

Data-Driven Program Analysis

Hakjoo Oh

Korea University

30 October 2017 @KAIST

(co-work with Sooyoung Cha, Kwonsoo Chae, Kihong Heo, Seongjun Hong, Minseok Jeon, Sehun Jeong, Junhee Lee, 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 Program Analysis



Astrée

DOOP

TAJS

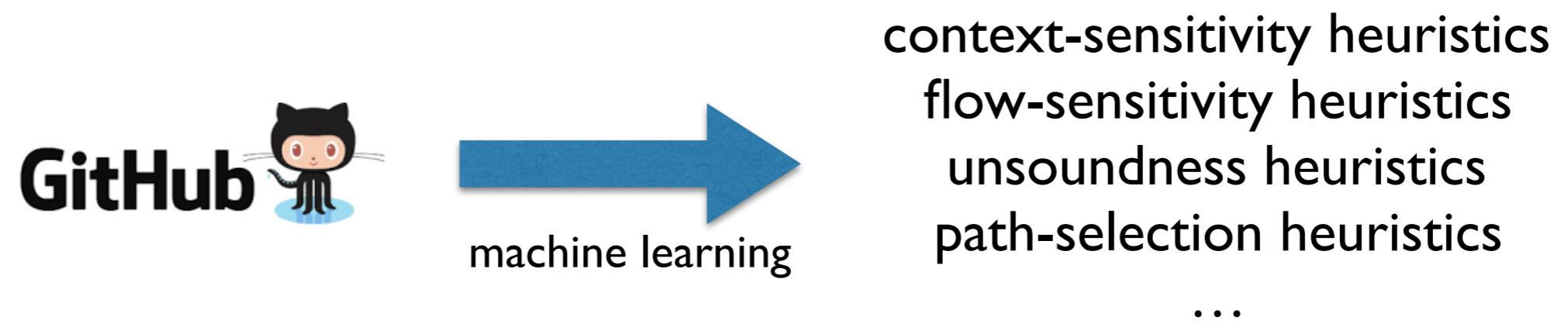
SAFE



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

Automatically Generating Analysis Heuristics from Data

- Use data to make empirical decisions in program analysis

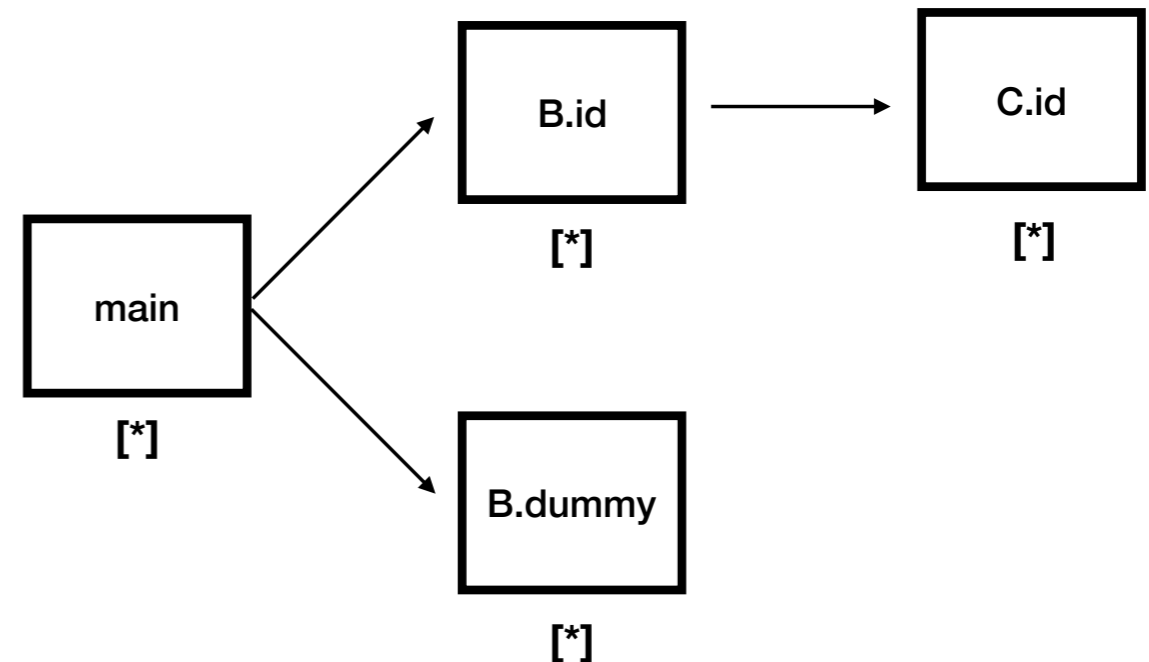


- **Automatic:** little reliance on analysis designers
- **Powerful:** machine-tuning outperforms hand-tuning
- **Stable:** can be tuned for target programs

Example: 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.id(new D());//query1
17: E e = (E) b2.id(new E());//query2
18: b1.dummy();
19: b2.dummy();}}
```

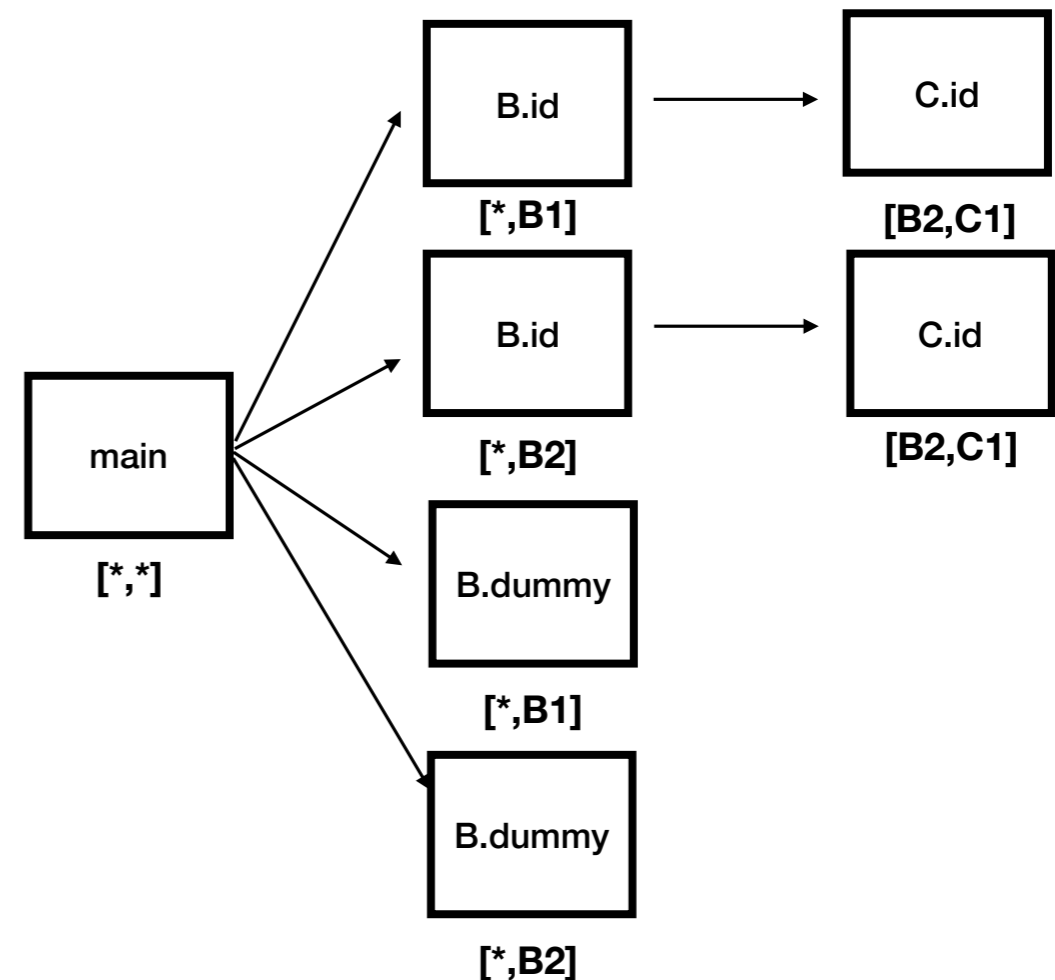
Without context-sensitivity,
analysis fails to prove queries



Example: 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.id(new D());//query1
17: E e = (E) b2.id(new E());//query2
18: b1.dummy();
19: b2.dummy();}}
```

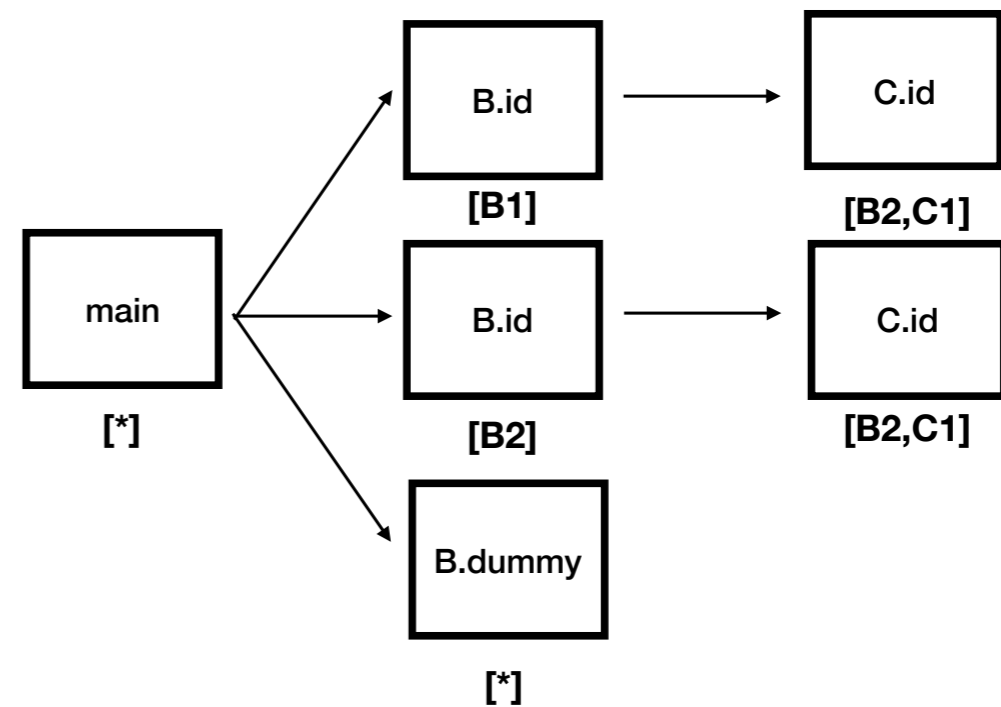
2-object-sensitivity succeeds
but does not scale



Example: 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.id(new D());//query1
17: E e = (E) b2.id(new E());//query2
18: b1.dummy();
19: b2.dummy();}}
```

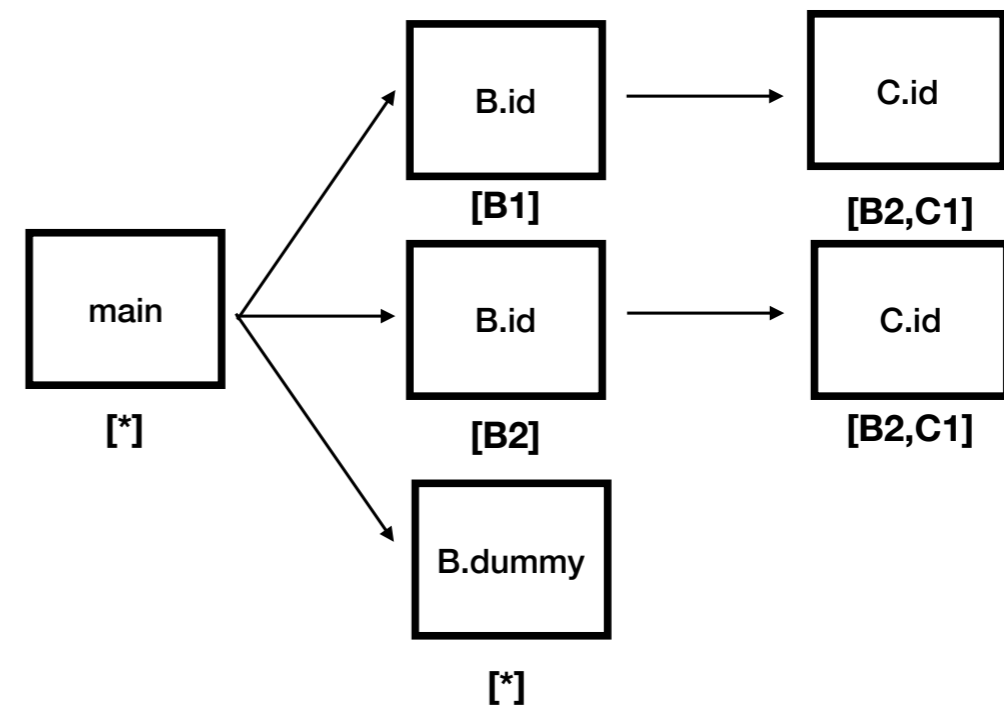
Apply 2-obj-sens: {C.id}
Apply 1-obj-sens: {B.id}
Apply insens: {B.m}



Example: 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.id(new D());//query1
17: E e = (E) b2.id(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}



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

Our Data-Driven Framework

Parametric
static analyzer

Training data
(programs)

Atomic features
(a_1, a_2, \dots, a_{25})

```
graph TD; A[Parametric static analyzer] --> D[Our DD Framework]; B[Training data (programs)] --> D; C[Atomic features (a1, a2, ..., a25)] --> D; D --> E[ ];
```

Our DD Framework

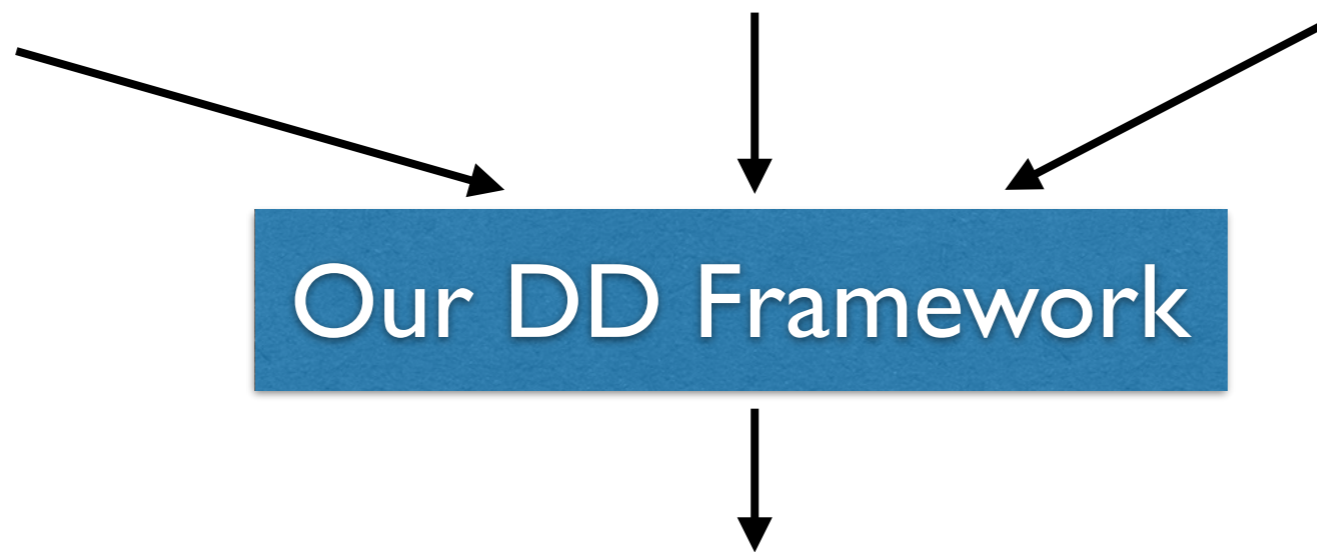
e.g., methods have
invocation stmt,
methods return
strings, etc

Our Data-Driven Framework

Parametric
static analyzer

Training data
(programs)

Atomic features
(a_1, a_2, \dots, a_{25})



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

Heuristic for applying 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

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

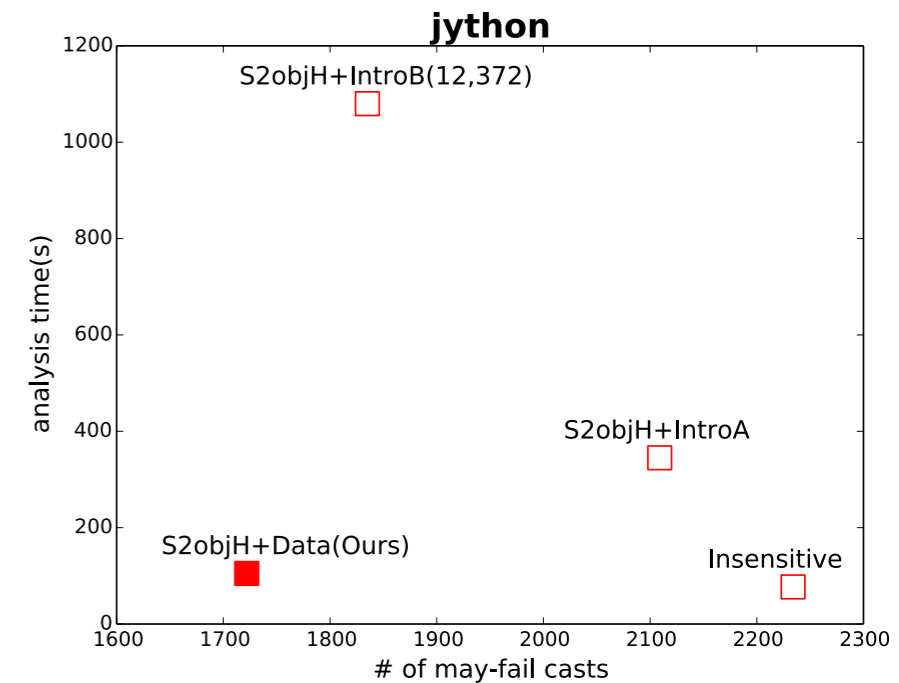
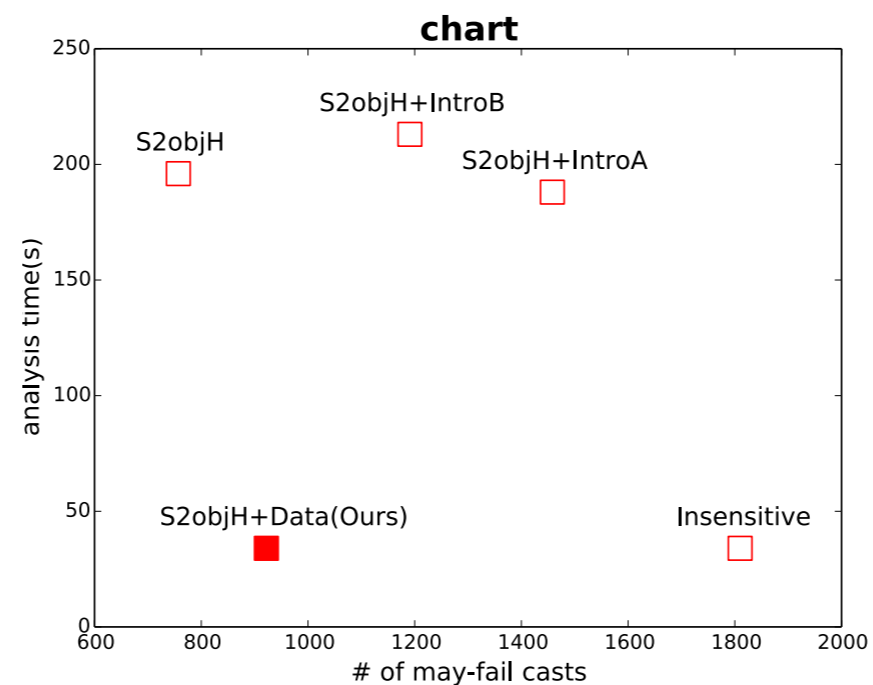
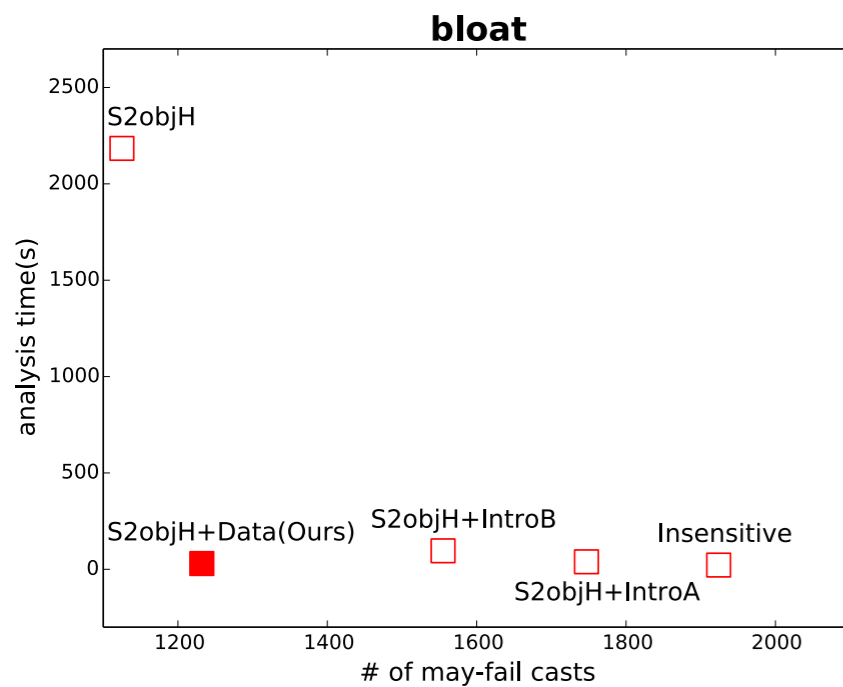
$$(\neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge 10 \wedge 11 \wedge 12 \wedge 13 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$$

$$(\neg 3 \wedge \neg 9 \wedge 13 \wedge 14 \wedge 15 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$$

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

Performance

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



Heuristics for Other Ctx-Sens

- **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):

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

cf) 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 fact from data:

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

Summary of Current Results

- **Techniques**

- Learning via black-box optimization [OOPSLA'15]
- Learning with disjunctive model [OOPSLA'17a]
- Learning with automatically generated features [OOPSLA'17b]
- Learning with automatically labelled data [ICSE'17, SAS'16]

- **Applications**

- flow-sensitivity, context-sensitivity, variable clustering, widening thresholds, unsoundness, search strategy, etc

Automatically Generating Search Heuristics for 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? ($0 \leq x, y \leq 100$)

< 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=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

x=22, y=7,
z=14

Symbolic
State

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

$x=22, y=7,$
 $z=14$

Symbolic
State

$x=\alpha, y=\beta, z=2*\beta$
 $2*\beta \neq \alpha$

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

Solve: $2*\beta = a$
Solution: $a=2, \beta=1$

$x=22, y=7,$
 $z=14$

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

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;
```

```
        }
```

```
    }
```

```
}
```

Concrete
State

x=2, y=1

Symbolic
State

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

x=2, y=1,
z=2

Symbolic
State

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

$x=2, y=1,$
 $z=2$

Symbolic
State

$x=\alpha, y=\beta, z=2*\beta$
 $2*\beta = \alpha$

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

$x=2, y=1,$
 $z=2$

Symbolic
State

$x=\alpha, y=\beta, z=2*\beta$

$2*\beta = \alpha \wedge$

$\alpha \leq \beta+10$

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

Solve: $2*\beta = a \wedge a > \beta+10$
Solution: $a=30, \beta=15$

$x=2, y=1,$
 $z=2$

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

$2*\beta = a \wedge$
 $a \leq \beta+10$

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

x=30, y=15

Symbolic
State

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

x=30, y=15,
z=30

Symbolic
State

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

x=30, y=15,
z=30

Symbolic
State

x=α, y=β, z=2*β
2*β = α

3rd iteration

Concolic Testing

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

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

Concrete
State

error-triggering
input

x=30, y=15,
z=30

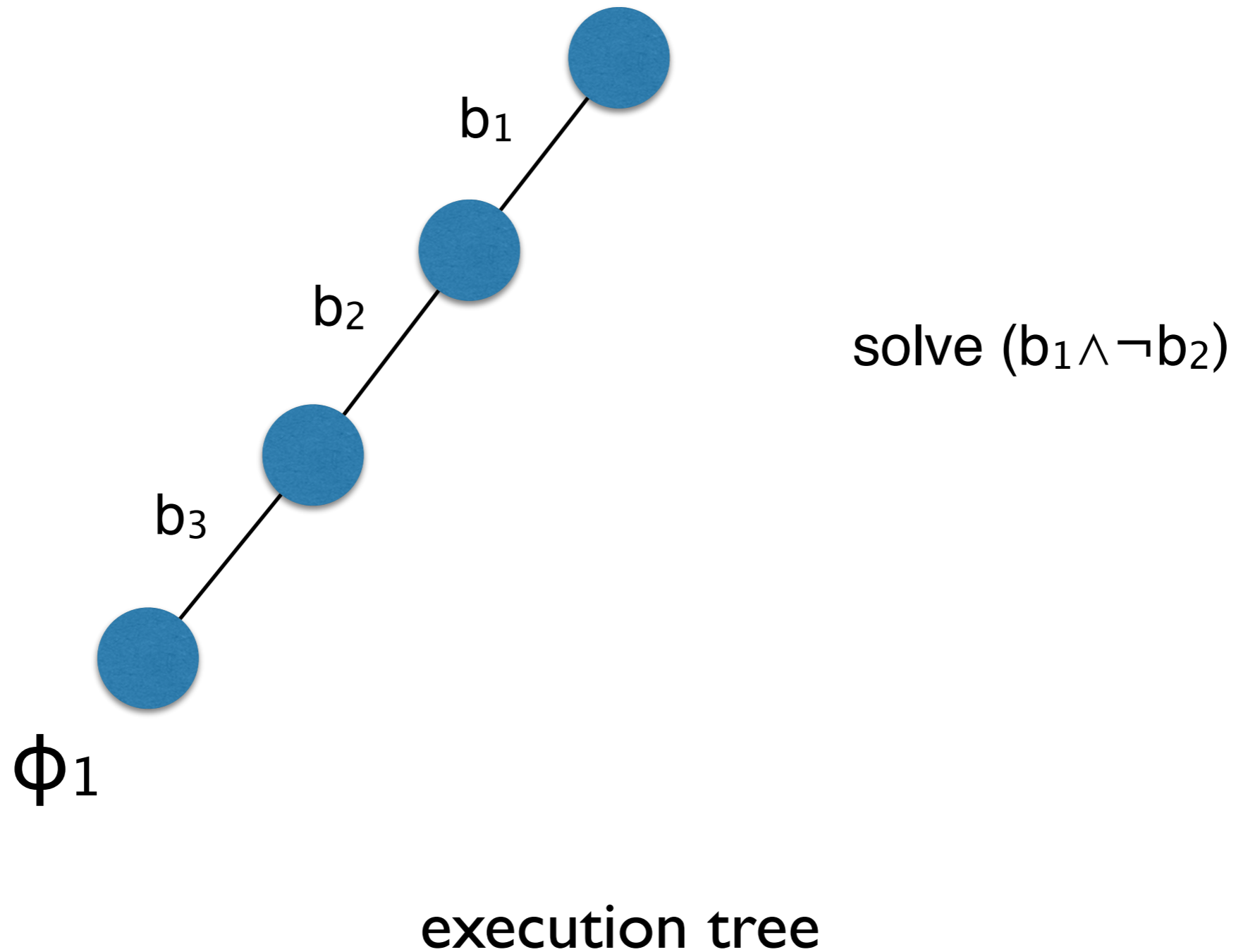
Symbolic
State

$x=\alpha, y=\beta, z=2*\beta$

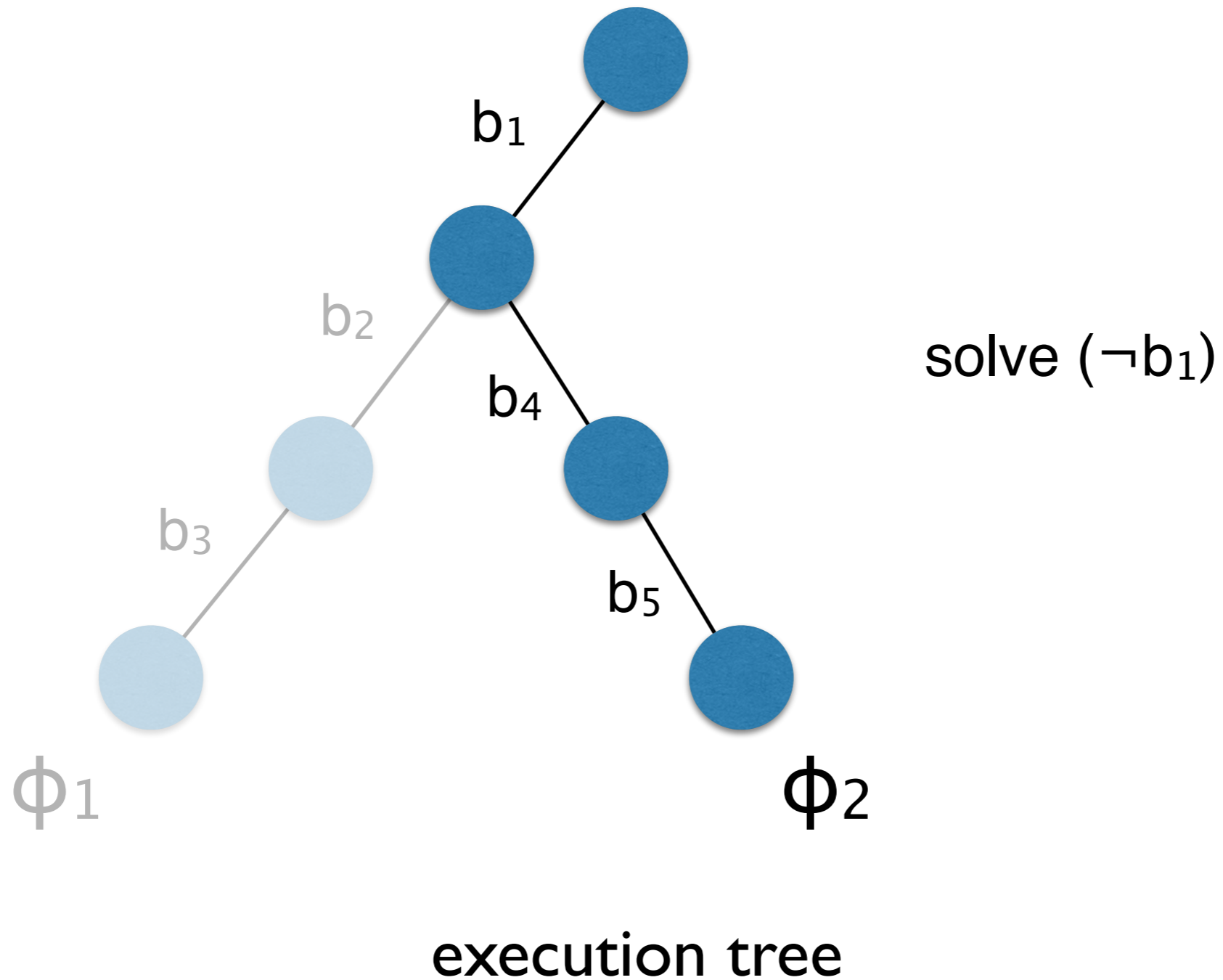
$2*\beta = \alpha \wedge$
 $\alpha > \beta+15$

3rd iteration

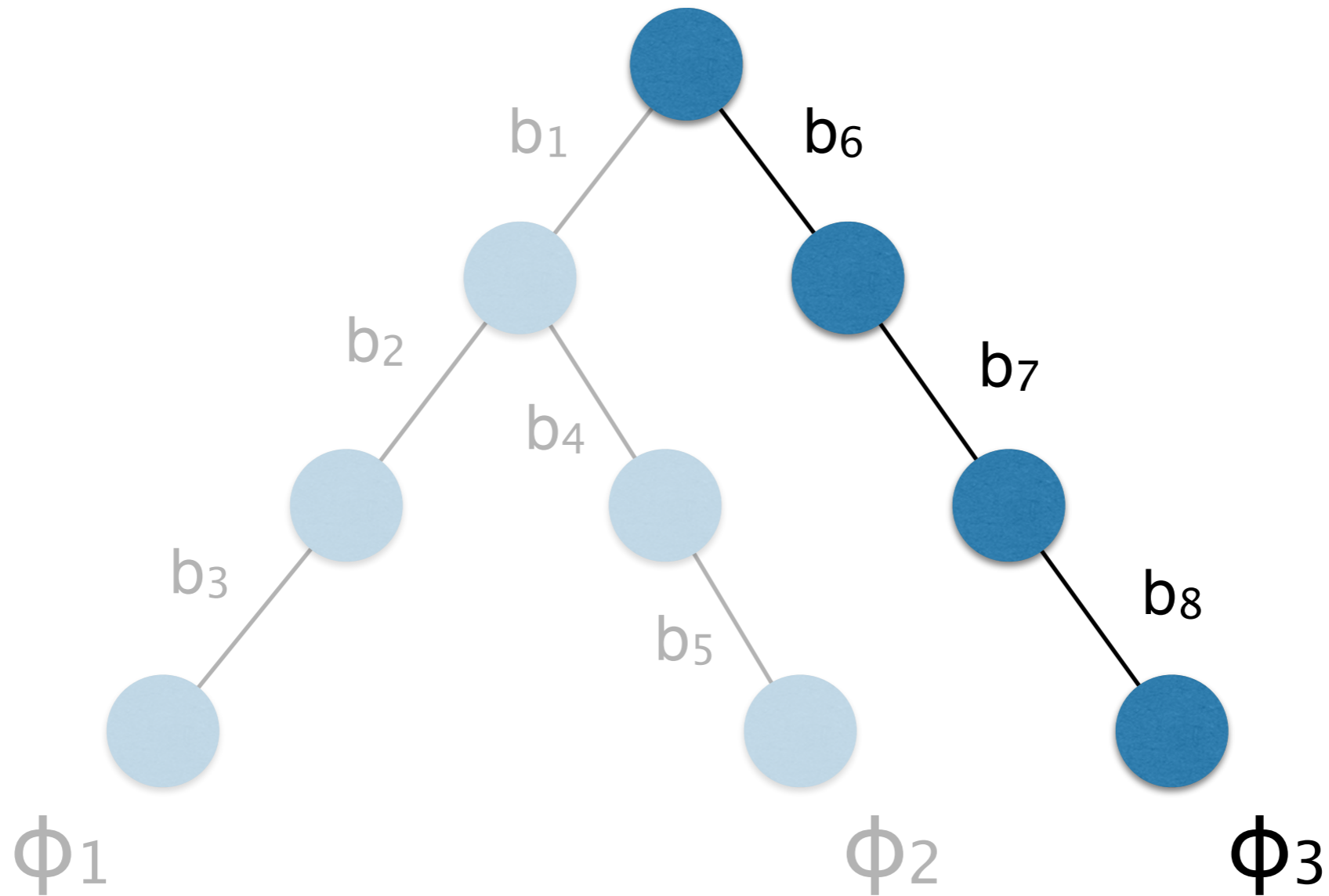
Concolic Testing



Concolic Testing



Concolic Testing



execution tree

Concolic Testing Algorithm

Input : Program P , initial input vector v_0 , budget N

Output: The number of branches covered

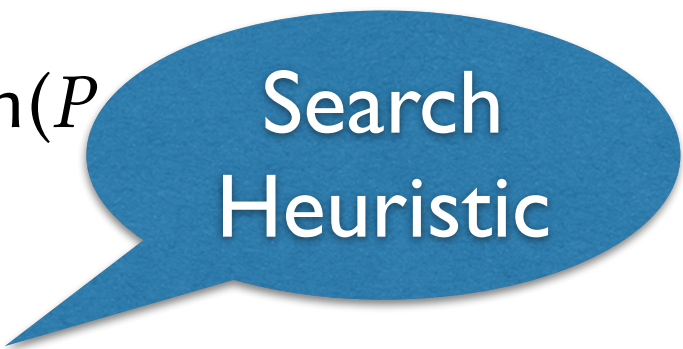
```
1:  $T \leftarrow \langle \rangle$ 
2:  $v \leftarrow v_0$ 
3: for  $m = 1$  to  $N$  do
4:    $\Phi_m \leftarrow \text{RunProgram}(P, v)$ 
5:    $T \leftarrow T \cdot \Phi_m$ 
6:   repeat
7:      $(\Phi, \phi_i) \leftarrow \text{Choose}(T)$       ( $\Phi = \phi_1 \wedge \dots \wedge \phi_n$ )
8:     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  $|\text{Branches}(T)|$ 
```

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:    $\Phi_m \leftarrow \text{RunProgram}(P, v)$ 
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  $|\text{Branches}(T)|$ 
```



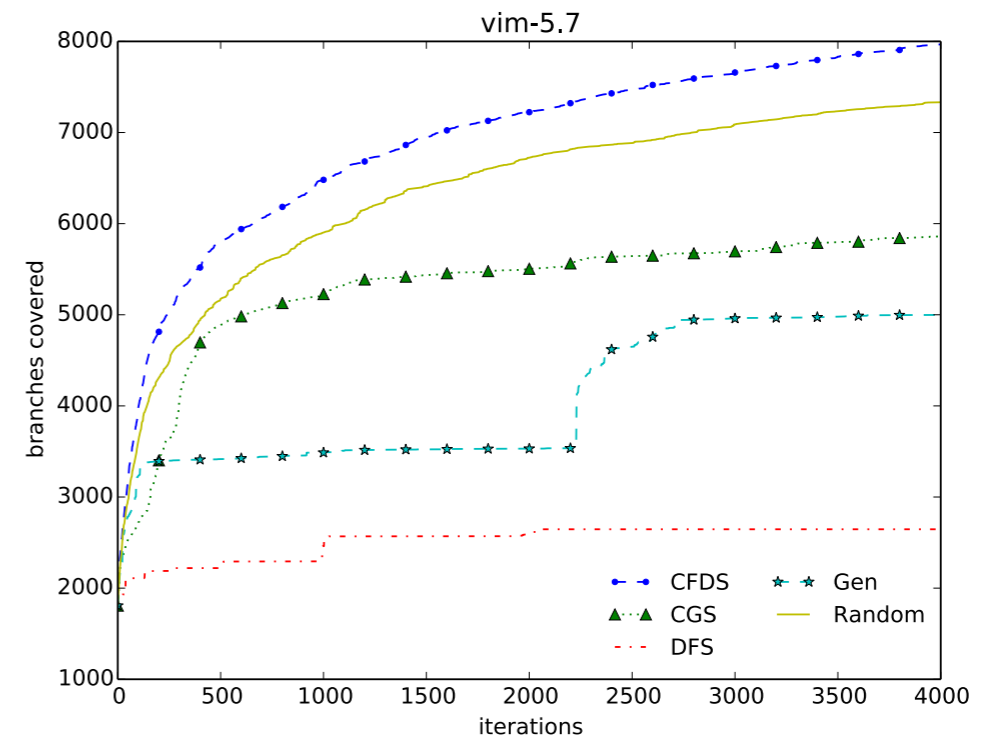
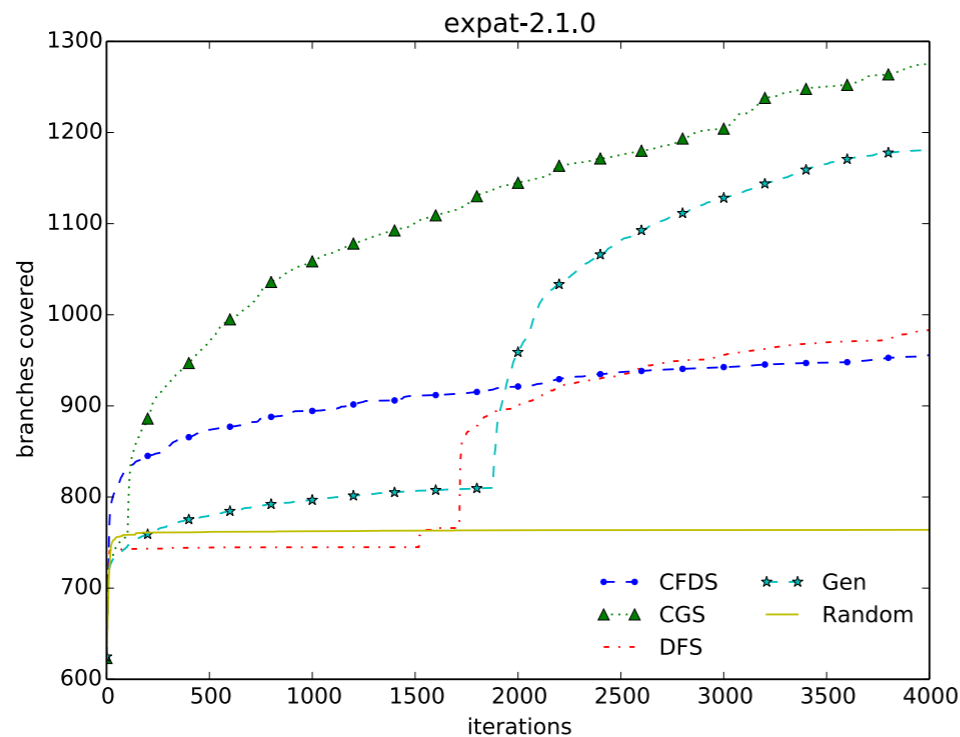
Search
Heuristic

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:
 - E.g., DFS [PLDI'05], BFS, Random, CFDS [ASE'08], Generational [NDSS'08], CarFast[FSE'12], CGS [FSE'14], ...

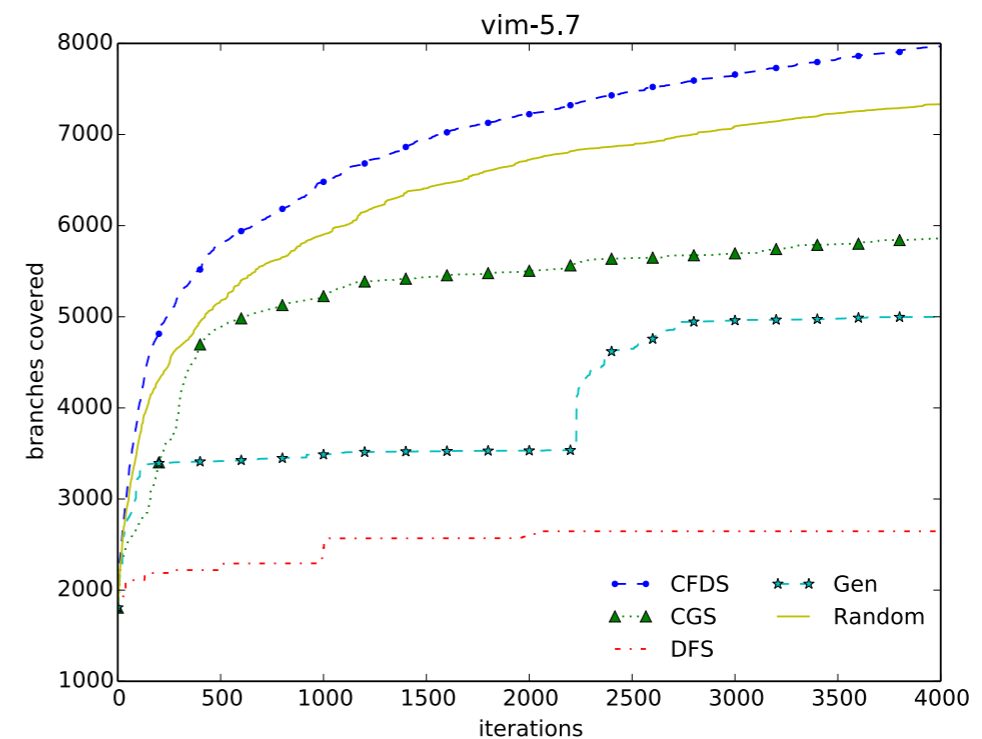
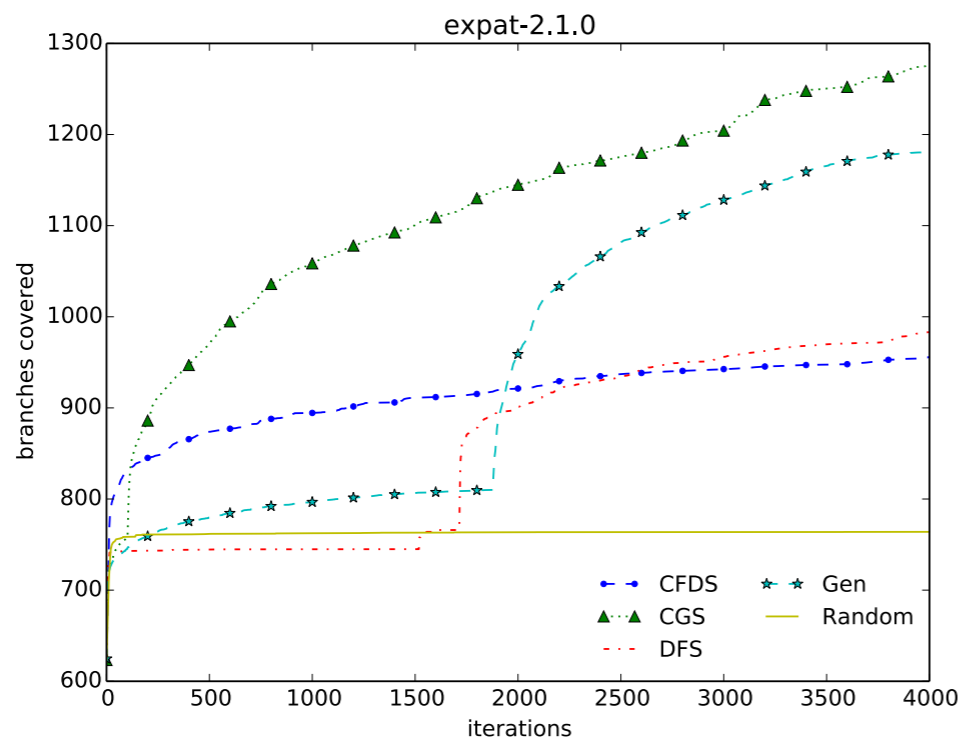
Existing Limitations

- Manually developing a search heuristic requires a huge amount of engineering effort and expertise.
- Manual approaches are suboptimal and unstable.



Existing Limitations

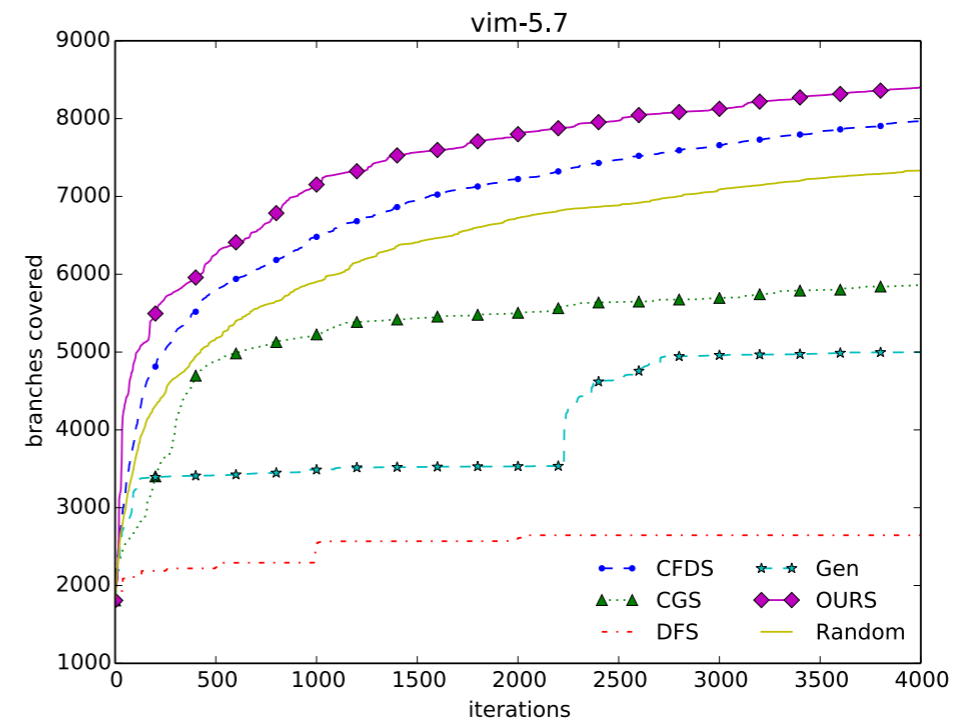
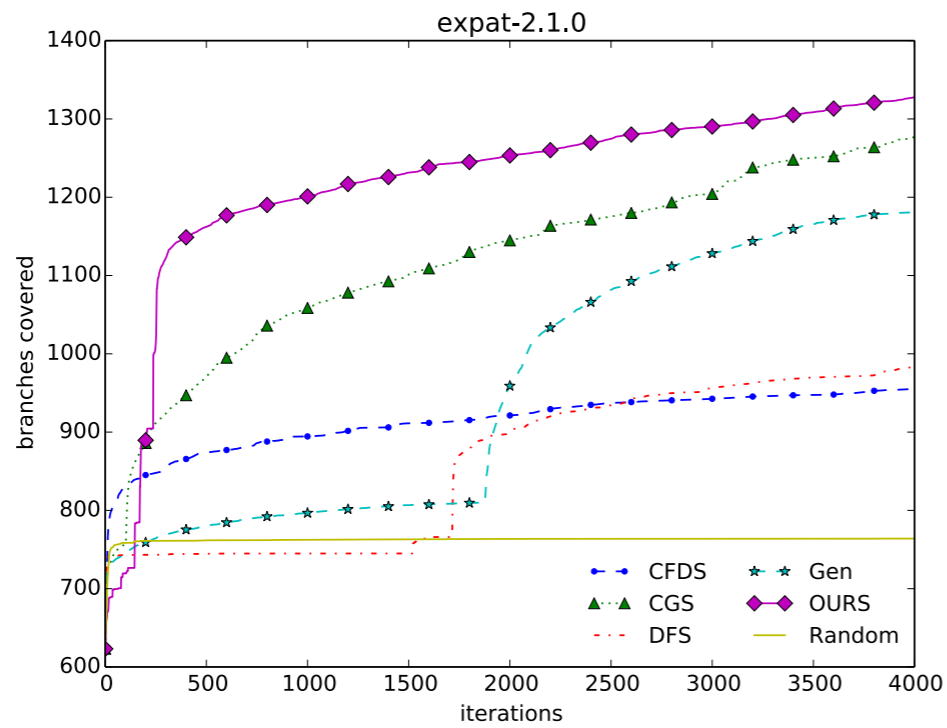
- Manually developing a search heuristic requires a huge amount of engineering effort and expertise.
- Manual approaches are suboptimal and unstable.



Our goal: automatically generating search heuristics

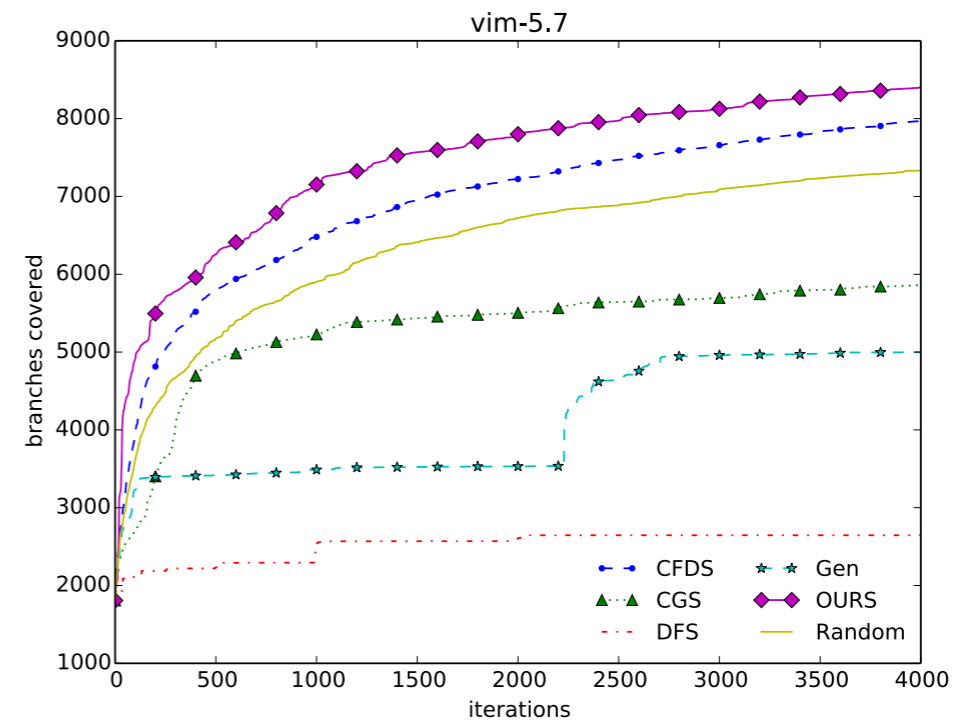
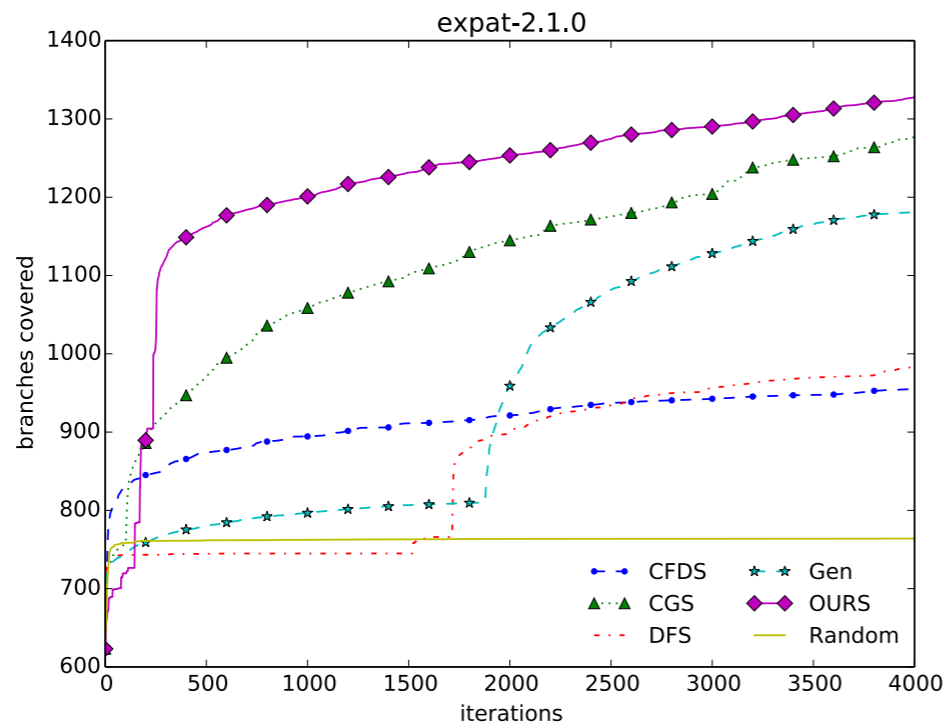
Effectiveness

- Considerable increase in branch coverage



Effectiveness

- Considerable increase in branch coverage

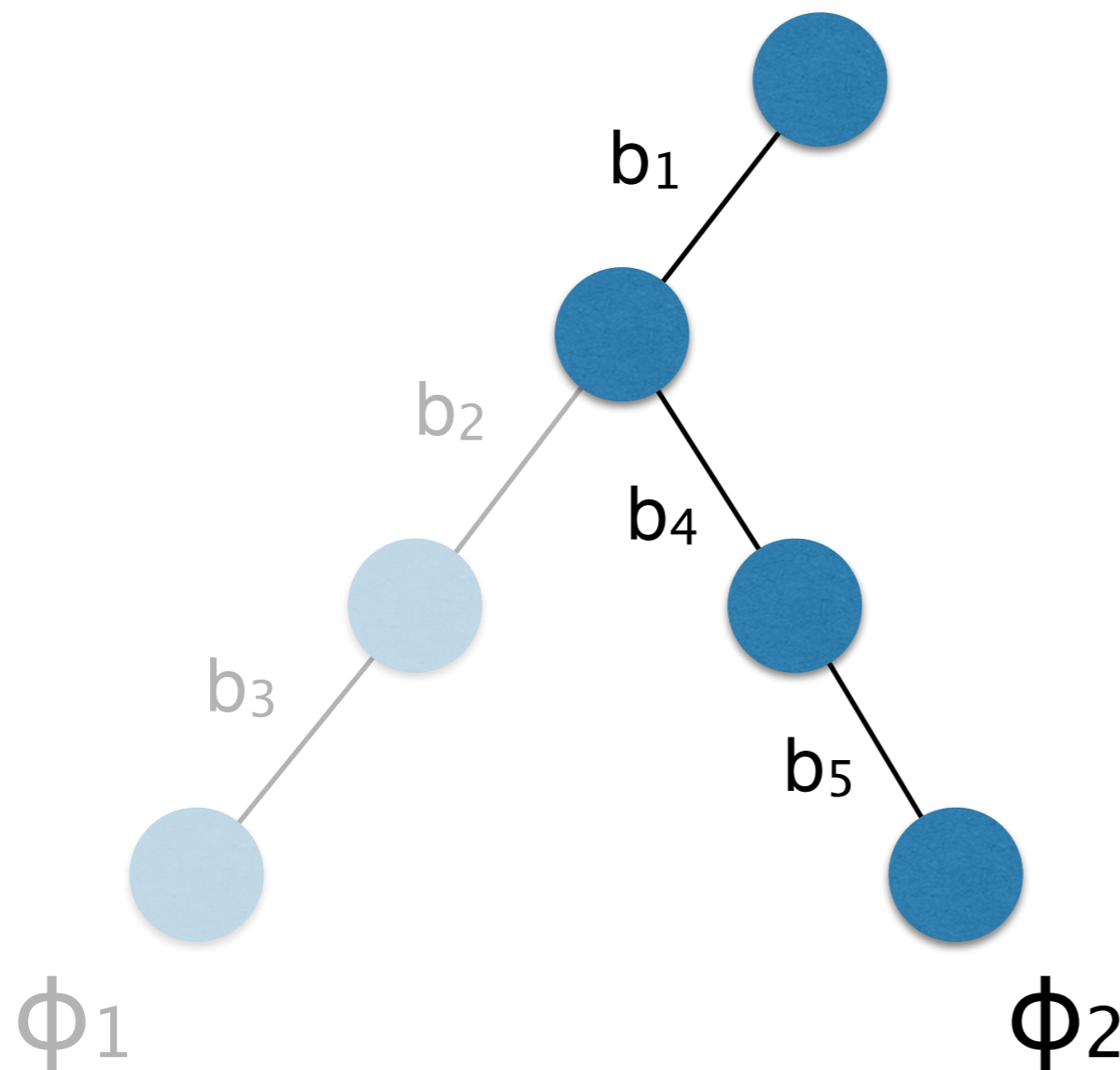


- Dramatic increase in bug-finding

	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

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



$$\text{score}_\theta(b_1) = 1.3$$

$$\text{score}_\theta(b_4) = 0.0$$

$$\text{score}_\theta(b_5) = 0.7$$

(I) 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

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

- 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 from code
- 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

$$\text{score}_{\theta}(\phi) = \pi(\phi) \cdot \theta$$

$$\text{score}_{\theta}(\mathbf{b}_1) = \langle 1, 0, 1, 1, 0 \rangle \cdot \langle 0.8, -0.5, 0.3, 0.2, -0.7 \rangle = 1.3$$

$$\text{score}_{\theta}(\mathbf{b}_4) = \langle 0, 1, 1, 1, 0 \rangle \cdot \langle 0.8, -0.5, 0.3, 0.2, -0.7 \rangle = 0.0$$

$$\text{score}_{\theta}(\mathbf{b}_5) = \langle 1, 0, 0, 0, 1 \rangle \cdot \langle 0.8, -0.5, 0.3, 0.2, -0.7 \rangle = 0.1$$

Optimization Algorithm

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

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

where

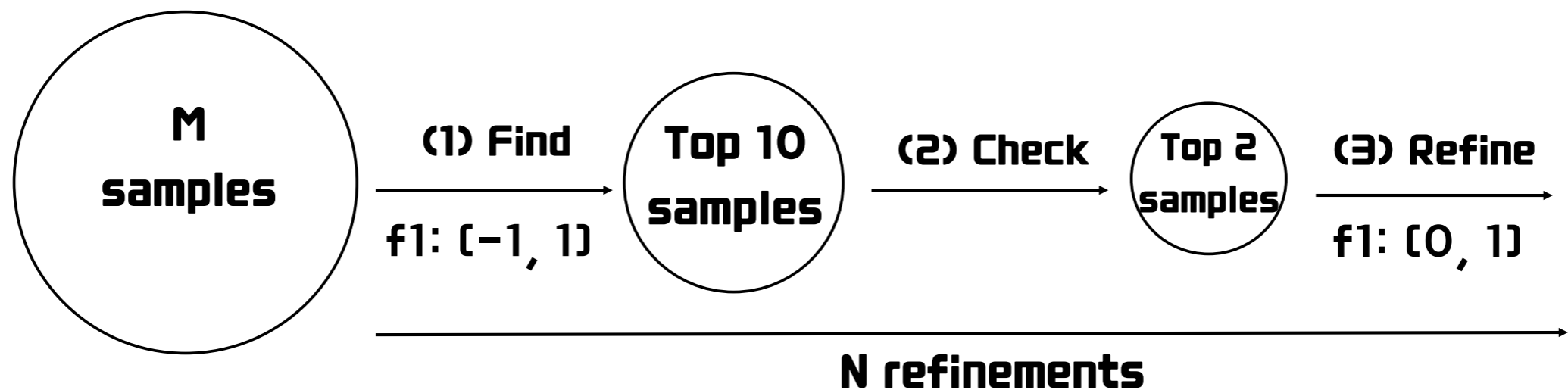
$$C : \textit{Program} \times \textit{SearchHeuristic} \rightarrow \textit{Coverage}$$

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 θ found
- Failed to find good parameters due to large search space

Our Algorithm

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



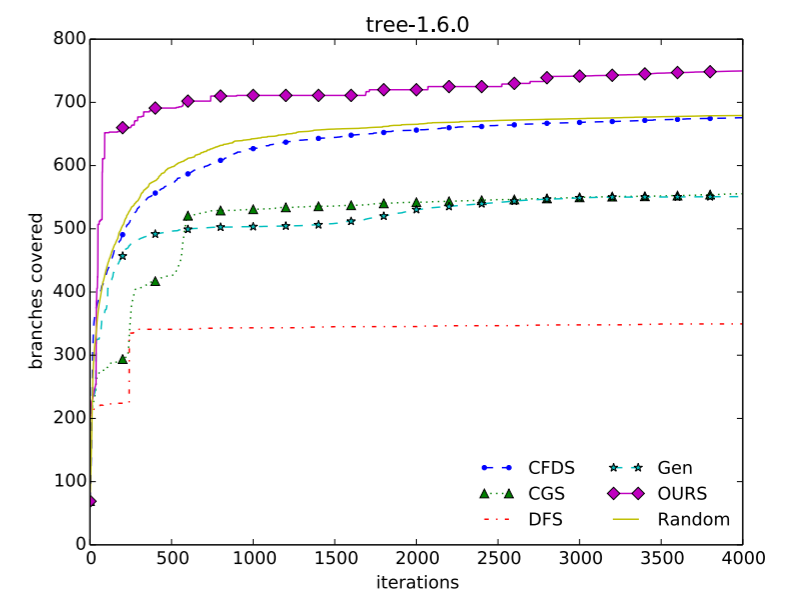
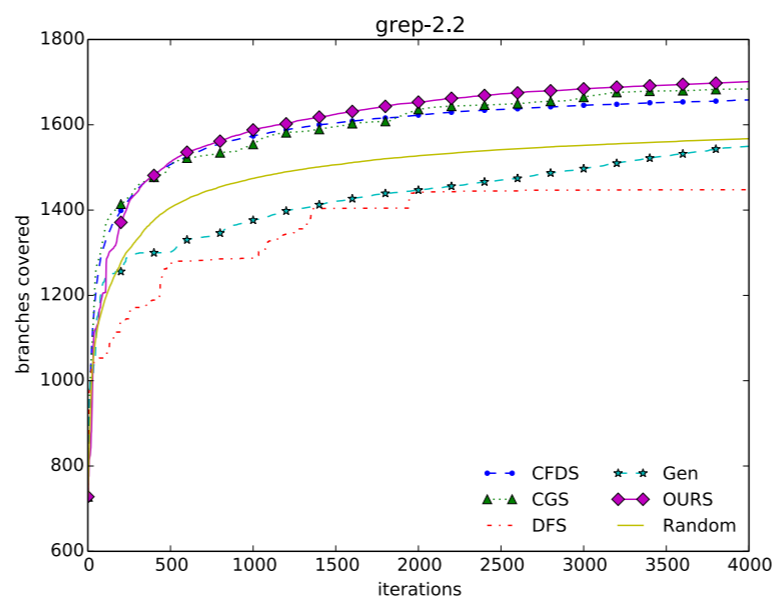
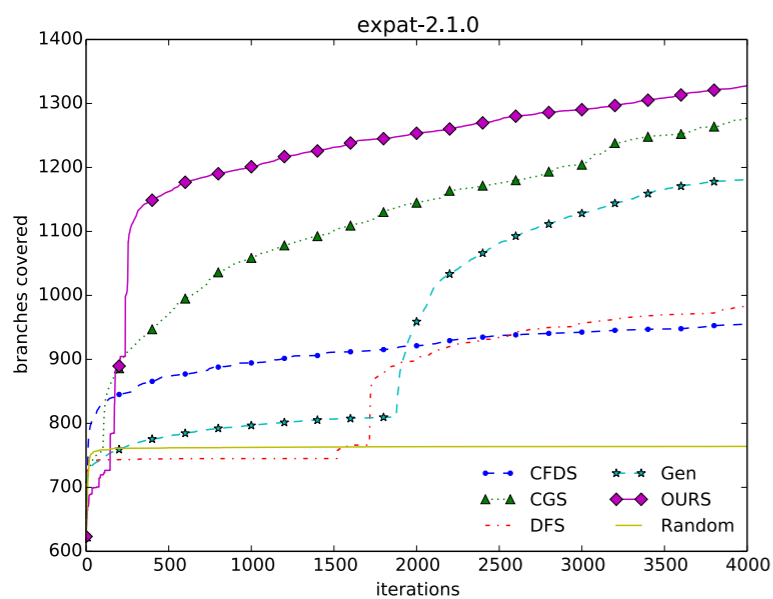
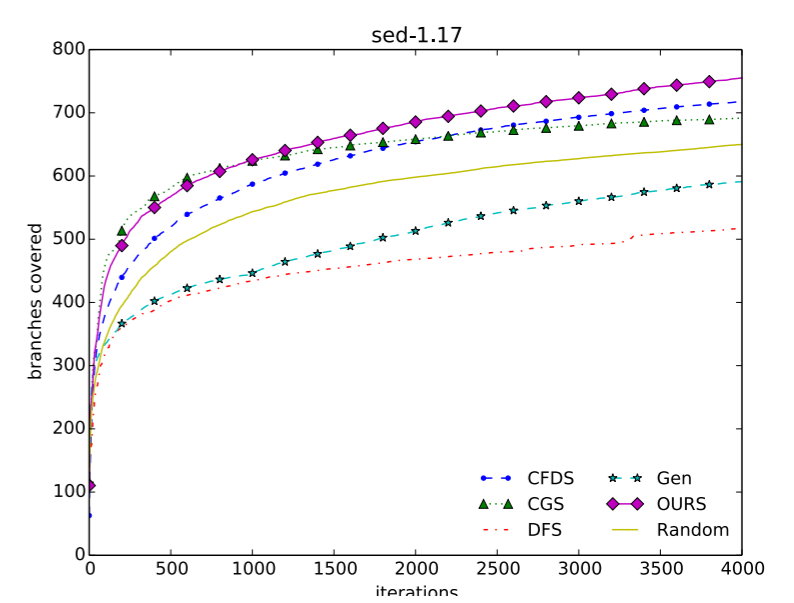
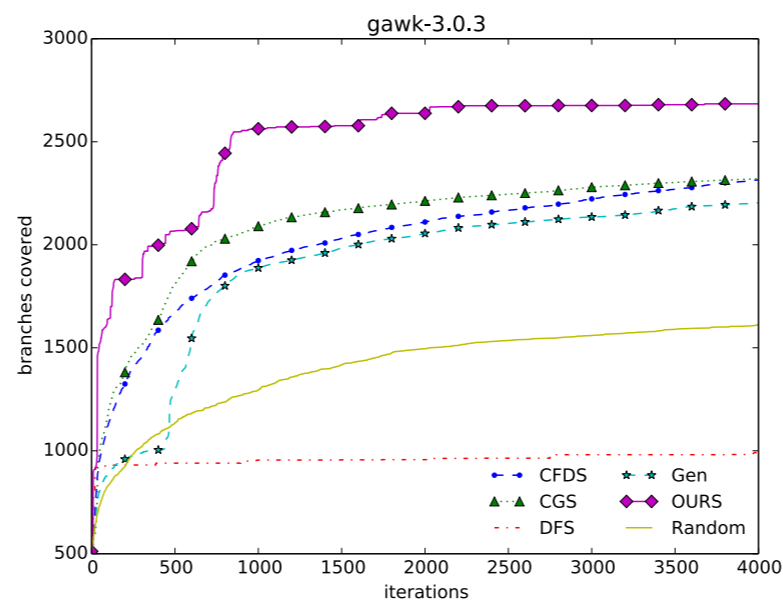
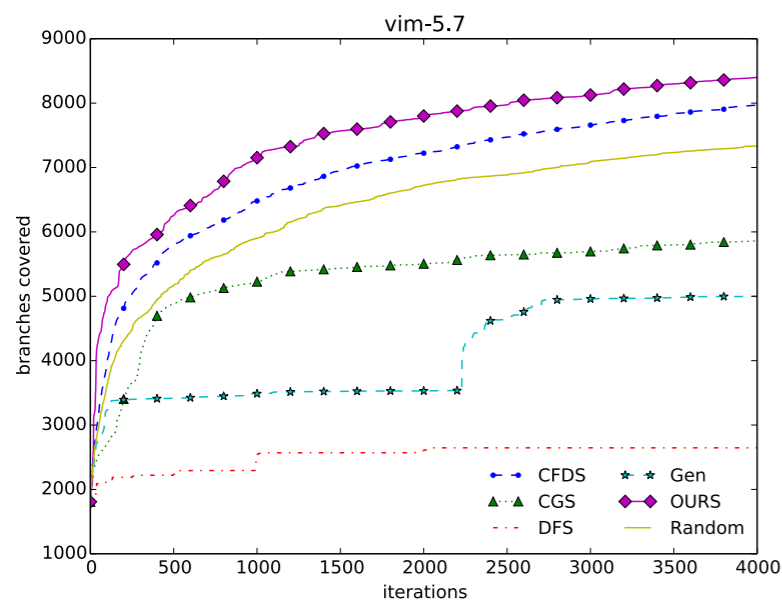
Experiments

- Implemented in CREST
- Heuristics were generated for each benchmark
- Compared with 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 of Generated Heuristics

- Branch coverage averaged over 100 runs (50 for vim)



Effectiveness of Generated Heuristics

- 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

- Average coverage 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 of Generated Heuristics

- Higher branch coverage leads to 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

Generation Time

- Our approach requires training cost.
- Time for obtaining the heuristics (with 20 cores):

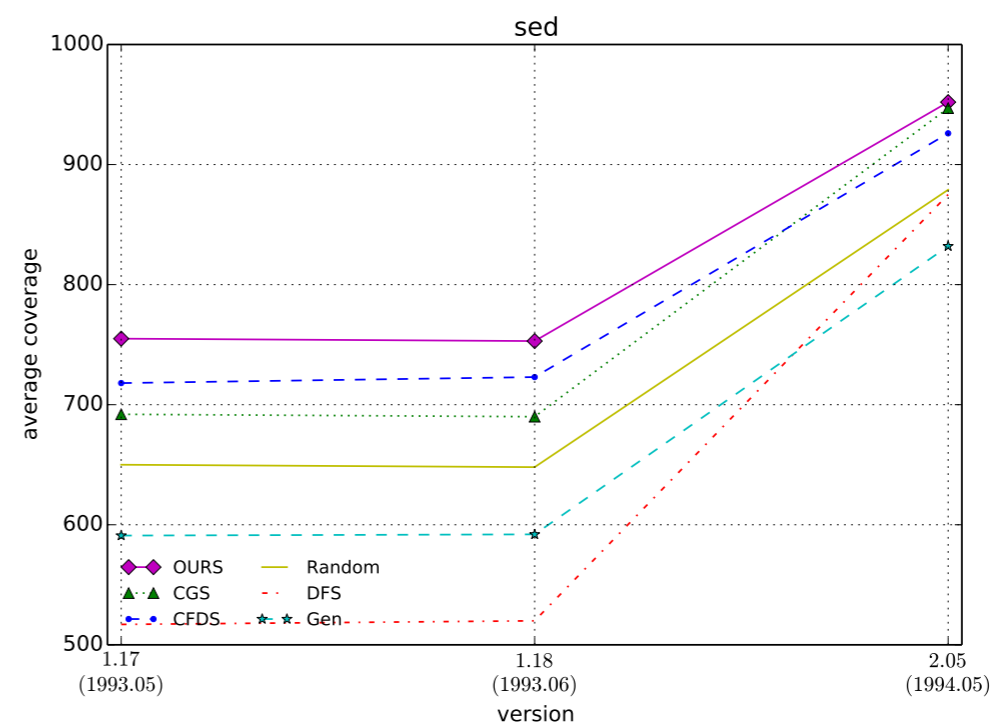
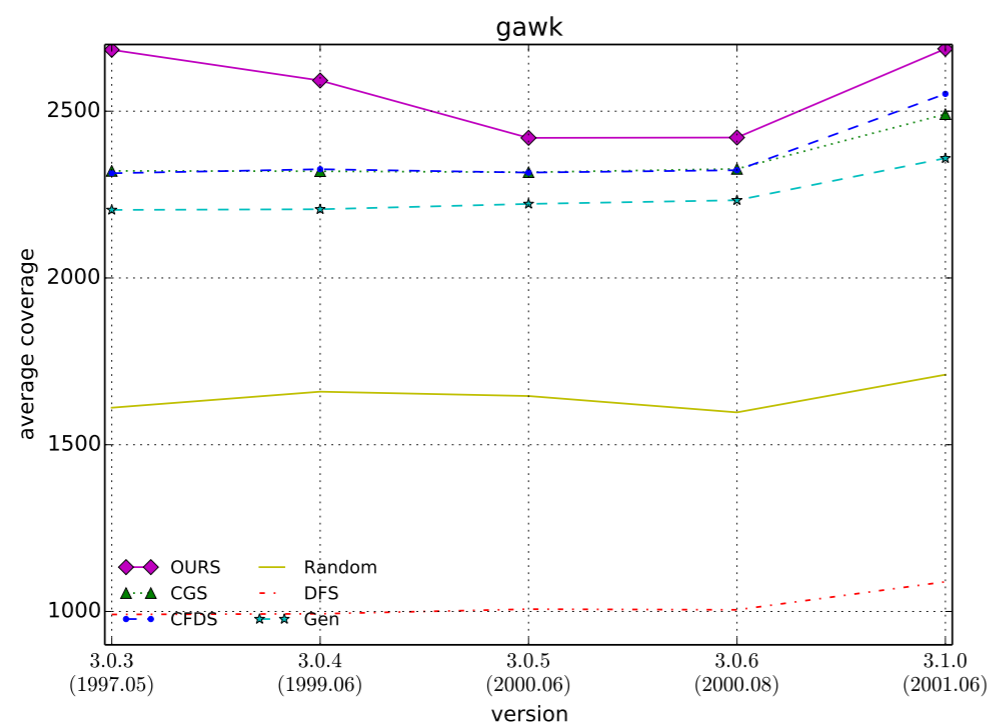
Benchmarks	# Sample	# Iteration	Total times
vim-5.7	300	5	24h 18min
expat-2.1.0	1,000	6	10h 25min
gawk-3.0.3	1,000	4	6h 30min
grep-2.2	1,000	5	5h 24min
sed-1.17	1,000	4	8h 54min
tree-1.6.0	1,000	4	3h 18min

Still useful

- 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

- Reusable as programs evolve



Summary and Future Plan

- Program analyzers are designed by analysis designers based on their limited insights on target programs
 - Not tuned for programs that are actually analyzed
- Our vision: “Synthesize” program analyzers from data
 - Every design decisions is parameterized and learned from actual data



Summary and Future Plan

- Program analyzers are designed by analysis designers based on their limited insights on target programs
 - Not tuned for programs that are actually analyzed
- Our vision: “Synthesize” program analyzers from data
 - Every design decisions is parameterized and learned from actual data



Thank you