Automatically Generating Search Heuristics for Concolic Testing
(+ program analysis, synthesis, and repair)

Hakjoo Oh

Programming Research Laboratory
Korea University

Aug. 8, 2017 @Samsung SW Center

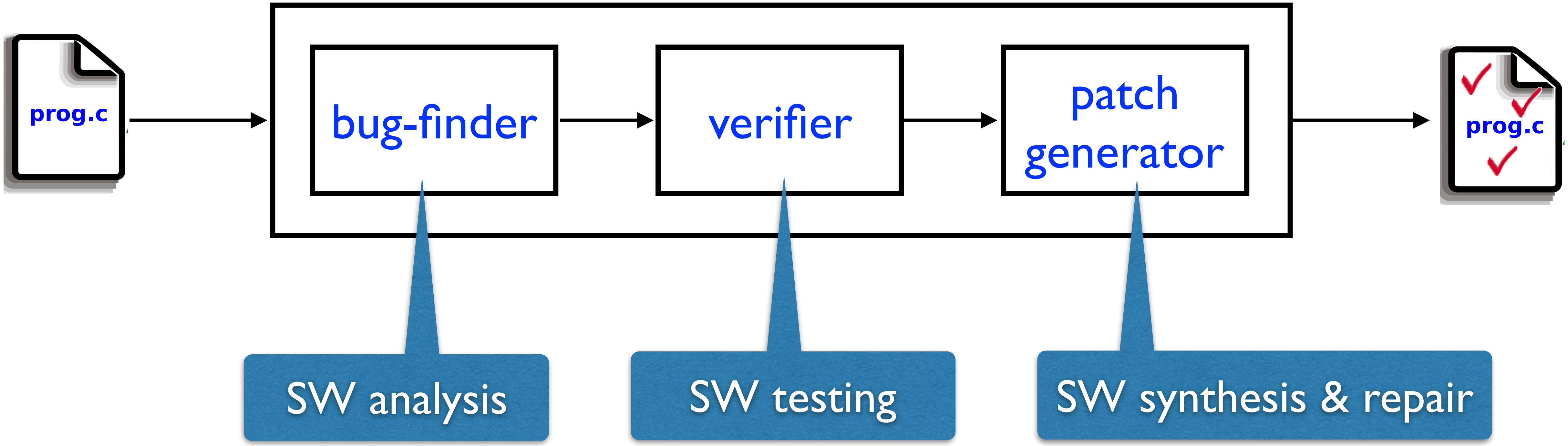
(co-work with Sooyoung Cha, Kwonsoo Chae, Kihong Heo, Seongjoon Hong, Minseok Jeon, Sehun Jeong, Junhee Lee, Jonho Lee, Sunbeom So, Donwon Song, Hongseok Yang)

Our Research

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

Our Long-term Goal

- Achieving technologies for automatically finding, verifying, and fixing software errors and vulnerabilities



Today: Concolic Testing

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

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

Limitation of Random Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

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

Limitation of Random Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

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

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

1st iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

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

1st iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

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

1st iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

Concrete State	Symbolic State
x=22, y=7, z=14	$x=\alpha, y=\beta, z=2*\beta$
	$2*\beta \neq \alpha$

Solve: $2*\beta = \alpha$

Solution: $\alpha=2, \beta=1$

1st iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

Concrete State	Symbolic State
$x=2, y=1$	$x=\alpha, y=\beta$
	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	Symbolic State
x=2, y=1, z=2	x=α, y=β, z=2*β
	true

2nd iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

Concrete State	Symbolic State
x=2, y=1, z=2	$x=\alpha, y=\beta, z=2^*\beta$
	$2^*\beta = \alpha$

2nd iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

Concrete State	Symbolic State
x=2, y=1, z=2	$x=\alpha, y=\beta, z=2*\beta$
	$2*\beta = \alpha \wedge \alpha \leq \beta+10$

2nd iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

Concrete State	Symbolic State
$x=2, y=1, z=2$	$x=\alpha, y=\beta, z=2^*\beta$
	$2^*\beta = \alpha \wedge \alpha \leq \beta+10$

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

Solution: $\alpha=30, \beta=15$

2nd iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

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

3rd iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

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

3rd iteration

Concolic Testing

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

Concrete State

$x=30, y=15,$
$z=30$

Symbolic State

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

$2^*\beta = \alpha$

3rd iteration

Concolic Testing

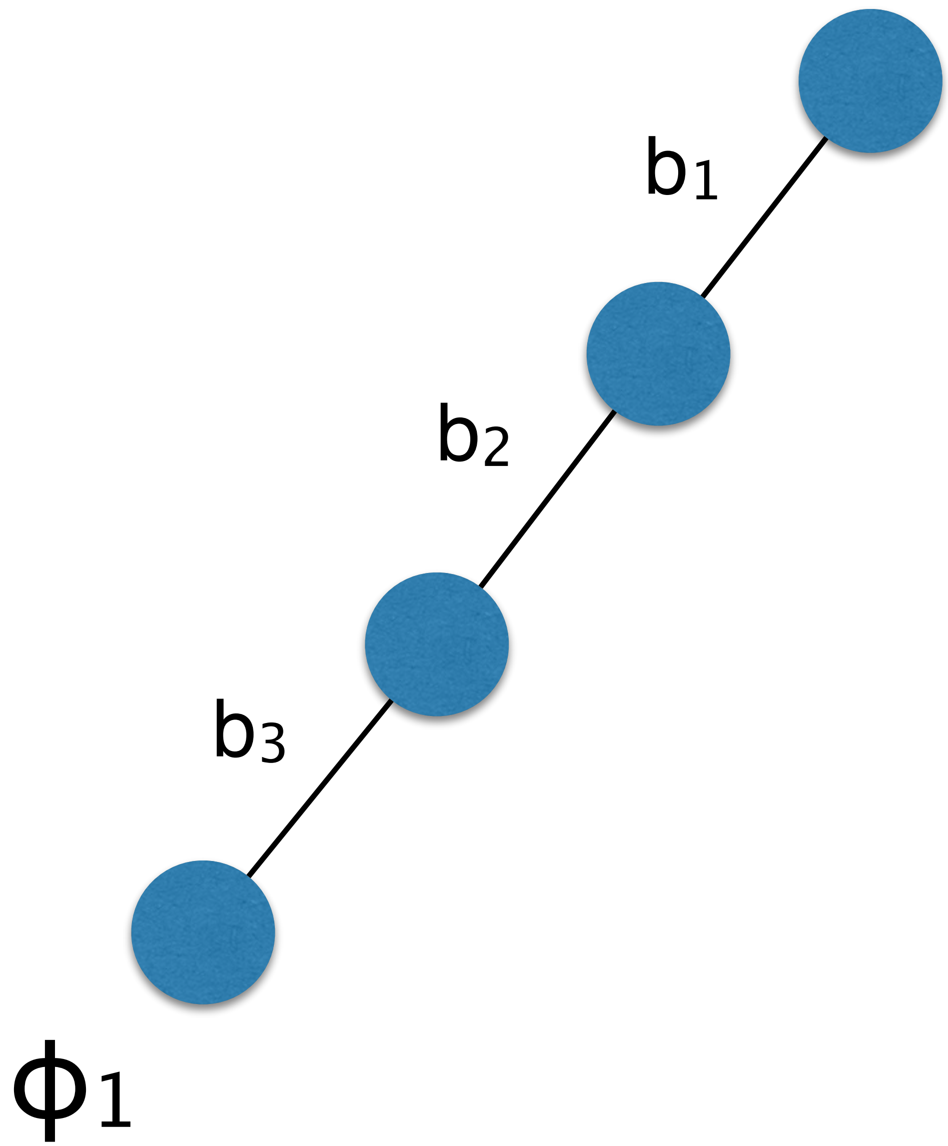
```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

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

error-triggering input

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

3rd iteration

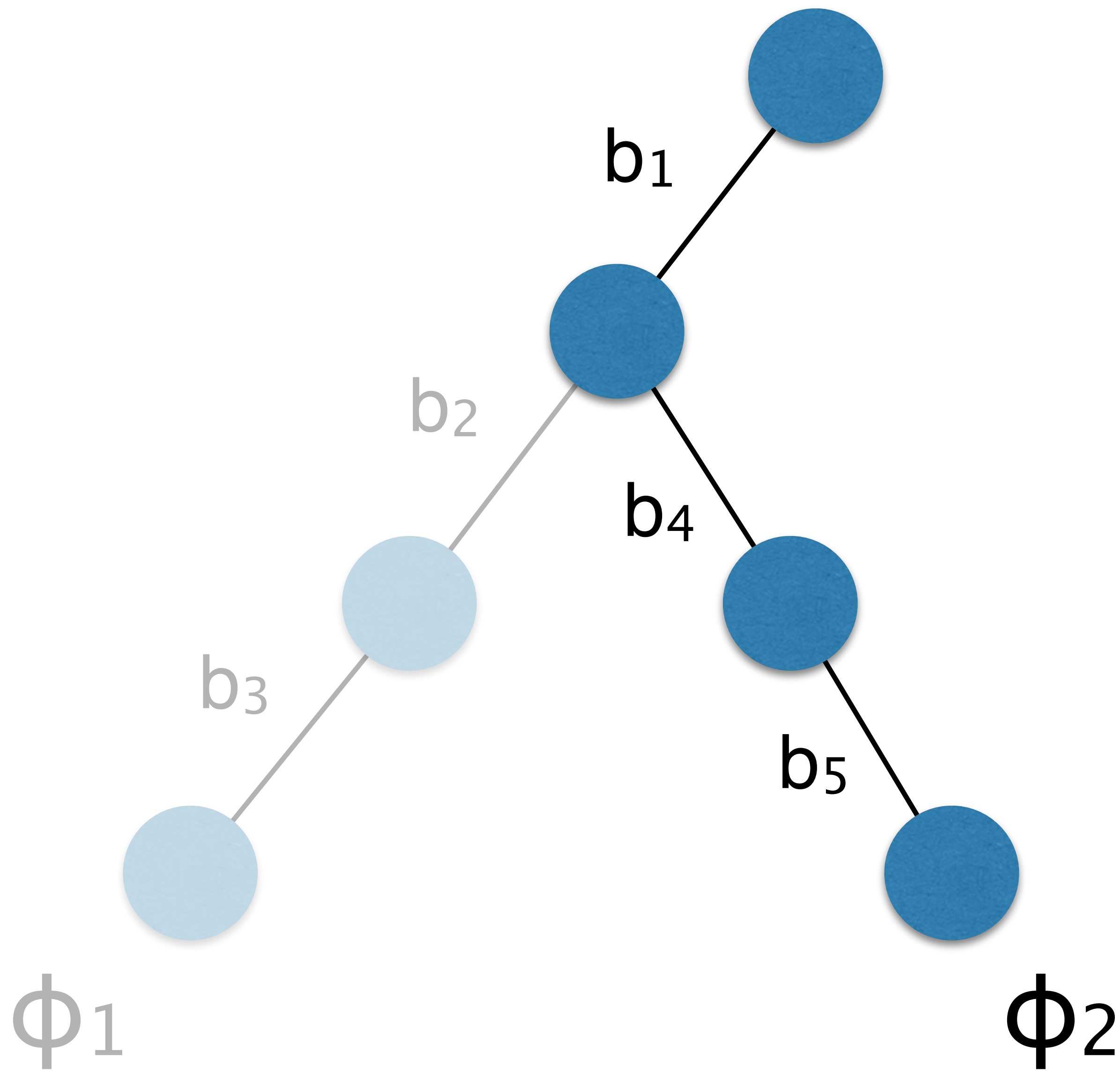
Concolic Testing



solve ($b_1 \wedge \neg b_2$)

execution tree

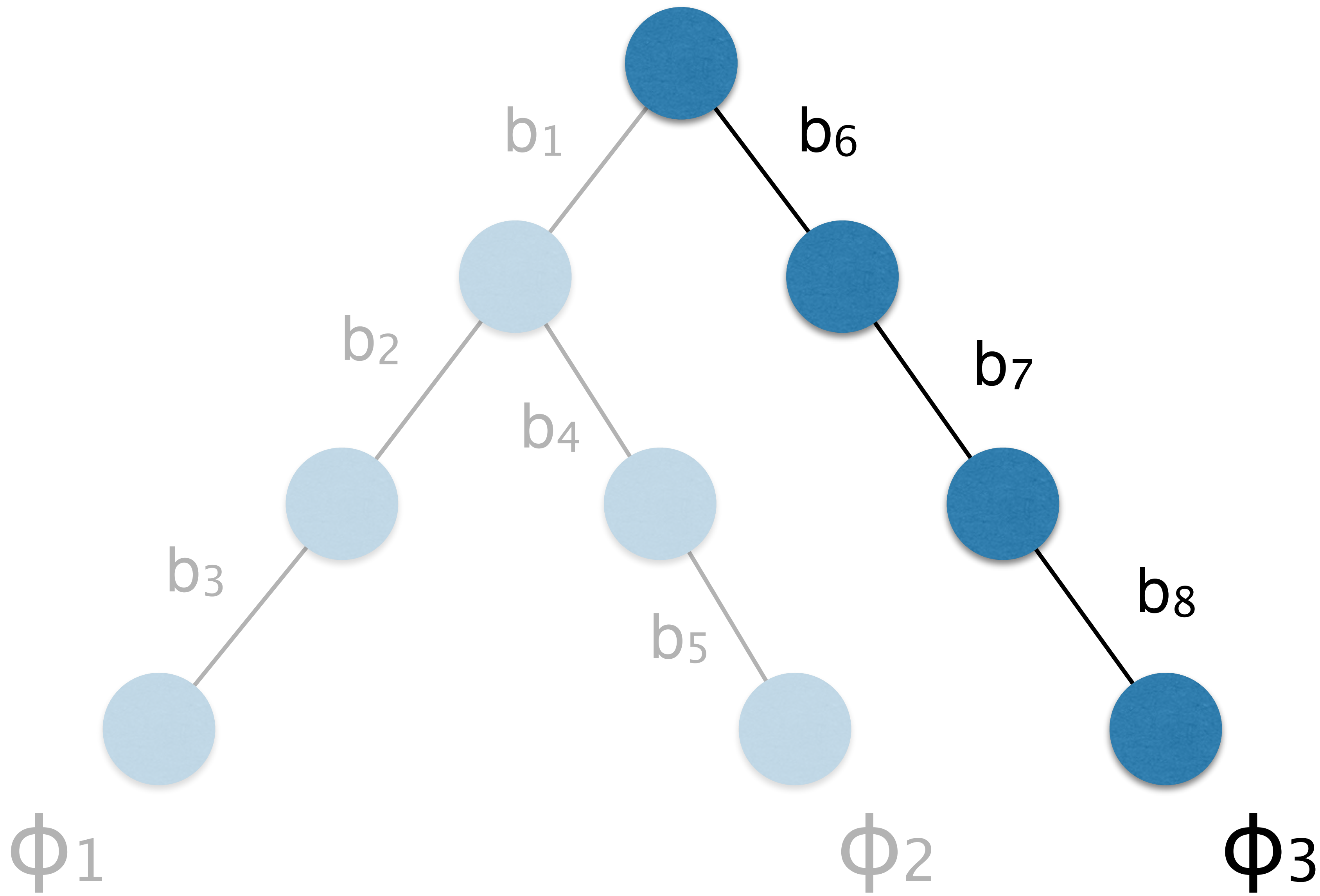
Concolic Testing



solve ($\neg b_1$)

execution tree

Concolic Testing



execution tree

Concolic Testing Algorithm

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

Output : The number of branches covered

$T \leftarrow \langle\rangle$

$v \leftarrow v_0$

for $m = 1$ to N **do**

  $\Phi_m \leftarrow \text{RunProgram}(P, v)$

  $T \leftarrow T \cdot \Phi_m$

  repeat

    $(\Phi, \phi_i) \leftarrow \text{Choose}(T) \qquad (\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$

  until $\text{SAT}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$

  $v \leftarrow \text{model}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$

end for

return $|\text{Branches}(T)|$

Concolic Testing Algorithm

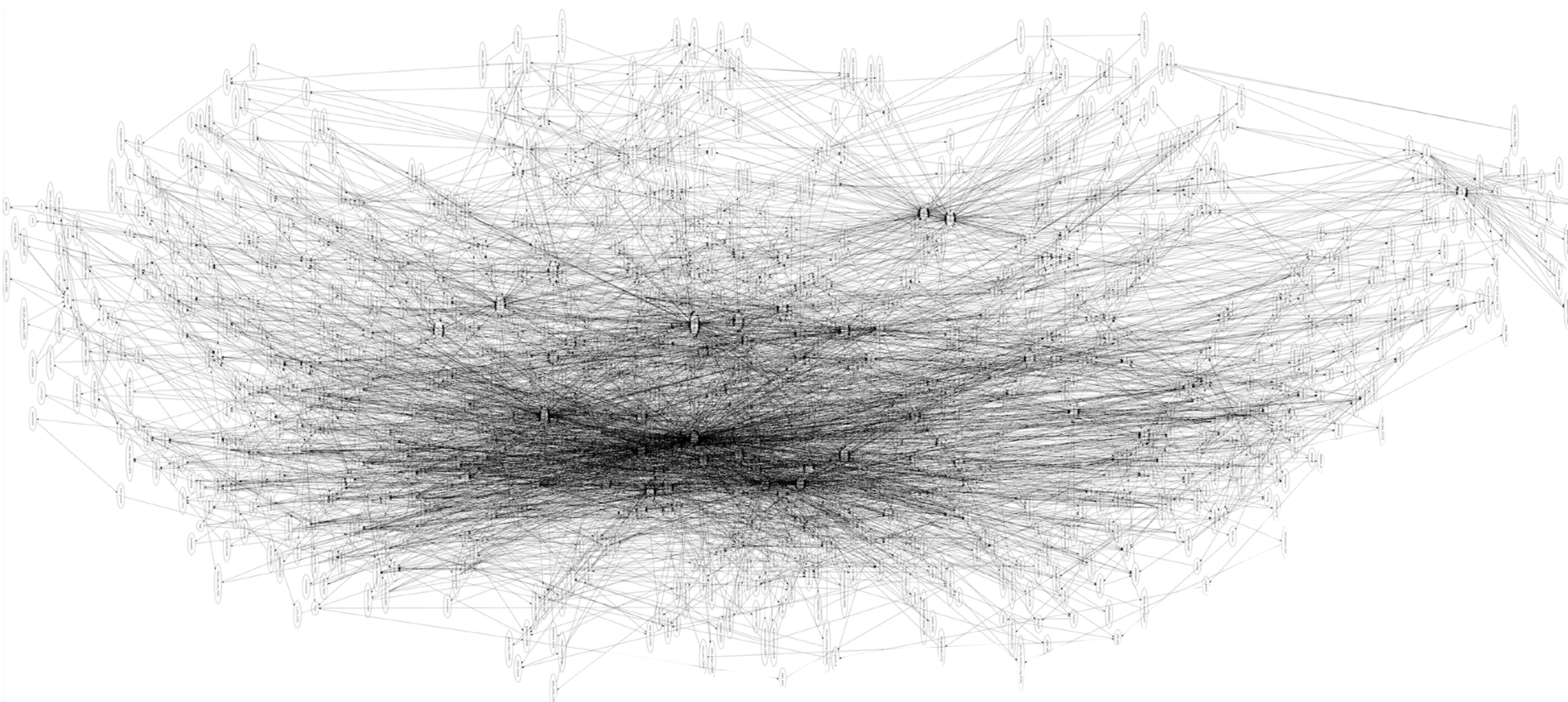
Input : Program P, initial input vector v_0, budget N
Output : The number of branches covered

```
T ← ⟨⟩
v ← v_0
for m = 1 to N do
    Φ_m ← RunProgram(P
    T ← T · Φ_m
    repeat
        (Φ, φ_i) ← Choose(T)        (Φ = φ_1 ∧ ··· ∧ φ_n)
    until SAT(⋀_{j<i} φ_j ∧ ¬φ_i)
    v ← model(⋀_{j<i} φ_j ∧ ¬φ_i)
end for
return |Branches(T)|
```

Search Heuristic

Path Explosion

- Concolic testing relies on search heuristics to maximize code coverage in a limited budget.



Existing Search Heuristics

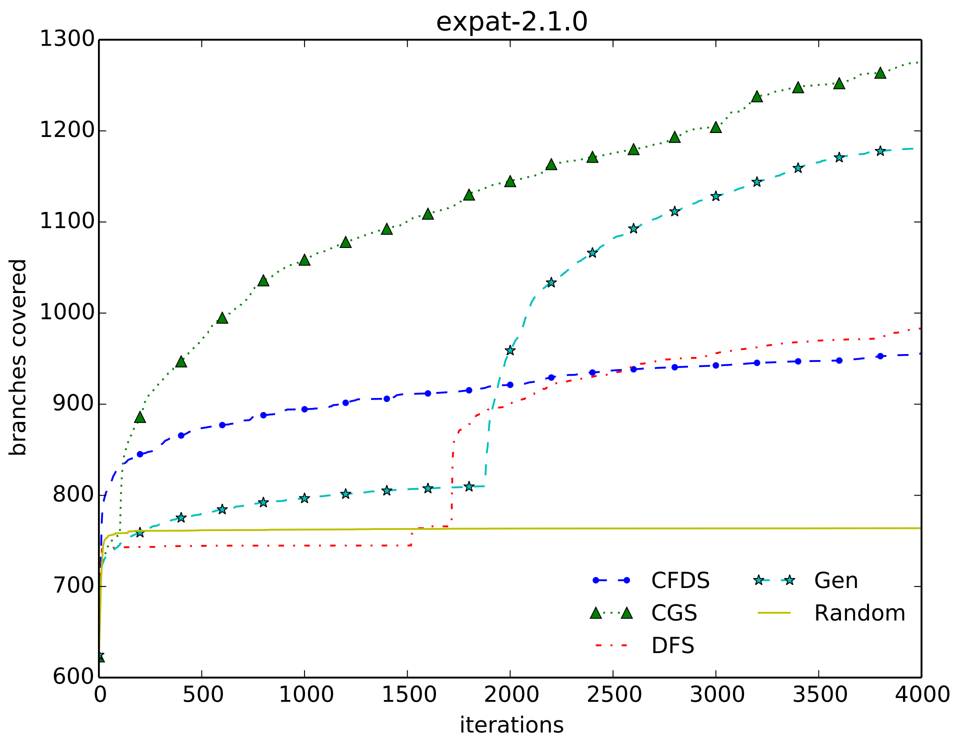
- Numerous heuristics have been proposed, e.g.,
  - DFS, BFS, Random, Generational, CFDS, CGS, etc
- CFDS (Control-Flow-Directed Search) [1]
  - selects a branch whose opposite branch is the nearest from the unseen branches
- CGS (Context-Guided Search) [2]
  - basically performs BFS while excluding branches whose contexts are previously explored

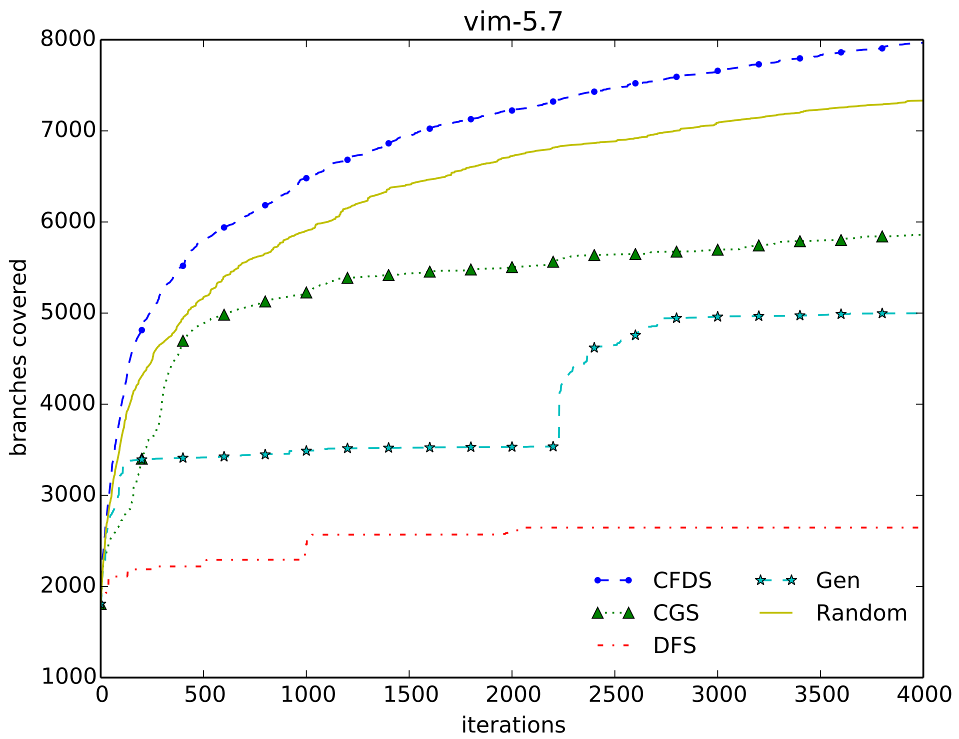
---

[1] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. ASE 2008.
[2] Hyunmin Seo and Sunghun Kim. How we get there: A context-guided search strategy in con colic testing. FSE 2014

Limitations of Existing Search Heuristics

- No existing heuristics consistently perform well in practice





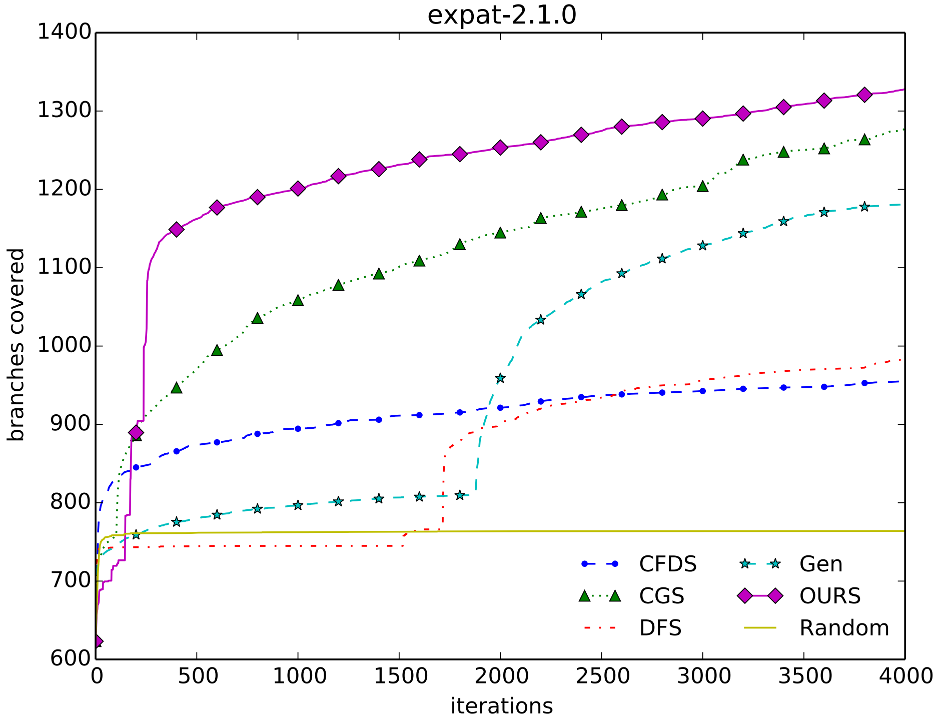
Limitations of Existing Search Heuristics

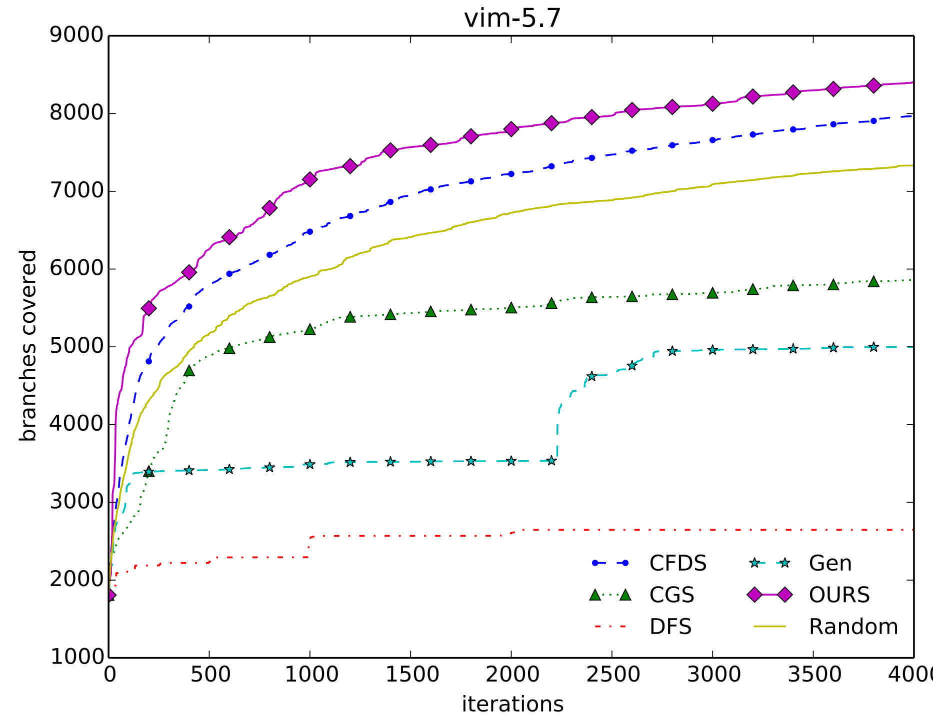
- Furthermore, manually developing a search heuristic is nontrivial, requiring a huge amount of engineering effort and expertise.
- Ordinary developers and testers cannot fully benefit from concolic testing technology.

Our goal: automatically generating search heuristics

Effectiveness

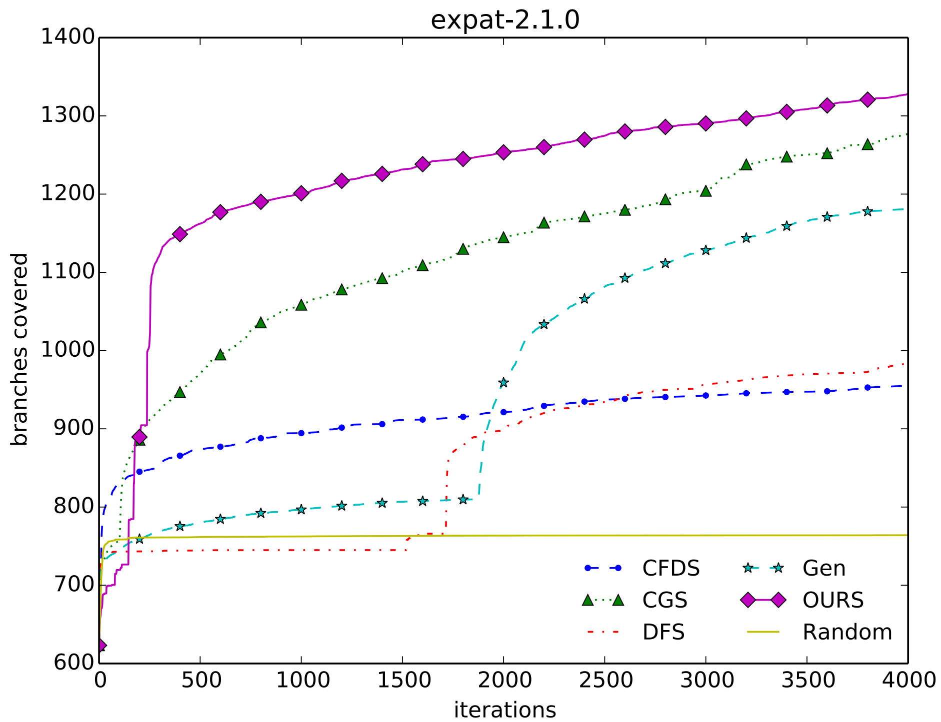
- Considerable increase in branch coverage

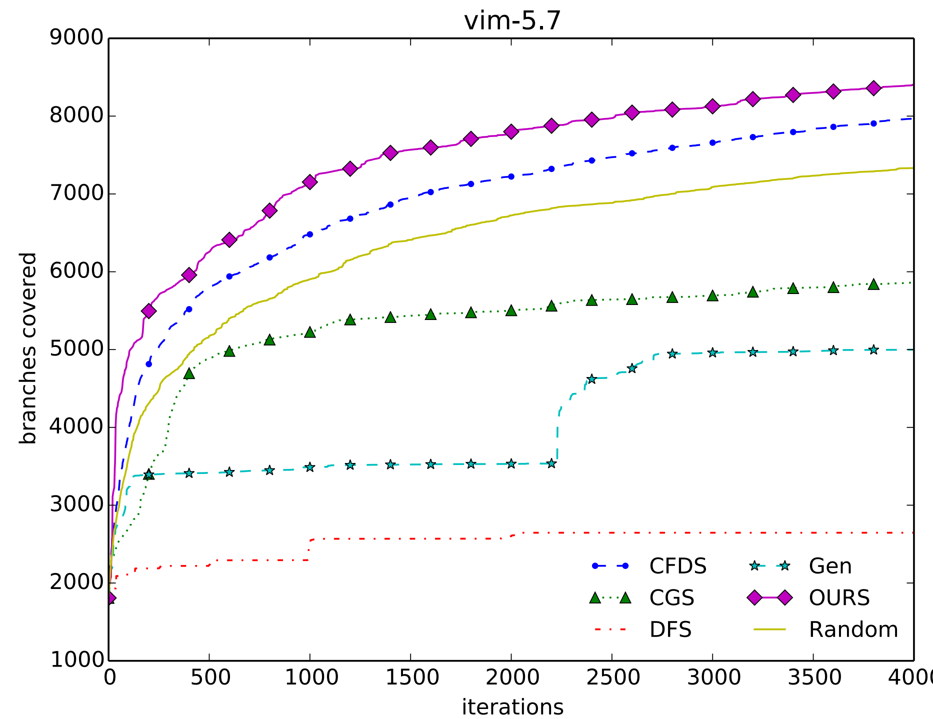




Effectiveness

- Considerable increase in branch coverage





- Dramatic increase in bug-finding capability

	OURS	CFDS	CGS	Random	Gen	DFS
`gawk-3.0.3`	**100/100**	0/100	0/100	0/100	0/100	0/100
`grep-2.2`	**85/100**	0/100	7/100	0/100	0/100	0/100

Key Ideas

- Parameterization of search heuristics
- Searching good parameters for

Parameterization?

- Fixed (non-parameterized) heuristics define single instances of search heuristics:

$$\text{Choose} \in \mathit{SearchHeuristic}$$

ex) DFS

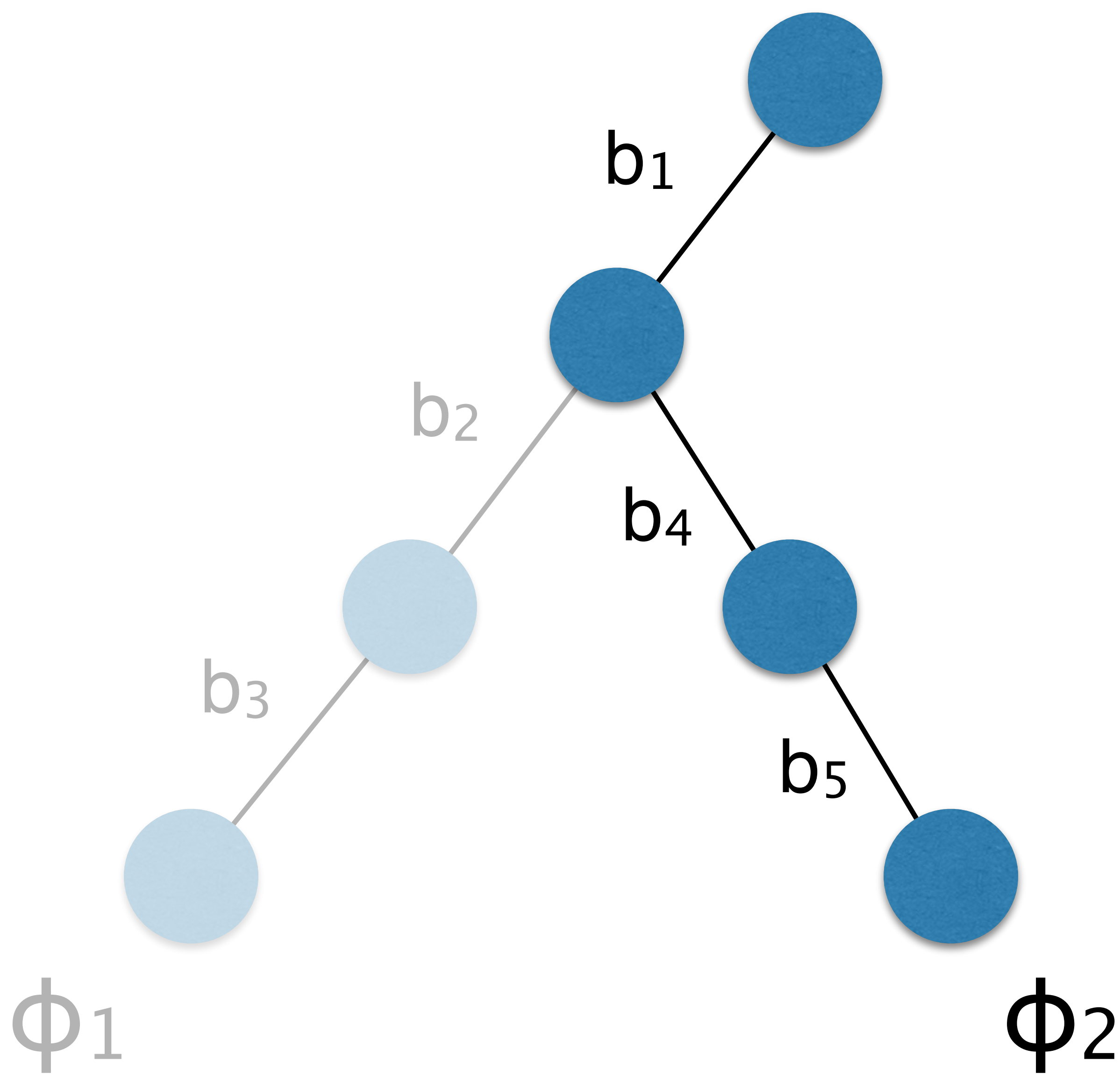
$$\text{Choose}(\langle \Phi_1 \Phi_2 \cdots \Phi_m \rangle) = (\Phi_m, \phi_{|\Phi_m|})$$

- Parameterized heuristic defines a class of search heuristics:

$$\text{Choose}_\theta \subseteq \mathit{SearchHeuristic}$$

Our Parameterized Search Heuristic

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



$score_{\theta}(b_1) = 1.3$

$score_{\theta}(b_4) = 0.0$

$score_{\theta}(b_5) = 0.7$

(1) Feature Extraction

- A feature is a predicate on branches:

$$\pi_i : \mathit{Branch} \to \{0, 1\}$$

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

- Represent a branch by a feature vector

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

- Example

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

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

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

Branch Features

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

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

(2) Scoring

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

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

- Compute score by linear combination of feature vector and parameter

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

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

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

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

Optimization Algorithm

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

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

where

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

Naive Algorithm

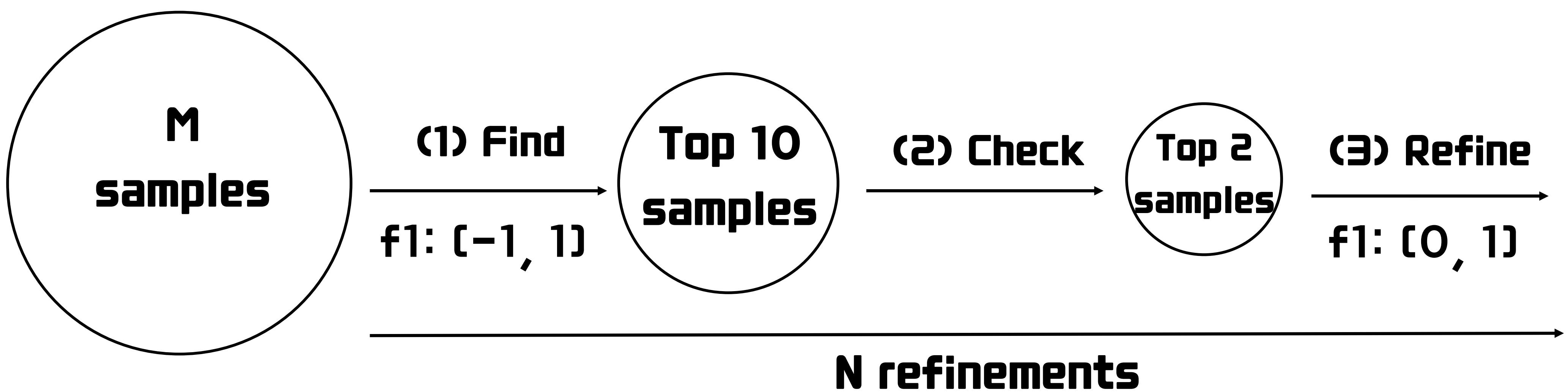
- Naive algorithm based on random sampling

repeat
  $\theta \leftarrow$ sample from $\mathbb{R}^k$
  $B \leftarrow C(P, \mathrm{Choose}_\theta)$
until timeout
return best θ found

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

Our Algorithm

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



Experiments

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

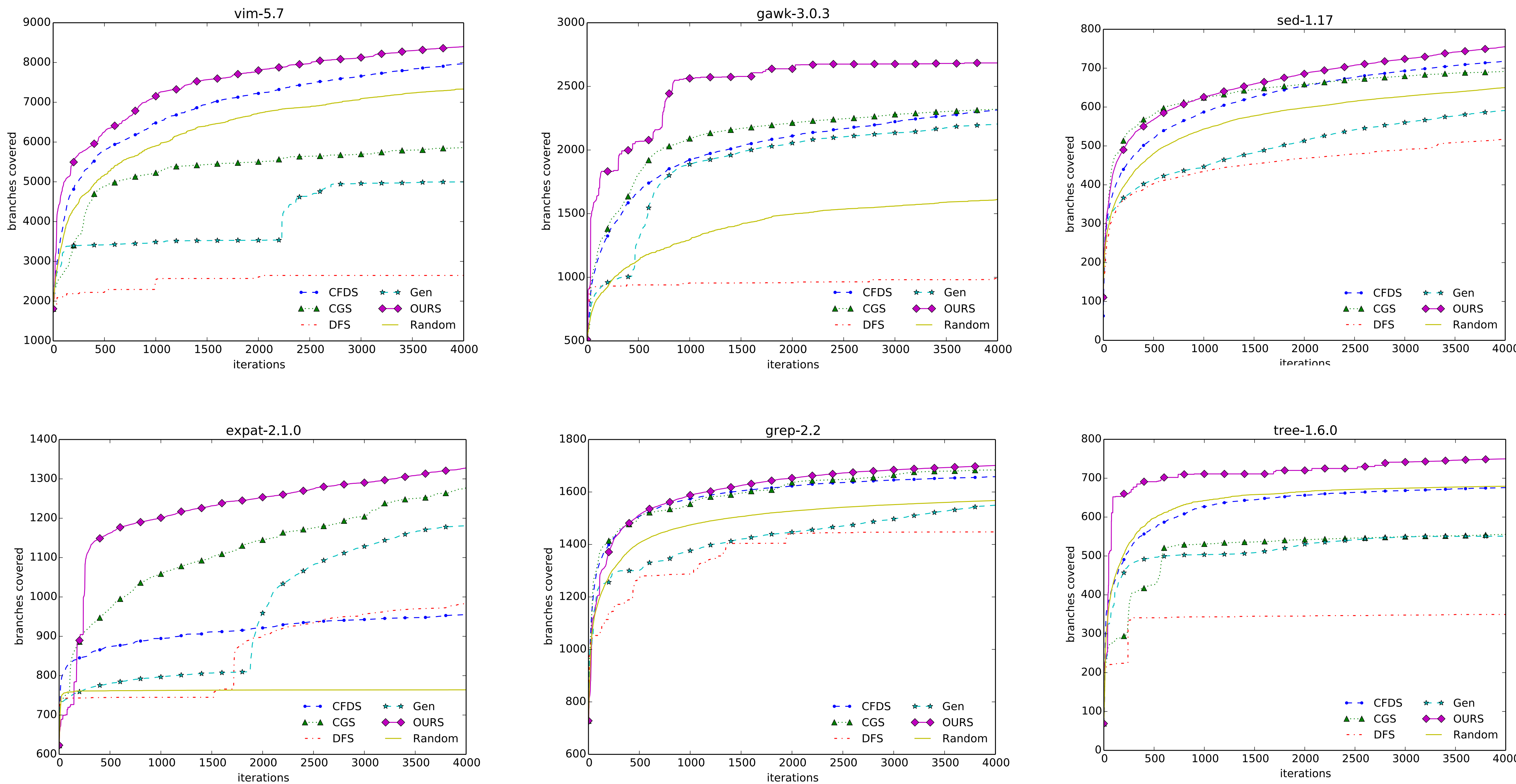
Program	# Total branches	LOC
vim-5.7	35,464	165K
gawk-3.0.3	8,038	30K
expat-2.1.0	8,500	49K
grep-2.2	3,836	15K
sed-1.17	2,565	9K
tree-1.6.0	1,438	4K
cdaudio	358	3K
floppy	268	2K
kbfiltr	204	1K
replace	196	0.5K

Evaluation Setting

- The same initial inputs
- The same testing budget (4,000 executions)
- Performance averaged over 100 trials (20 for vim)

Effectiveness

- Average branch coverage (on large programs)



Effectiveness

- Maximum branch coverage

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

- On small benchmarks

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

Effectiveness

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

	OURS	CFDS	CGS	Random	Gen	DFS
gawk-3.0.3	**100/100**	0/100	0/100	0/100	0/100	0/100
grep-2.2	**85/100**	0/100	7/100	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

- Concolic testing is run in the training phase

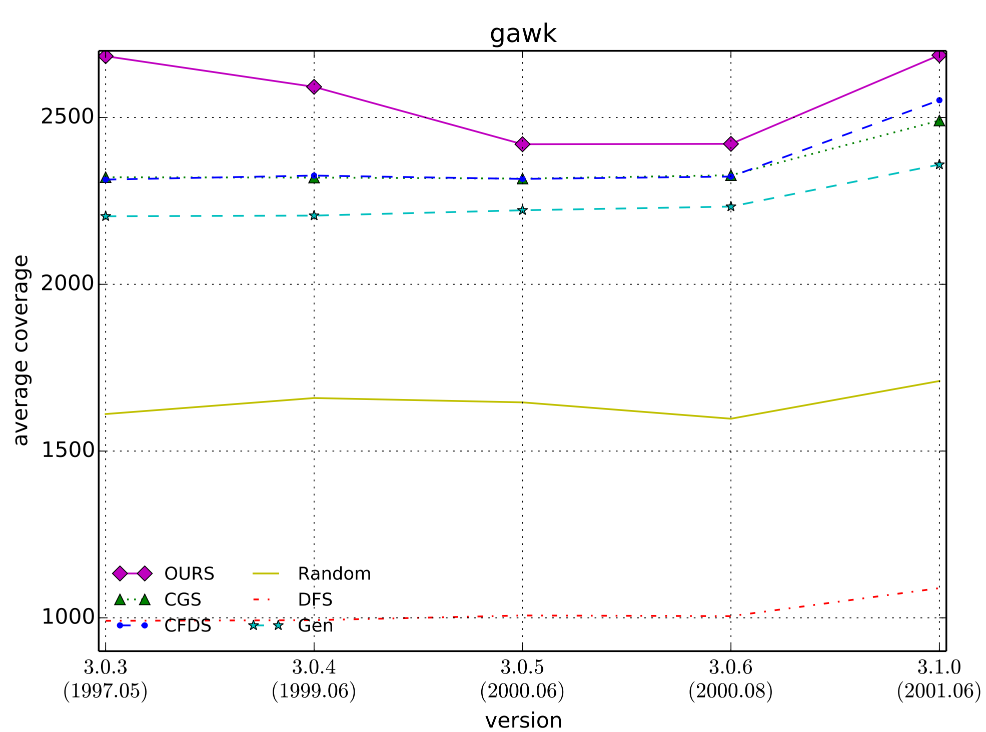
	OURS	CFDS	CGS	Random	Gen	DFS
vim	**14,003**	13,706	7,934	13,835	7,290	7,934
expat	**2,455**	2,339	2,157	1,325	2,116	2,036
gawk	**3,473**	3,382	3,261	3,367	3,302	1,905
grep	**2,167**	2,024	2,016	2,066	1,965	1,478
sed	1,019	1,041	**1,042**	1,007	979	937
tree	**808**	800	737	796	730	665

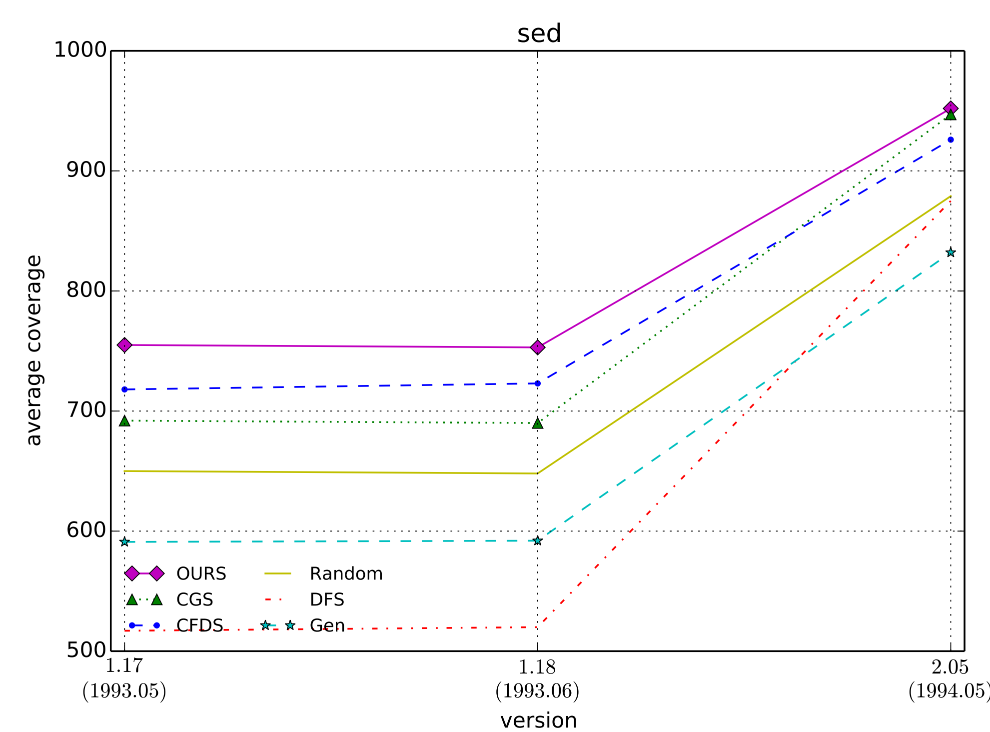
Still useful

- 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



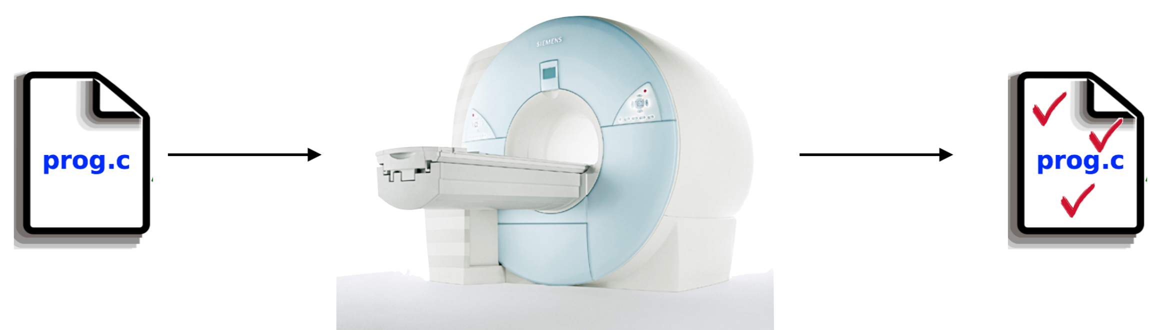


Other On-Going Projects

(program analysis, synthesis, and repair)

Static Program Analysis

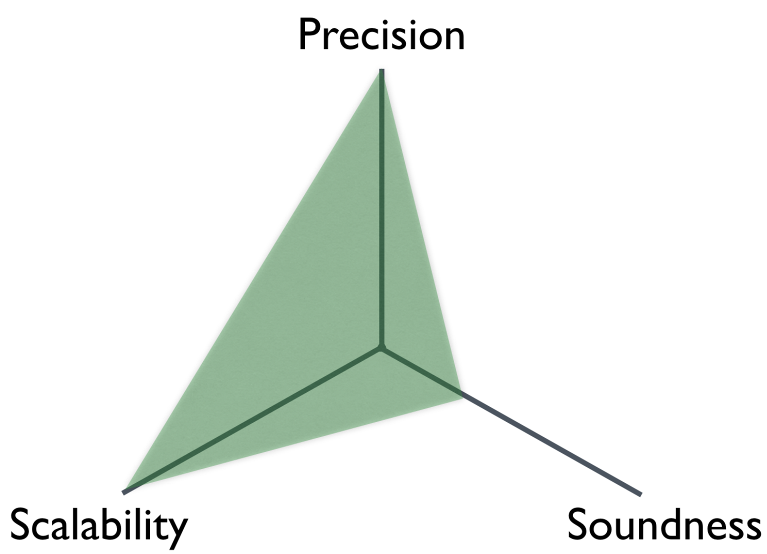
Technology for "Software MRI"

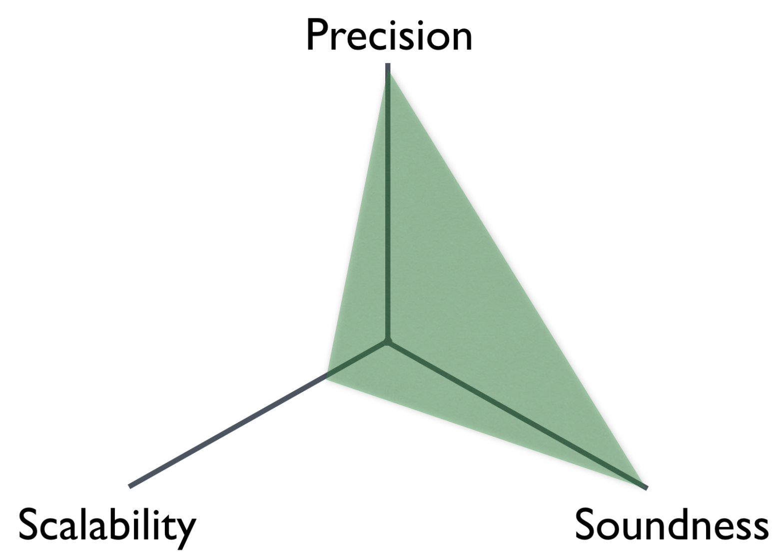


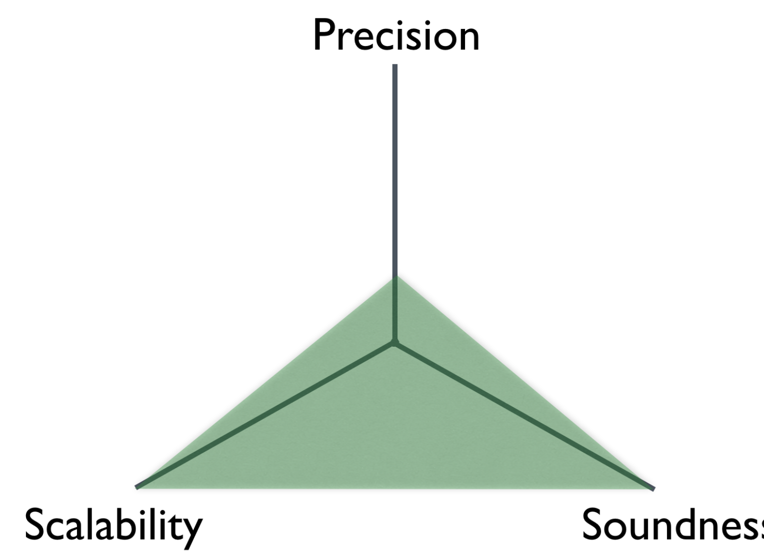
- Detect software defects statically and automatically
- Being widely used in sw industry

Towards More Sound, Precise, and Scalable Static Analysis

- No existing technologies achieve the three

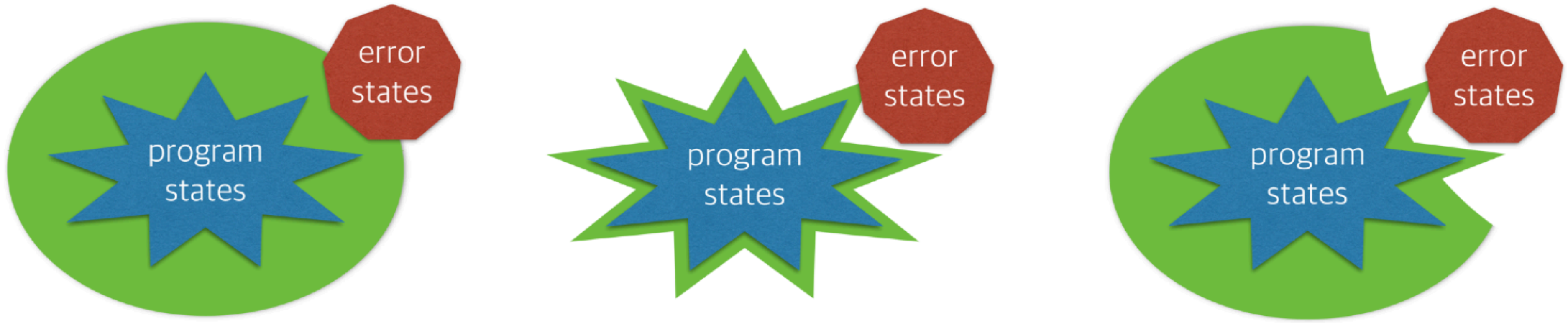






Towards More Sound, Precise, and Scalable Static Analysis

- Our direction: selective program analysis



cheap but imprecise   precise but expensive   cheap yet precise

ICSE'17

Selectively Unsound Analysis

- Selectively apply unsoundness only when harmless

			BASELINE		SELECTIVE		UNIFORM	
Program	**LOC**	**Bug**	**T**	**F**	**T**	**F**	**T**	**F**
mp3rename-0.6	0.6K	1	1	0	1	0	1	0
ghostscript-8.71	1.5K	2	2	0	2	0	2	0
uni2ascii-4.14	5.7K	7	7	0	7	0	7	0
pal-0.4.3	7.4K	3	3	0	0	0	0	0
shntool-3.0.1	16.3K	1	1	10	1	1	1	0
sdop-0.61	23.9K	65	65	78	65	0	0	0
latex2rtf-2.3.8	28.7K	2	2	9	2	8	0	1
rrdtool-1.4.8	34.8K	1	1	12	1	1	1	0
daemon-0.6.4	58.4K	1	1	7	1	1	1	0
rplay-3.3.2	61.0K	3	3	7	2	4	1	2
urjtag-0.10	64.2K	12	12	78	6	0	0	0
a2ps-4.14	64.6K	6	6	26	3	12	1	0
dico-2.0	84.3K	2	2	46	1	1	1	2
Total		106	106	273	92	28	16	5

Our Approaches to Selective Program Analysis

- Pre-analysis approach
  - Selective context-sensitivity [PLDI'14]
- Data-driven approach
  - Selective flow-sensitivity [OOPSLA'15]
  - Selective relational analysis [SAS'16]
  - Selective unsoundness [ICSE'17]
  - Disjunctive model and algorithm [OOPSLA'17a]
  - Automatic feature construction [OOPSLA'17b]

Program Analysis vs. Synthesis

- Program Analysis derives specifications from code
- Program Synthesis derives code from specifications

```
int f(int n) {
  int i = 0;
  int r = 1;
  while (i < n)
  {
    r = r * i;
    i = i + 1;
  }
  return r;
}
```

program analysis →

← program synthesis

$f(1) = 1$

$f(2) = 2$

$f(3) = 6$

…

$f(n) = n!$

Program Synthesis

- Generate program code from specifications automatically
  - specification: logics, examples, implementation, etc
- Applications
  - programming assistance: e.g., complete tricky parts of programs
  - end-user programming: e.g., automate repetitive tasks
  - algorithm discovery: find a new solution for a problem
  - automatic patch generation: automatically fix software bugs

Example

- Specification is given as test cases

$$\text{reverse}(12) = 21, \; \text{reverse}(123) = 321$$

```
reverse(n) {
  r := 0;
  while (          ) {

  };
  return r;
}
```

2.5s

```
reverse(n) {
  r := 0;
  while ( n > 0 ) {
    x := n % 10;
    r := r * 10;
    r := r + x;
    n := n / 10;
  };
  return r;
}
```

Status

- Better than humans for introductory programming tasks

Domain	No	Description	Vars		Ints	Exs	Time (sec)		
			IVars	AVars			Base	Base+Opt	Ours
Integer	1	Given n, return $n!$.	2	0	2	4	0.0	0.0	0.0
	2	Given n, return $n!!$ (i.e., double factorial).	3	0	3	4	0.0	0.0	0.0
	3	Given n, return $\sum_{i=1}^{n} i$.	3	0	2	4	0.1	0.0	0.0
	4	Given n, return $\sum_{i=1}^{n} i^2$.	4	0	2	3	122.4	18.1	0.3
	5	Given n, return $\prod_{i=1}^{n} i^2$.	4	0	2	3	102.9	13.6	0.2
	6	Given a and n, return a^n.	4	0	2	4	0.7	0.1	0.1
	7	Given n and m, return $\sum_{i=n}^{m} i$.	3	0	2	3	0.2	0.0	0.0
	8	Given n and m, return $\prod_{i=n}^{m} i$.	3	0	2	3	0.2	0.0	0.1
	9	Count the number of digit for an integer.	3	0	3	3	0.0	0.0	0.0
	10	Sum the digits of an integer.	3	0	3	4	5.2	2.2	1.3
	11	Calculate product of digits of an intger.	3	0	3	3	0.7	2.3	0.3
	12	Count the number of binary digit of an integer.	2	0	3	3	0.0	0.0	0.0
	13	Find the nth Fibonacci number.	3	0	3	4	98.7	13.9	2.6
	14	Given n, return $\sum_{i=1}^{n}(\sum_{m=1}^{i} m))$.	3	0	2	4	$\perp$	324.9	37.6
	15	Given n, return $\prod_{i=1}^{n}(\prod_{m=1}^{i} m))$.	3	0	2	4	$\perp$	316.6	86.9
	16	Reverse a given integer.	3	0	3	3	$\perp$	367.3	2.5
Array	17	Find the sum of all elements of an array.	3	1	2	2	8.1	3.6	0.9
	18	Find the product of all elements of an array.	3	1	2	2	7.6	3.9	0.9
	19	Sum two arrays of same length into one array.	3	2	2	2	44.6	29.9	0.2
	20	Multiply two arrays of same length into one array.	3	2	2	2	47.4	26.4	0.3
	21	Cube each element of an array.	3	1	1	2	1283.3	716.1	13.0
	22	Manipulate each element into 4th power.	3	1	1	2	1265.8	715.5	13.0
	23	Find a maximum element.	3	1	2	2	0.9	0.7	0.4
	24	Find a minimum element.	3	1	2	2	0.8	0.3	0.1
	25	Add 1 to each element.	2	1	1	3	0.3	0.0	0.0
	26	Find the sum of square of each element.	3	1	2	2	2700.0	186.2	11.5
	27	Find the multiplication of square of each element.	3	1	1	2	1709.8	1040.3	12.6
	28	Sum the products of matching elements of two arrays.	3	2	1	3	20.5	38.7	1.5
	29	Sum the absolute values of each element.	2	1	1	2	45.0	50.5	12.1
	30	Count the number of each element.	3	1	3	2	238.9	1094.1	0.2
	Average						> 616.8	165.5	6.6

Automatic Program Repair

- Memory management errors (memory leak, use-after-free, double free) are common:

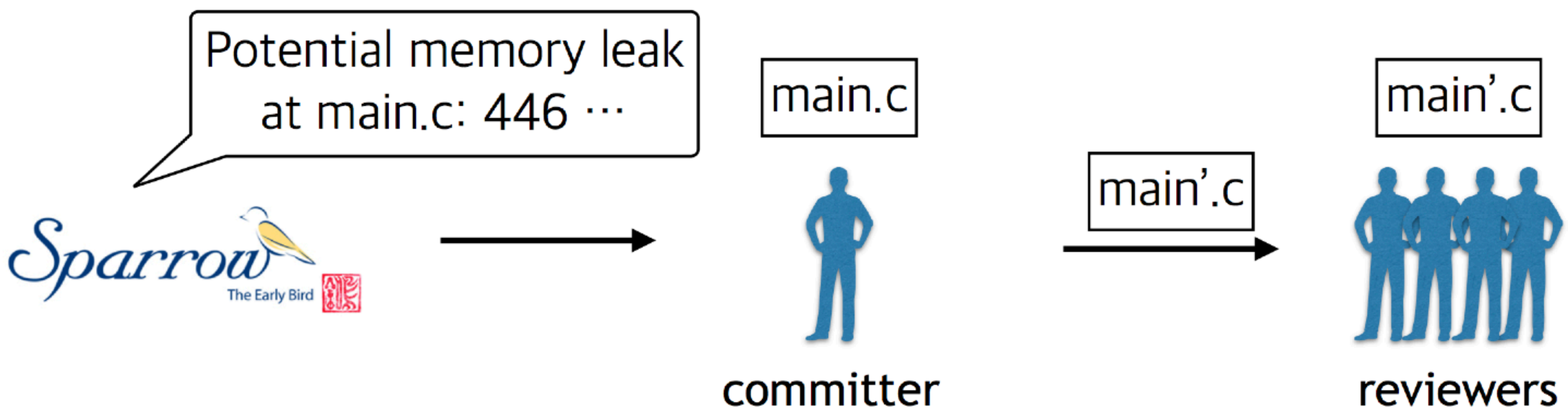
<Statistics of common bug types>

	StackOverflow	GitHub	Tizen CodeReview
BO	10,099	586,385	155
IO	11,356	594,032	15
ND	3,592	755,354	220
MM	107,869	3,804,869	1,320

*BO/IO: Buffer/Integer-overflow
*ND: Null-dereference
*MM: Memory management errors

Automatic Program Repair

- Manual repair is error-prone, requiring multiple iterations of review process



<패치 과정>

Automatic Program Repair

- Automatically patching memory management errors
- Patched programs are guaranteed to be error-free

```
p = malloc(); // o1

if(...){
    q = malloc(); // o2
    *q = *p;
} else {
    q = p;
}
// Use q
free(p);

// o2 leaks here
return;
```

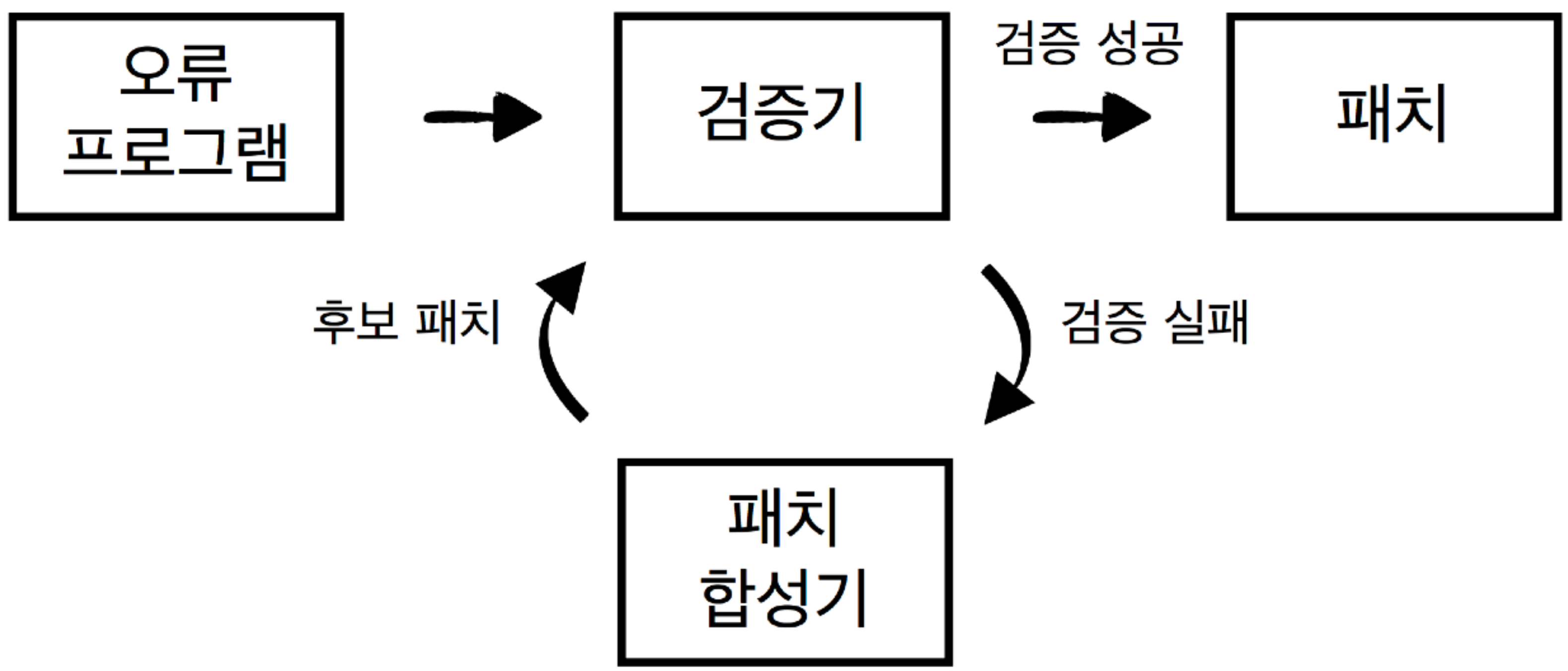
<원본 코드>

```
p = malloc(); // o1

if(...){
    q = malloc(); // o2
    *q = *p;
    free(p);
} else {
    q = p;
}
// Use q
free(q);

return;
```

<올바른 패치>

Automatic Program Repair

- Combination of program analysis and synthesis



Automatic Feedback Generation

- In typical programming courses (e.g., MOOC):
  - students receive no personalized feedback
  - solutions are not much helpful

오답 코드

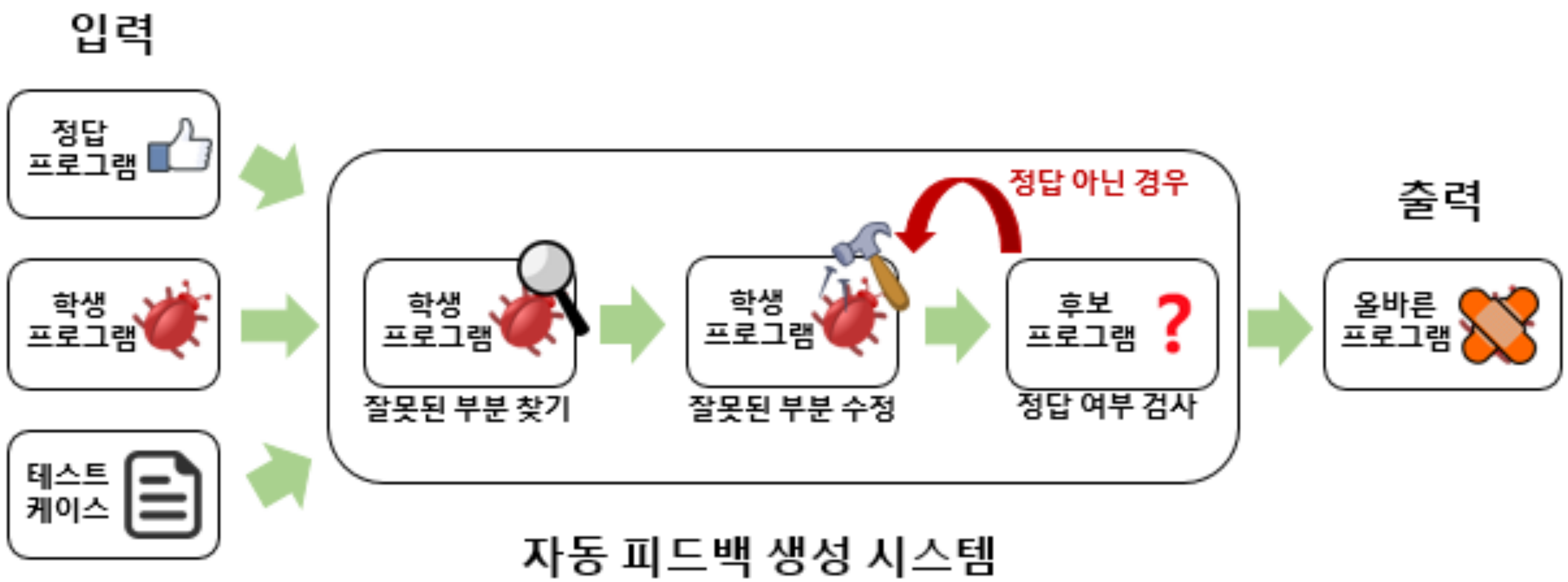
```
...
| Sum plus ->
 (match plus with
  []     -> Const 0
  | [hd]   -> diff( hd, var)
  | hd::tl -> Sum [diff(hd, var); diff(Times tl, var)]
 ) ...
```

Sum

모범 답안

```
let rec map f (l,var) =
   match l with
   | [] -> []
   | hd::tl -> (f (hd,var))::(map f (tl,var))
...
| Sum lst -> Sum (map diff (lst,var))
...
```

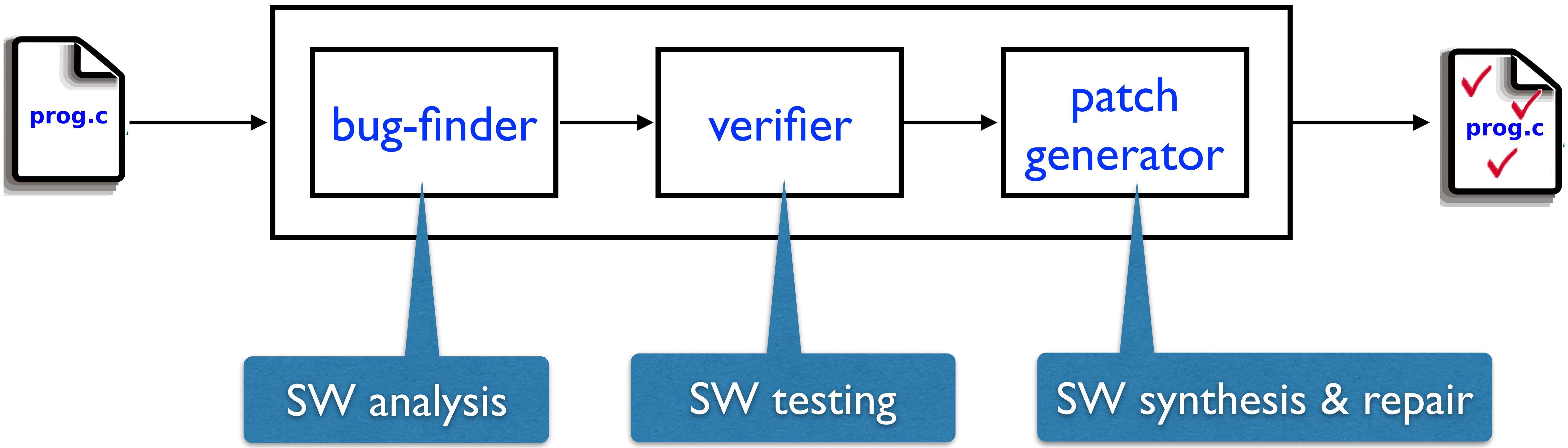
Automatic Feedback Generation



- Error localization by MAX-SAT solving
- Correction by program synthesis

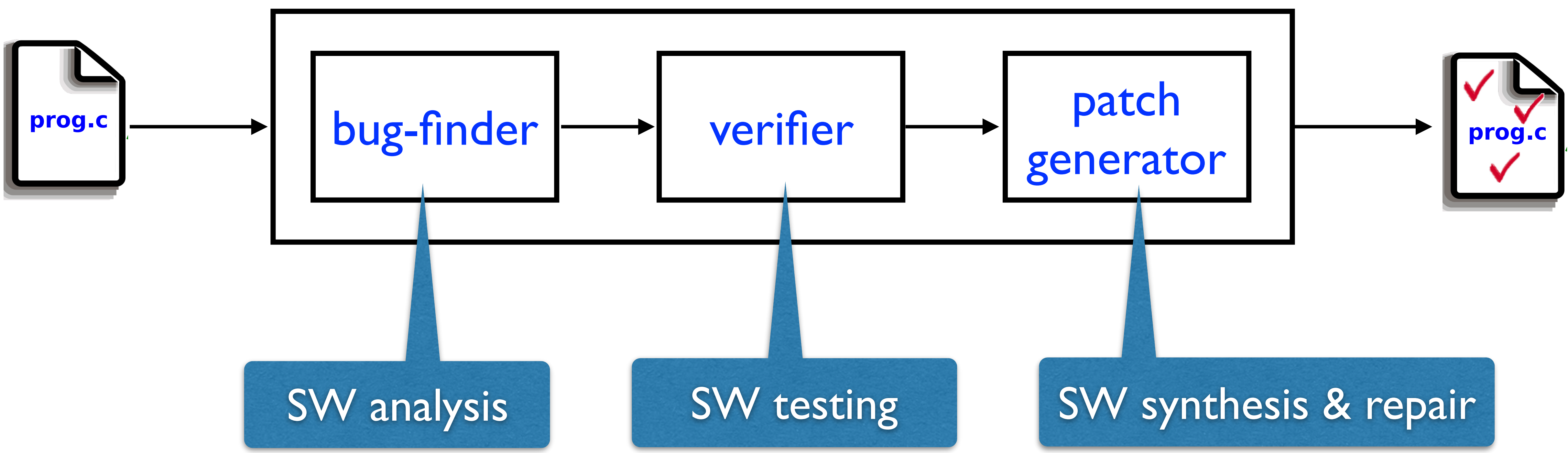
Summary

- Technologies for automatically finding, verifying, and fixing software errors and vulnerabilities



Summary

- Technologies for automatically finding, verifying, and fixing software errors and vulnerabilities



Thank you