategy tests a program uniformly from the space of possible inputs. Our experiments demonstrate that a random search strategy is extremely unlikely to test all possible inputs of a program.

A number of search techniques for automated test generation [5, 6] have been proposed to address the limitations of random testing. Such techniques attempt to systematically explore a program, not alone to symbolically generate paths, generating and solving constraints to produce concrete inputs that test path feasibility, constraint solving [7, 8] and a variety of other heuristics to produce test inputs from symbolic execution simultaneously with concrete executions. These approaches are generally more scalable in practice because they do not require generating all possible inputs, precisely about complex data structures as well as to simply irreducible constraints.

A concrete search strategy operates on full, concrete inputs. In contrast, a symbolic search strategy, e.g.,  $\{l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8\}$ , corresponds to a set of inputs  $x = 1, y = 0$  along with symbolic path constraints – e.g.  $x > y \wedge y = 0 \wedge x \neq 4$ . For such an execution, a search must select one of the alternate branches

ASE'08

## CarFast: Achieving Higher Statement Coverage Faster

Sangmin Park  
Georgia Institute of Technology  
Atlanta, Georgia 30332, USA  
sangming@gatech.edu

Ishlaque Hussain,  
Christoph Csallner  
University of Illinois at Urbana-Champaign  
Urbana, IL 61801, USA  
ishlaque.hussain@illinois.edu

Kunal Tareja  
Accenture Technology Labs  
and University of North Carolina  
Raleigh, NC 27695, USA  
ktareja@unc.edu

B. M. Mainul Hossain  
University of Illinois at Chicago  
Chicago, IL 60612, USA  
bhoss2@uic.edu

Mark Gruebnik  
Accenture Technology Labs  
Chicago, IL 60612, USA  
drmain@accenture.com

that could quickly cover a significant portion of the branches in a test program despite searching only a small fraction of the program's path space.

The first strategy is guided by the static structure of the program under test, namely the control flow graph (CFG). In this strategy, we choose branches to refine the search space until we reach a specific node in the CFG to correctly branched. We experimentally show that this greedy approach to maintaining the branch coverage budget leads to a significant increase in the overall branch coverage.

The second strategy is to randomly search the path space of the program under test. We experimentally show that this strategy achieves greater branch coverage than the default depth-first search strategy of symbolic testing.

We further propose two random search strategies. While in theory random testing is a program is run on random inputs, these two strategies test a program along random execution paths. The second strategy tests a program uniformly from the space of possible inputs. Our experiments demonstrate that a random search strategy is extremely unlikely to test all possible inputs of a program.

In conclusion, we contrast our three proposed concrete search strategies with a traditional depth-first search strategy. Due to space limitations, we describe these search strategies by example, leaving the formal details to the accompanying technical report. Also included are the new standard details of our search strategies [7].

Figure 1 contains a short program in a C-like imperative language. We use this program as our running example to compare the performance of our three search strategies. For a conditional statement, we call the true statements in the true and false blocks a pair of branches. Thus, for the first conditional statement, pairs of branches are  $\{l_1, l_2\}, \{l_3, l_4\}, \{l_5, l_6\}$ , and  $\{l_7, l_8\}$ .

A concrete search strategy operates on full, concrete inputs. In contrast, a symbolic search strategy, e.g.,  $\{l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8\}$ , corresponds to a set of inputs  $x = 1, y = 0$  along with symbolic path constraints – e.g.  $x > y \wedge y = 0 \wedge x \neq 4$ . For such an execution, a search must select one of the alternate branches

that could quickly cover a significant portion of the branches in a test program despite searching only a small fraction of the program's path space.

The first strategy is guided by the static structure of the program under test, namely the control flow graph (CFG). In this strategy, we choose branches to refine the search space until we reach a specific node in the CFG to correctly branched. We experimentally show that this greedy approach to maintaining the branch coverage budget leads to a significant increase in the overall branch coverage.

The second strategy is to randomly search the path space of the program under test. We experimentally show that this strategy achieves greater branch coverage than the default depth-first search strategy of symbolic testing.

We further propose two random search strategies. While in theory random testing is a program is run on random inputs, these two strategies test a program along random execution paths. The second strategy tests a program uniformly from the space of possible inputs. Our experiments demonstrate that a random search strategy is extremely unlikely to test all possible inputs of a program.

In conclusion, we contrast our three proposed concrete search strategies with a traditional depth-first search strategy. Due to space limitations, we describe these search strategies by example, leaving the formal details to the accompanying technical report. Also included are the new standard details of our search strategies [7].

Figure 1 contains a short program in a C-like imperative language. We use this program as our running example to compare the performance of our three search strategies. For a conditional statement, we call the true statements in the true and false blocks a pair of branches. Thus, for the first conditional statement, pairs of branches are  $\{l_1, l_2\}, \{l_3, l_4\}, \{l_5, l_6\}$ , and  $\{l_7, l_8\}$ .

A concrete search strategy operates on full, concrete inputs. In contrast, a symbolic search strategy, e.g.,  $\{l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8\}$ , corresponds to a set of inputs  $x = 1, y = 0$  along with symbolic path constraints – e.g.  $x > y \wedge y = 0 \wedge x \neq 4$ . For such an execution, a search must select one of the alternate branches

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

Hyunmin Seo and Sunghun Kim  
Department of Computer Science and Engineering  
The Hong Kong University of Science and Technology, Hong Kong, China  
{hmseo, hunkim}@csie.ust.hk

**ABSTRACT**  
Statement coverage is an important metric of software quality, since it measures the completeness of testing. In this paper, we propose a context-guided search strategy to achieve higher statement coverage. A fundamental problem of software testing is how to achieve higher statement coverage, since it is difficult to find a test input that covers all statements in a program. To solve this problem, we propose a context-guided search strategy that explores the next branch of the selected branch. By doing so, the search space is reduced and the search cost is decreased. Moreover, the search cost is decreased by using a search heuristic that can generate redundant inputs for the same path and achieve high code coverage.

In this paper, we introduce a context-guided search (CGS) strategy. CGS has a test input that is used to find the next branch of the selected branch. The search process begins to do so as soon as it reaches a certain point. Each strategy applies different criteria to the branch selection process but CGS does not consider whether the branch is reached or not. Instead, CGS uses a context-guided search heuristic that can generate redundant inputs for the same path and achieve high code coverage.

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [10, 11].

CGS is evaluated on the largest dual-larger-than test input in the largest dual-larger-than testing setting [10, 11]. The search space is reduced by 10% and the search cost is decreased by 10%. CGS also increases the reliability of software testing [

# Limitations of Existing Search Heuristics

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

## Heuristics for Scalable Dynamic Test Generation

Jacob Burnim  
EECS, UC Berkeley  
Email: jburnim@berkeley.edu

Koushik Sen  
EECS, UC Berkeley  
Email: koushik@berkeley.edu

**Abstract** Recently there has been great success in using symbolic execution to automatically generate test inputs for small to larger programs in the combinatorial explosion of the path space. It is likely that sophisticated strategies for searching the path space will be required to scale up to larger programs. We present two approaches for generating test inputs for large programs (by e.g., achieving significant branch coverage). The first approach is based on a search strategy guided by a search strategy guided by the control flow graph of the program under test. The second approach is based on a search strategy that explores open source code and generates test inputs for the program under test. Our results show that the first approach achieves greater branch coverage than the second approach. On the other hand, the second approach achieves significantly greater branch coverage than the same testing budget from symbolic testing with a traditional depth-first search.

**INTRODUCTION**

Testing with manually generated inputs is the predominant technique in industry to ensure software quality – such testing accounts for 50-80% of the typical cost of software development. But random testing is often considered to be too slow and not very exhaustive.

A number of search techniques have been proposed for automated test generation in random testing [1], [2], [3], [4]. In random testing, the program is simply executed on randomly-generated inputs and the test suite is built by collecting all the errors found in the sense that random test input generation requires negligible time. However, random testing is extremely unlikely to test all possible paths of a program.

A number of search techniques for automated test generation [5], [6] have been proposed to address the limitations of random testing. Such techniques attempt to systematically explore a program, not alone to generate random program paths, generating and solving constraints to produce concrete inputs that test each path. Specifically, constraint testing [7], [8] and symbolic execution [9] have been proposed to generate symbolic execution simultaneously with concrete executions. These approaches are generally more scalable in practice because they can handle programs with many branches, especially about complex data structures as well as to simplify intermediate constraints.

Automated search techniques have been shown to be very effective in testing smaller programs, these approaches fail to scale to larger programs in which only a tiny fraction of the huge number of possible program paths can be explored. A natural question is how to derive search strategies

## CarFast: Achieving Higher Statement Coverage Faster

Sangmin Park  
Georgia Institute of Technology  
Atlanta, Georgia 30332, USA  
sangming@gatech.edu

Ishlaque Hussain,  
Christoph Csallner  
University of Illinois at Urbana-Champaign  
Urbana, IL 61801, USA  
ishlaque.hussain@illinois.edu

Kunal Tareja  
Accenture Technology Labs  
Raleigh, NC 27605, USA  
ktareja@accenture.com

### ABSTRACT

Statement coverage is an important metric of software quality. It is often used as a measure of test effectiveness. A fundamental problem of software testing is how to achieve higher statement coverage faster. In this paper, we propose CarFast, a search-based approach for achieving higher statement coverage. CarFast uses a search-based strategy to find test inputs that can steer execution toward sections of application that have the lowest statement coverage.

We compare our fast search-based approach for achieving higher statement coverage (CarFast) with two baseline approaches: random testing and symbolic testing. Our experiments demonstrate that CarFast achieves higher statement coverage in less time than random testing and symbolic testing. Our results indicate that random testing is not as effective as CarFast in terms of statement coverage. CarFast is also more effective than symbolic testing in terms of statement coverage.

**CATEGORIES AND SUBJECT DESCRIPTORS**

D.2.2 [Software Engineering]: Testing and Debugging  
Testing tools

Keywords

Symbolic testing, search-based testing, search strategies

### 1. HIGHER STATEMENT COVERAGE FASTER

Achieving higher statement coverage is the basic goal of test generation. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least once.

### 1.1. INTRODUCTION

Statement coverage is an important metric of software quality, since it indicates thoroughness of testing. Statement coverage, which measures the percentage of the executed statements in a program that have been executed at least once, is often used as a measure of quality of a program.

Recently, an automatic test generation technique called concolic testing [3], or Directed Automated Random Testing (DART) [7] has been proposed to achieve higher statement coverage than random testing. The idea is to use symbolic execution to generate test inputs that cover more statements in a program. For a conditional statement, we call the true statements in the true and false blocks a pair of statements. Thus, for a conditional statement with three branches are  $(l_1, l_2)$ ,  $(l_1, l_3)$ ,  $(l_2, l_3)$ ,  $(l_1, l_2, l_3)$ .

A concrete search strategy operates on full, concrete inputs. A search-based strategy, on the other hand, explores test inputs that are symbolic, i.e., inputs that are generated by a search engine. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Another search strategy operates on partial, symbolic inputs. A search-based strategy explores a program with a set of symbolic inputs. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Concolic testing is able to generate test inputs that cover more statements in a program than random testing. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least once.

Recently, an automatic test generation technique called concolic testing [3], or Directed Automated Random Testing (DART) [7] has been proposed to achieve higher statement coverage than random testing. The idea is to use symbolic execution to generate test inputs that cover more statements in a program. For a conditional statement, we call the true statements in the true and false blocks a pair of statements. Thus, for a conditional statement with three branches are  $(l_1, l_2)$ ,  $(l_1, l_3)$ ,  $(l_2, l_3)$ ,  $(l_1, l_2, l_3)$ .

A concrete search strategy operates on full, concrete inputs. A search-based strategy, on the other hand, explores test inputs that are symbolic, i.e., inputs that are generated by a search engine. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Another search strategy operates on partial, symbolic inputs. A search-based strategy explores a program with a set of symbolic inputs. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Concolic testing is able to generate test inputs that cover more statements in a program than random testing. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least once.

Recently, an automatic test generation technique called concolic testing [3], or Directed Automated Random Testing (DART) [7] has been proposed to achieve higher statement coverage than random testing. The idea is to use symbolic execution to generate test inputs that cover more statements in a program. For a conditional statement, we call the true statements in the true and false blocks a pair of statements. Thus, for a conditional statement with three branches are  $(l_1, l_2)$ ,  $(l_1, l_3)$ ,  $(l_2, l_3)$ ,  $(l_1, l_2, l_3)$ .

A concrete search strategy operates on full, concrete inputs. A search-based strategy, on the other hand, explores test inputs that are symbolic, i.e., inputs that are generated by a search engine. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Another search strategy operates on partial, symbolic inputs. A search-based strategy explores a program with a set of symbolic inputs. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Concolic testing is able to generate test inputs that cover more statements in a program than random testing. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least once.

Recently, an automatic test generation technique called concolic testing [3], or Directed Automated Random Testing (DART) [7] has been proposed to achieve higher statement coverage than random testing. The idea is to use symbolic execution to generate test inputs that cover more statements in a program. For a conditional statement, we call the true statements in the true and false blocks a pair of statements. Thus, for a conditional statement with three branches are  $(l_1, l_2)$ ,  $(l_1, l_3)$ ,  $(l_2, l_3)$ ,  $(l_1, l_2, l_3)$ .

A concrete search strategy operates on full, concrete inputs. A search-based strategy, on the other hand, explores test inputs that are symbolic, i.e., inputs that are generated by a search engine. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Another search strategy operates on partial, symbolic inputs. A search-based strategy explores a program with a set of symbolic inputs. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Concolic testing is able to generate test inputs that cover more statements in a program than random testing. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least once.

Recently, an automatic test generation technique called concolic testing [3], or Directed Automated Random Testing (DART) [7] has been proposed to achieve higher statement coverage than random testing. The idea is to use symbolic execution to generate test inputs that cover more statements in a program. For a conditional statement, we call the true statements in the true and false blocks a pair of statements. Thus, for a conditional statement with three branches are  $(l_1, l_2)$ ,  $(l_1, l_3)$ ,  $(l_2, l_3)$ ,  $(l_1, l_2, l_3)$ .

A concrete search strategy operates on full, concrete inputs. A search-based strategy, on the other hand, explores test inputs that are symbolic, i.e., inputs that are generated by a search engine. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Another search strategy operates on partial, symbolic inputs. A search-based strategy explores a program with a set of symbolic inputs. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Concolic testing is able to generate test inputs that cover more statements in a program than random testing. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least once.

Recently, an automatic test generation technique called concolic testing [3], or Directed Automated Random Testing (DART) [7] has been proposed to achieve higher statement coverage than random testing. The idea is to use symbolic execution to generate test inputs that cover more statements in a program. For a conditional statement, we call the true statements in the true and false blocks a pair of statements. Thus, for a conditional statement with three branches are  $(l_1, l_2)$ ,  $(l_1, l_3)$ ,  $(l_2, l_3)$ ,  $(l_1, l_2, l_3)$ .

A concrete search strategy operates on full, concrete inputs. A search-based strategy, on the other hand, explores test inputs that are symbolic, i.e., inputs that are generated by a search engine. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Another search strategy operates on partial, symbolic inputs. A search-based strategy explores a program with a set of symbolic inputs. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Concolic testing is able to generate test inputs that cover more statements in a program than random testing. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least once.

Recently, an automatic test generation technique called concolic testing [3], or Directed Automated Random Testing (DART) [7] has been proposed to achieve higher statement coverage than random testing. The idea is to use symbolic execution to generate test inputs that cover more statements in a program. For a conditional statement, we call the true statements in the true and false blocks a pair of statements. Thus, for a conditional statement with three branches are  $(l_1, l_2)$ ,  $(l_1, l_3)$ ,  $(l_2, l_3)$ ,  $(l_1, l_2, l_3)$ .

A concrete search strategy operates on full, concrete inputs. A search-based strategy, on the other hand, explores test inputs that are symbolic, i.e., inputs that are generated by a search engine. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Another search strategy operates on partial, symbolic inputs. A search-based strategy explores a program with a set of symbolic inputs. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Concolic testing is able to generate test inputs that cover more statements in a program than random testing. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least once.

Recently, an automatic test generation technique called concolic testing [3], or Directed Automated Random Testing (DART) [7] has been proposed to achieve higher statement coverage than random testing. The idea is to use symbolic execution to generate test inputs that cover more statements in a program. For a conditional statement, we call the true statements in the true and false blocks a pair of statements. Thus, for a conditional statement with three branches are  $(l_1, l_2)$ ,  $(l_1, l_3)$ ,  $(l_2, l_3)$ ,  $(l_1, l_2, l_3)$ .

A concrete search strategy operates on full, concrete inputs. A search-based strategy, on the other hand, explores test inputs that are symbolic, i.e., inputs that are generated by a search engine. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Another search strategy operates on partial, symbolic inputs. A search-based strategy explores a program with a set of symbolic inputs. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Concolic testing is able to generate test inputs that cover more statements in a program than random testing. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least once.

Recently, an automatic test generation technique called concolic testing [3], or Directed Automated Random Testing (DART) [7] has been proposed to achieve higher statement coverage than random testing. The idea is to use symbolic execution to generate test inputs that cover more statements in a program. For a conditional statement, we call the true statements in the true and false blocks a pair of statements. Thus, for a conditional statement with three branches are  $(l_1, l_2)$ ,  $(l_1, l_3)$ ,  $(l_2, l_3)$ ,  $(l_1, l_2, l_3)$ .

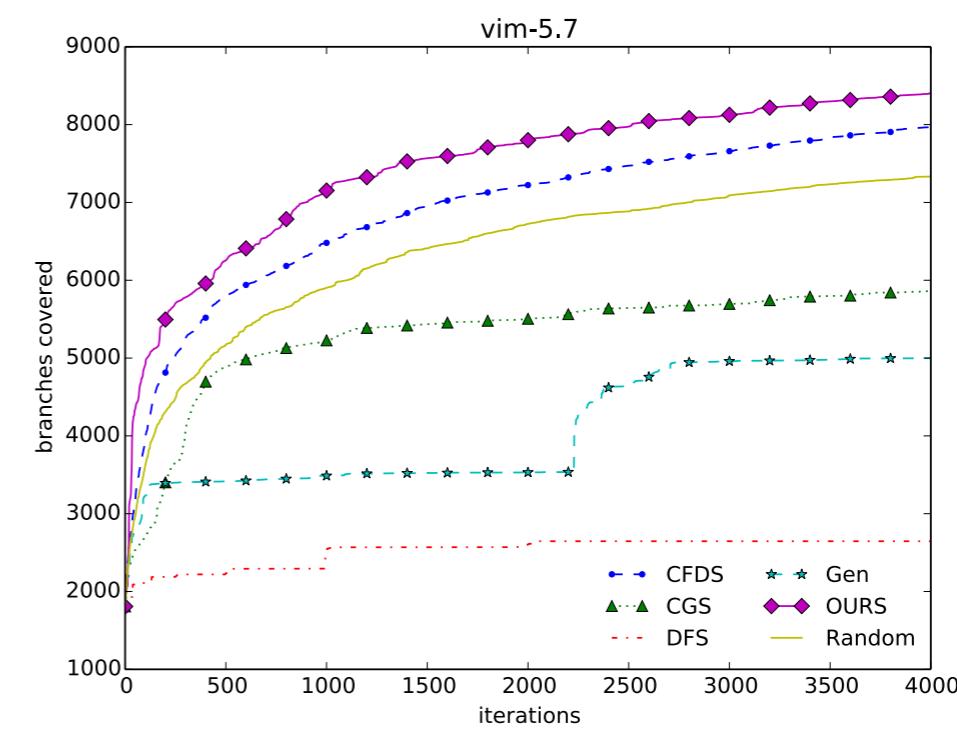
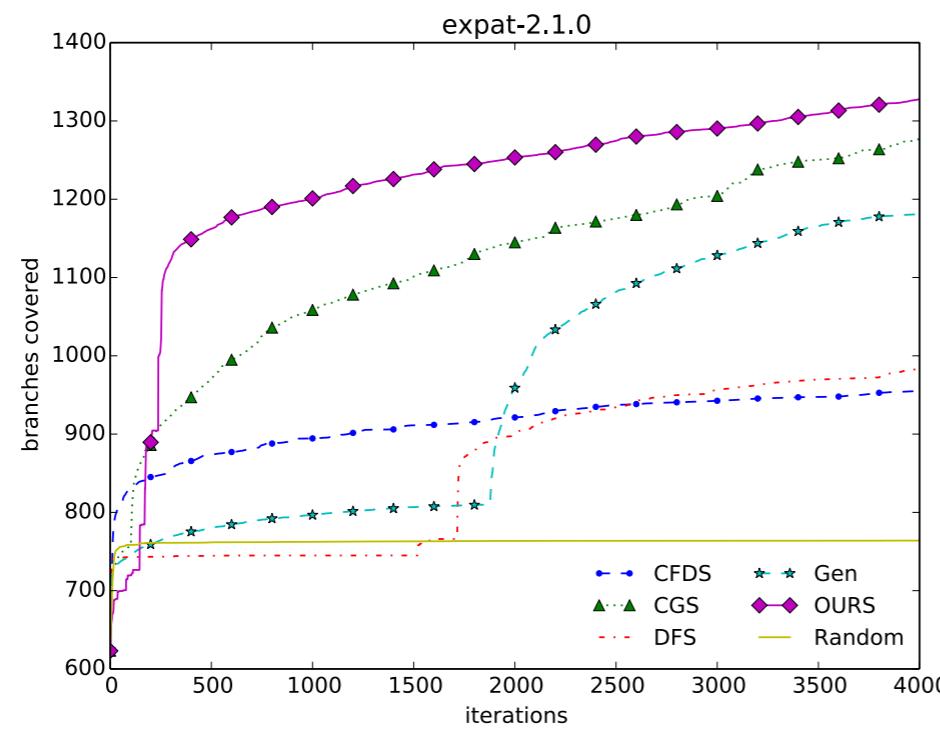
A concrete search strategy operates on full, concrete inputs. A search-based strategy, on the other hand, explores test inputs that are symbolic, i.e., inputs that are generated by a search engine. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Another search strategy operates on partial, symbolic inputs. A search-based strategy explores a program with a set of symbolic inputs. For example, consider a simple program that takes two integer inputs  $x$  and  $y$  and prints their sum. This program has three branches: one branch is the case when  $x = 0$  and  $y = 0$ ; another branch is the case when  $x \neq 0$  and  $y = 0$ ; and the third branch is the case when  $x = 0$  and  $y \neq 0$ . A search-based strategy would explore all three branches by generating three symbolic inputs  $x = 0, y = 0$ ,  $x \neq 0, y = 0$ , and  $x = 0, y \neq 0$ .

Concolic testing is able to generate test inputs that cover more statements in a program than random testing. Statement coverage is often used as a measure of quality of a program. Statement coverage measures the percentage of the executed statements in a program that have been executed at least

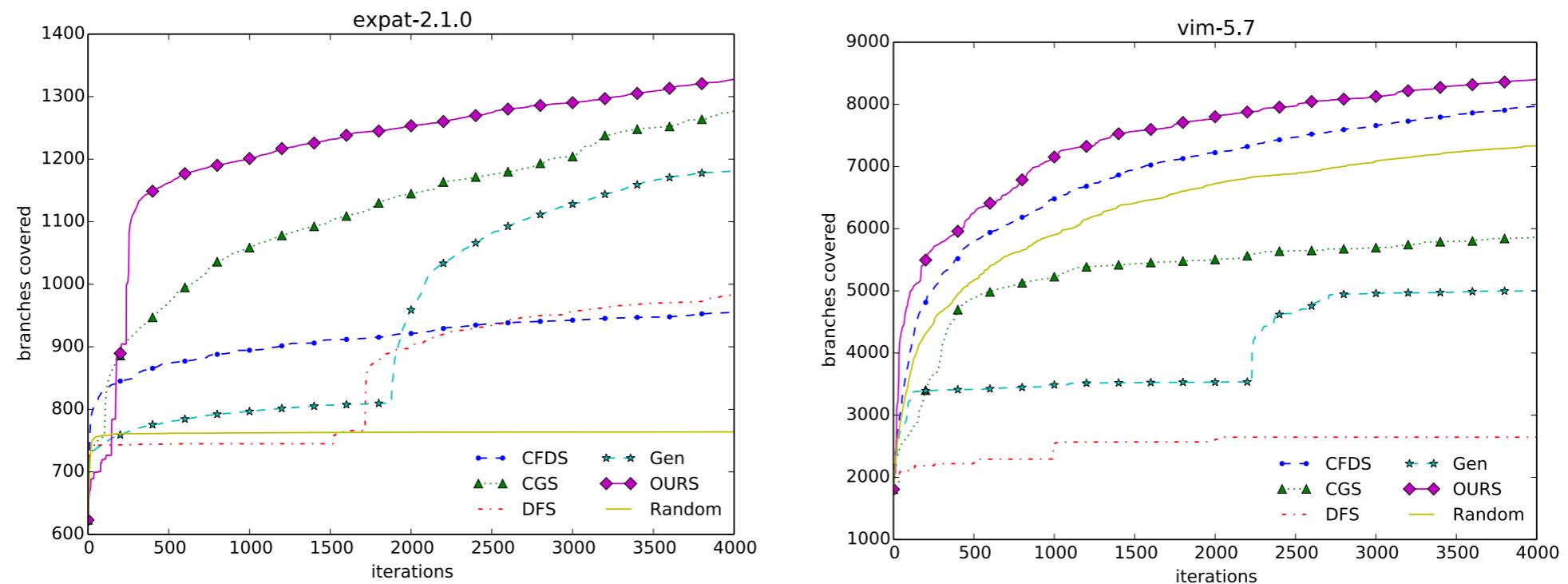
# Effectiveness

- Considerable increase in branch coverage



# Effectiveness

- Considerable increase in branch coverage

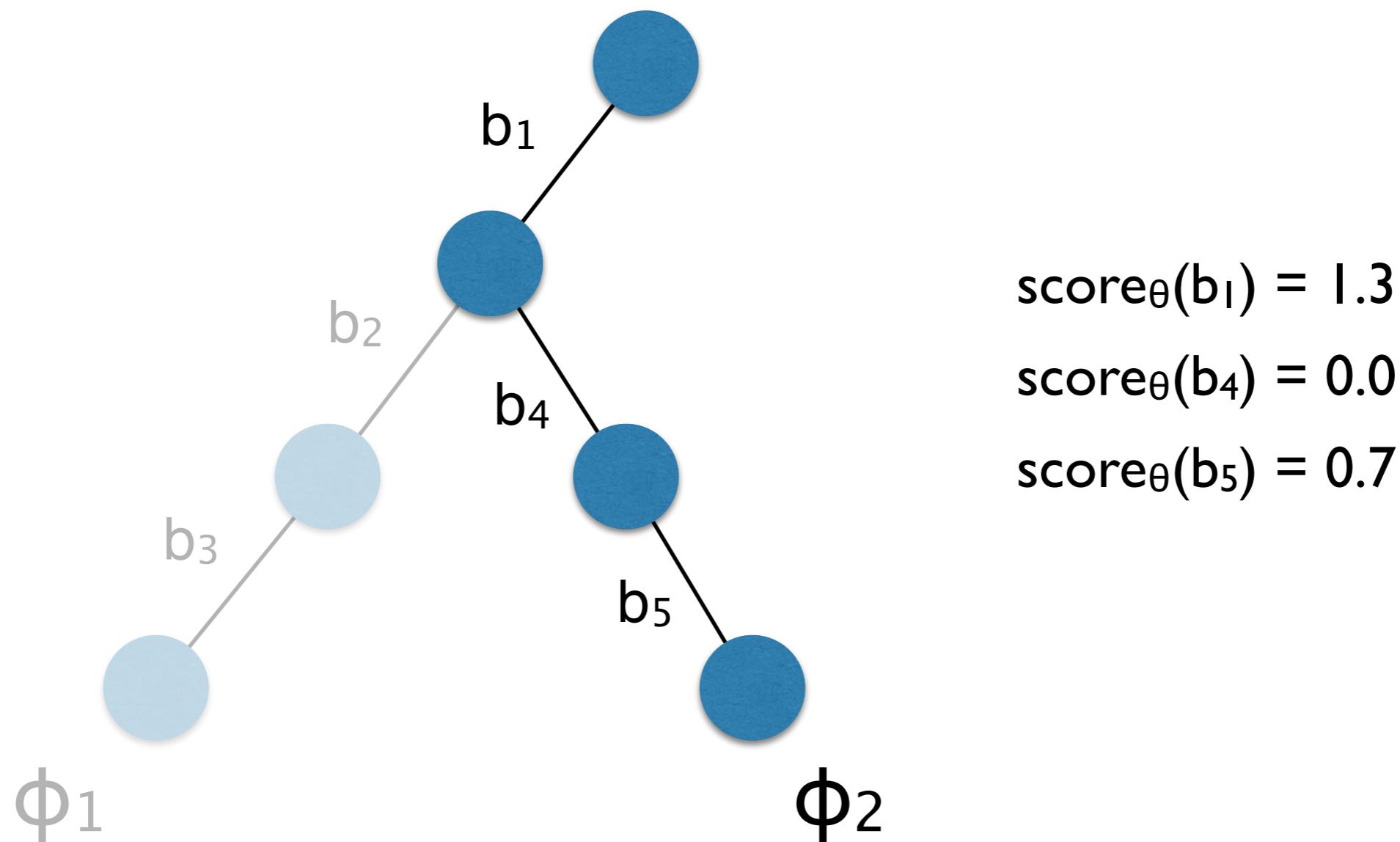


- Dramatic increase in bug-finding

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

# Parameterized Search Heuristic

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



# (I) Feature Extraction

- A feature is a predicate on branches:

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

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

- Represent a branch by a feature vector

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

- 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:

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

where

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

# Naive Algorithm

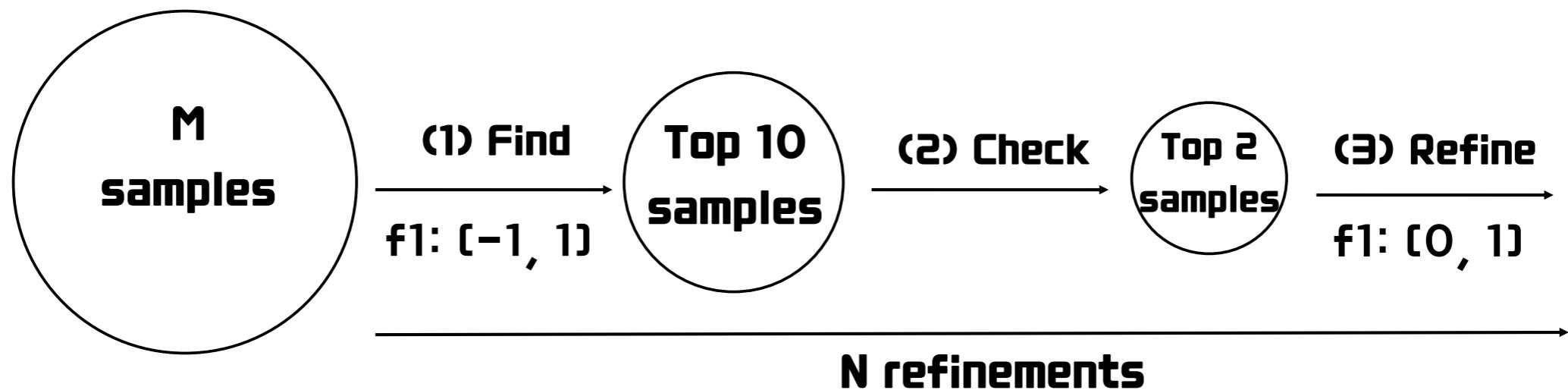
- Naive algorithm based on random sampling

```
1: repeat
2:    $\theta \leftarrow$  sample from  $\mathbb{R}^k$ 
3:    $B \leftarrow C(P, \text{Choose}_\theta)$ 
4: until timeout
5: return best  $\theta$  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



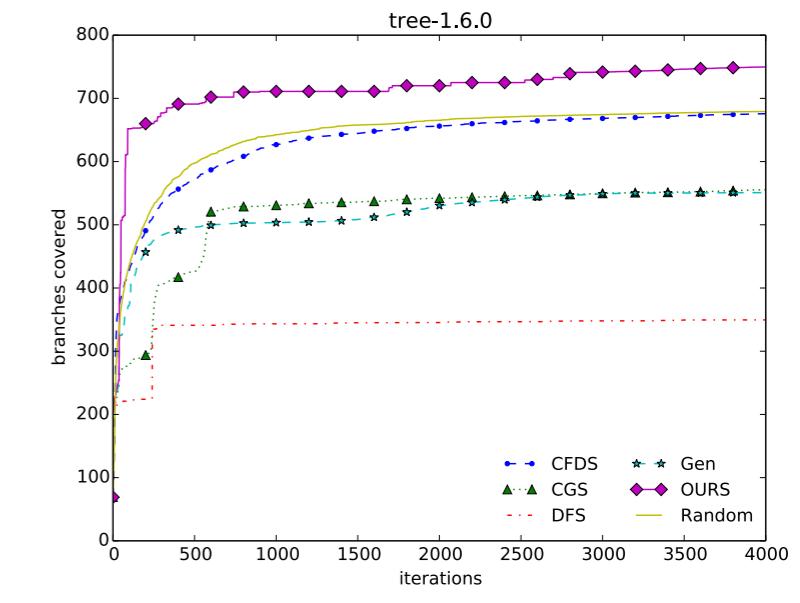
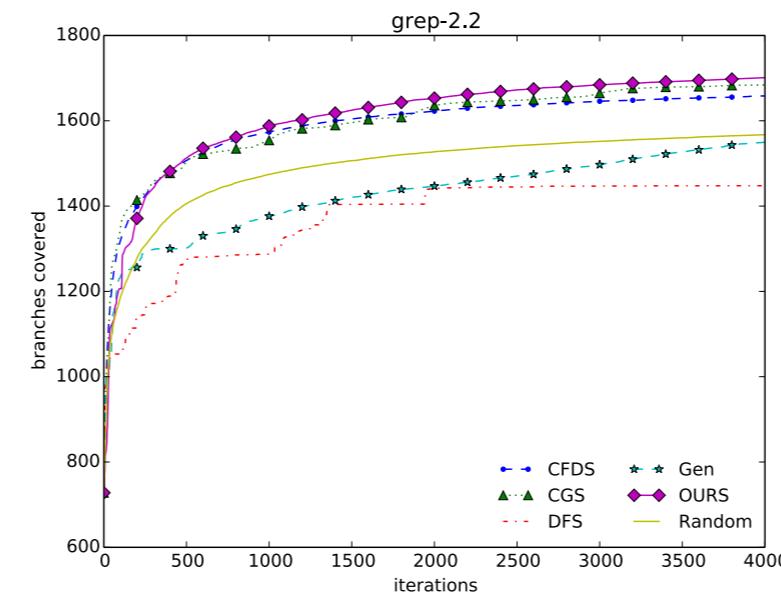
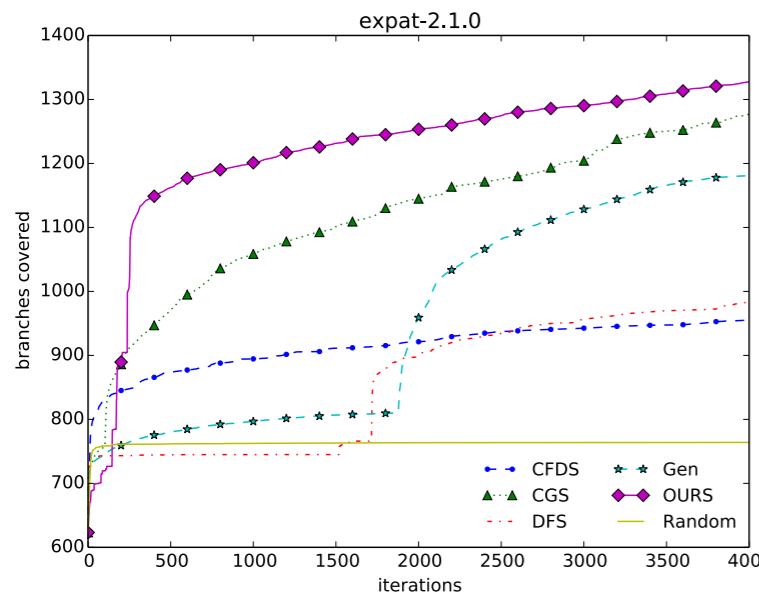
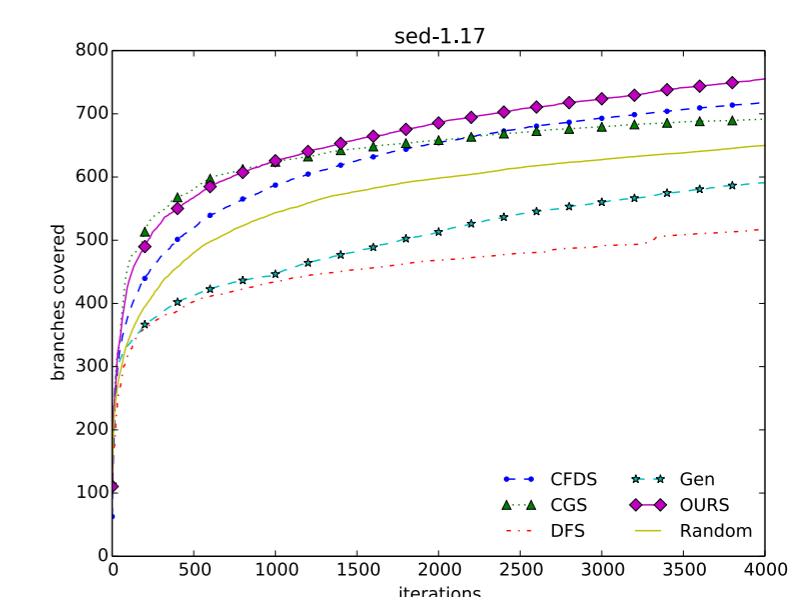
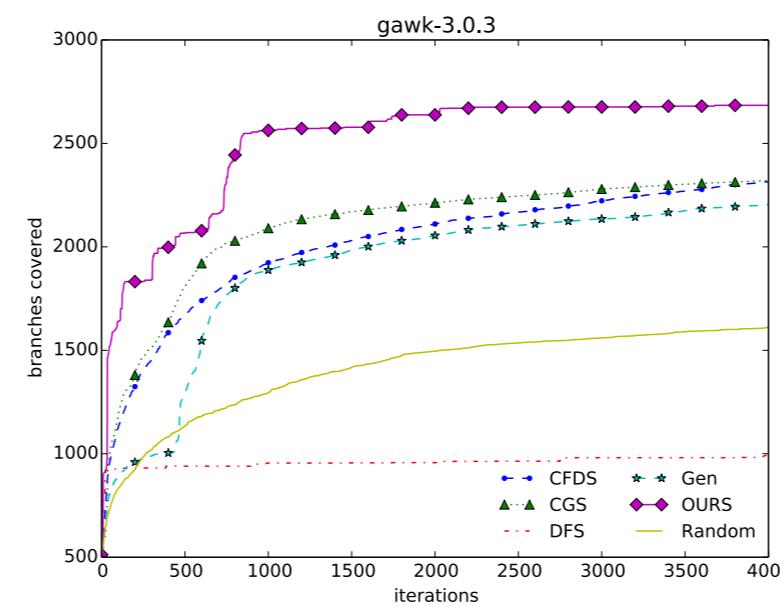
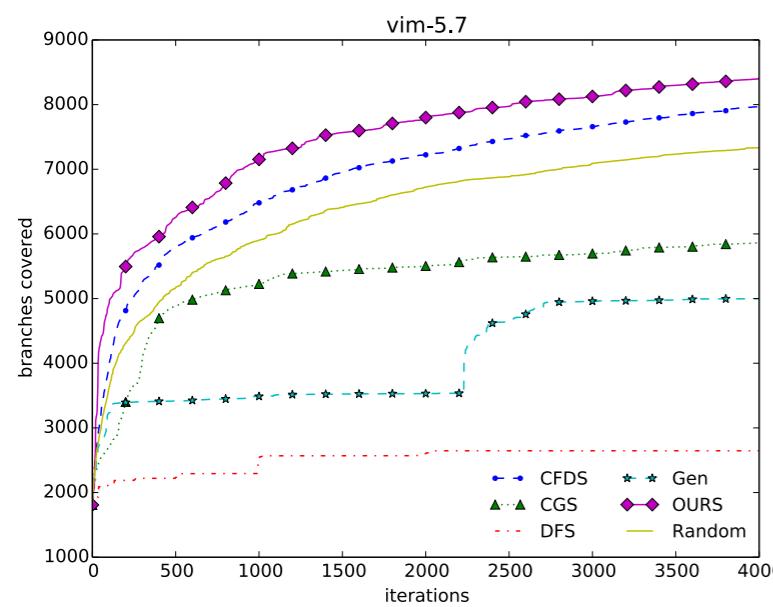
# Experiments

- 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	<b>8,744</b>	8,322	6,150	7,645	5,092	2,646
expat	<b>1,422</b>	1,060	1,337	965	1,348	1,027
gawk	<b>2,684</b>	2,532	2,449	2,035	2,443	1,025
grep	<b>1,807</b>	1,726	1,751	1,598	1,640	1,456
sed	<b>830</b>	780	781	690	698	568
tree	<b>797</b>	702	599	704	600	360

- On small benchmarks

	OURS	CFDS	CGS	Random	Gen	DFS
cdaudio	<b>250</b>	<b>250</b>	<b>250</b>	242	236	<b>250</b>
floppy	<b>205</b>	<b>205</b>	<b>205</b>	170	168	<b>205</b>
replace	<b>181</b>	177	<b>181</b>	174	171	176
kbfiltr	<b>149</b>	<b>149</b>	<b>149</b>	<b>149</b>	134	<b>149</b>

# Effectiveness

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

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

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

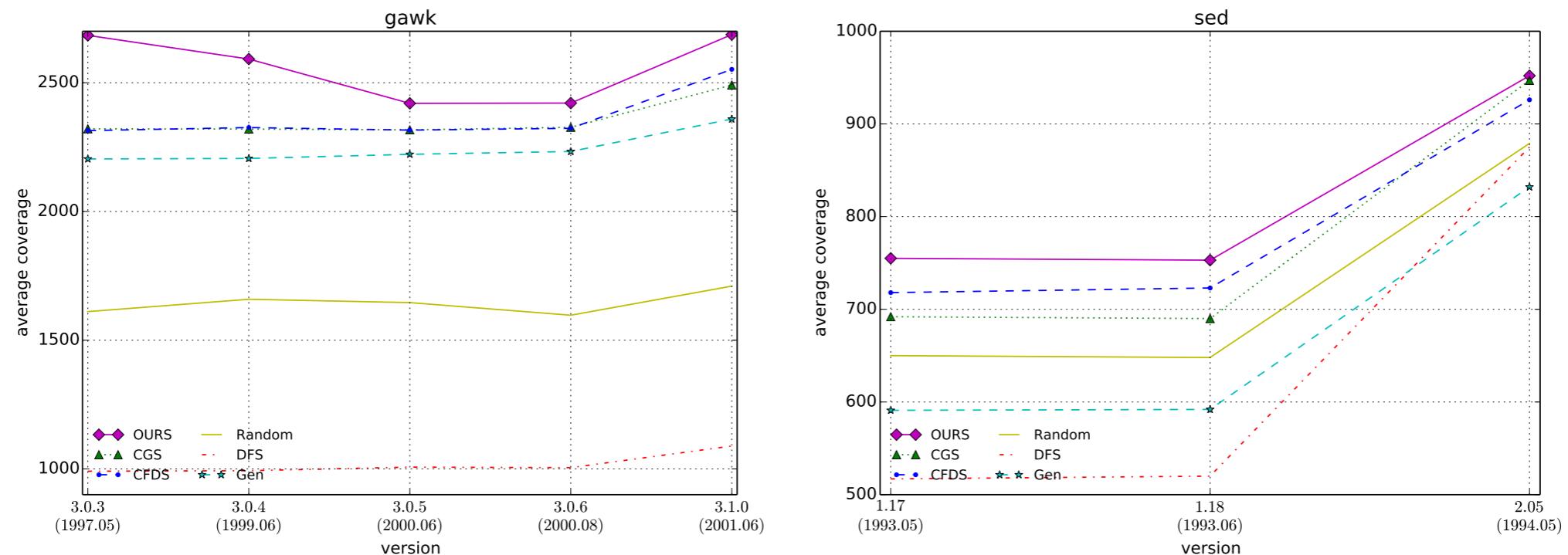
# Training Overhead

- 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	<b>14,003</b>	13,706	7,934	13,835	7,290	7,934
expat	<b>2,455</b>	2,339	2,157	1,325	2,116	2,036
gawk	<b>3,473</b>	3,382	3,261	3,367	3,302	1,905
grep	<b>2,167</b>	2,024	2,016	2,066	1,965	1,478
sed	1,019	1,041	<b>1,042</b>	1,007	979	937
tree	<b>808</b>	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