머신러닝을 통한 소프트웨어 테스팅의 탐색전략 자동생성 기술

(Automatically Generating Search Heuristics of Software Testing with Machine Learning)

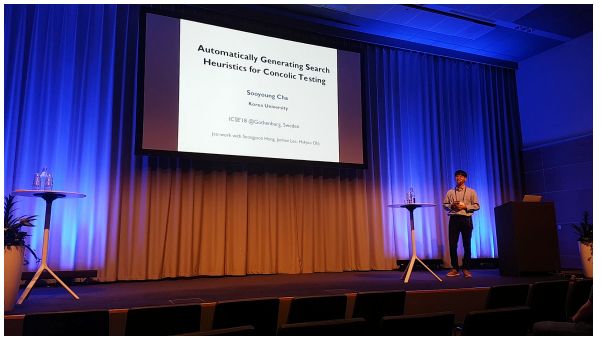
Sooyoung Cha

Korea University

18 October 2018 @Samsung R&D Campus

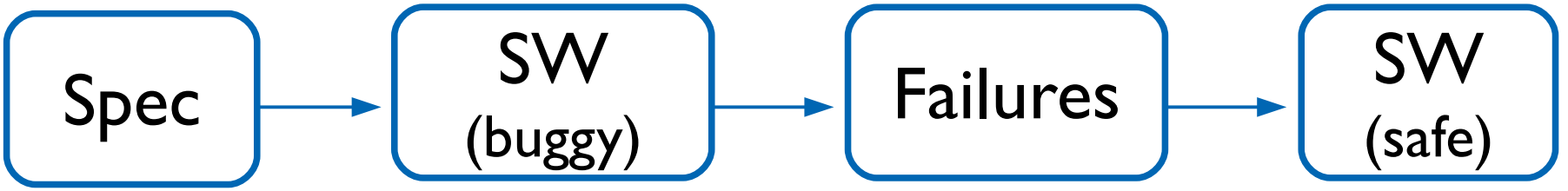
I am Sooyoung Cha

- Research Areas: software engineering.
  - Data-driven software testing.
  - Publications: top-venues in software engineering.
    - ICSE'18, ASE'18, ~~FSE'19?~~
- Enjoy presenting a talk.
  - Apr. 2018. Oxford Univ @UK
  - Jun. 2018. ICSE'18 @Sweden
  - Sep. 2018. ASE'18 @France



Software Analysis Lab (SAL)

- Goal: Automated programming / testing / debugging



programming (manual) → Program Synthesis

PAT(IJCAI'18)
FixML(OOPSLA'18)

testing (manual) → Program Analysis

ParaDySE (ICSE'18)
ConTest (ASE'18)

debugging (manual) → Program Repair

MemFix (FSE'18)

http://prl.korea.ac.kr

Today's Talk

- Concolic Testing (Dynamic Symbolic Execution)
  - An effective software testing method.
  - **Conc**rete and Symb**olic** executions.

- Open-Source Tool:
  - A state-of-the-art technique.
  - Parametric Dynamic Symbolic Execution.

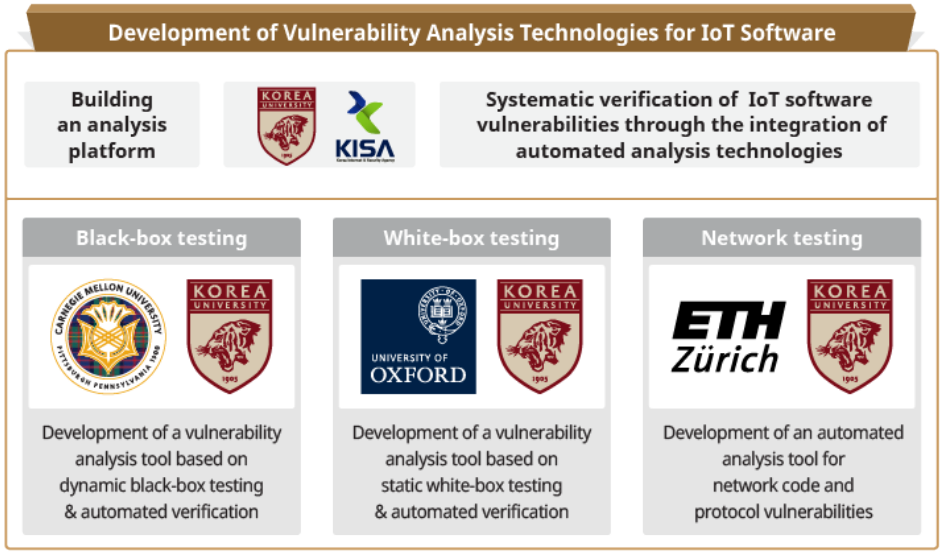
Today's Talk

- Concolic Testing (Dynamic Symbolic Execution)
  - An effective software testing method.
  - **Conc**rete and Symb**olic** executions.

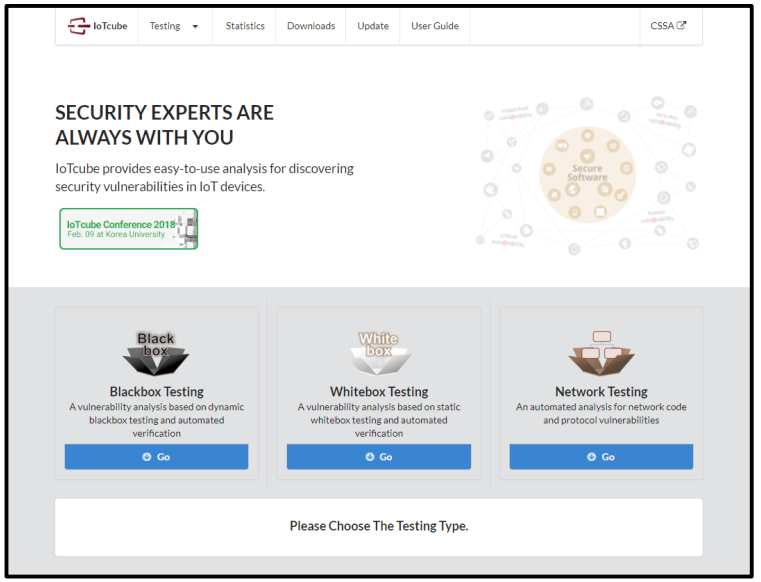
- Open-Source Tool: ParaDySE
  - A state-of-the-art technique.
  - Parametric Dynamic Symbolic Execution.
    (https://github.com/kupl/ParaDySE)

IoTcube

- Web service implementation (Users: 10,851)
  - Discovered security vulnerabilities in IoT devices.
  - Implemented our technique on ctest in IoTcube.



Center for Software Security and Assurance (CSSA)



(https://iotcube.korea.ac.kr)

Concolic Testing

Software Testing

- Motivation: Unsafe Software
  - Software failures everywhere

[1996] The Arian-5 rocket failure.
Cost: 10 years and $8 billion.

[2010] Accident in radiation therapy system.
Cost: 21 patients died.

[2014] HeartBleed.
Cost: 600,000 vulnerable servers.

Concolic Testing

- An effective software testing method.
  - Combine concrete and symbolic executions.
  - Enhance random testing.

- Goal
  - Improve code coverage in a limited budget.
  - Find software vulnerability. 
    - SAGE: Find 30% of all Windows 7 security bugs.

Microsoft SAGE

S²E

TRITON

Dynamic Binary Analysis

Limitation of Random Testing

```
int twice (int v) {
    return 2 × v;
}

void main (int x, int y) {

    z  = twice (y);

    if ( z == x ) {

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

- Probability of the error?

  (1≤ x,y ≤100)

Limitation of Random Testing

```
int twice (int v) {
    return 2 × v;
}

void main (int x, int y) {

    z  = twice (y);

    if ( z == x ) {

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

- Probability of the error?

  (1≤ x,y ≤100)

  0.4 %

  - Random testing requires 250 runs.
  - Concolic testing finds it in 3 runs.

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {
      z  = twice (y);

      if ( z == x ) {

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

Concrete State

Symbolic State

x = 22, y = 7

x = α, y = β

PathCond (PC): true

Initial Random Input
x=22, y=7

1st iteration

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {

      z  = twice (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

PC: true

1st iteration

Concolic Testing

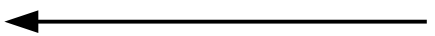
```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {

      z  = twice (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 *PC*: 2*β ≠ α

1st iteration

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {

      z  = twice (y);

      if ( z == x ) {

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

Concrete State	Symbolic State
	Solve: 2*β = α **Solution: α=2, β=1**
x = 22, y = 7, z = 14	x = α, y = β, z = 2*β *PC*: 2*β ≠ α

1st iteration

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {
      z  = twice (y);

      if ( z == x ) {

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

Concrete State

Symbolic State

x = 2, y = 1

x = α, y = β

PC: true

2nd iteration

Concolic Testing

```
int twice (int v) {
    return 2 × v;
}

void main (int x, int y) {

    z  = twice (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 *PC*: true

2nd iteration

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {

      z  = twice (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

PC: 2*β = α

2nd iteration

Concolic Testing

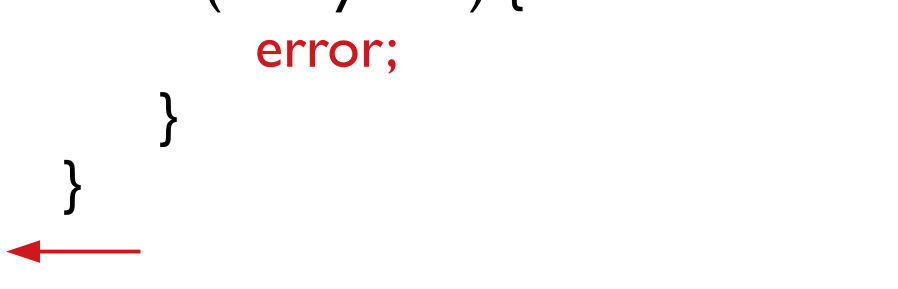
```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {

      z  = twice (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 *PC*: (2*β = α) ∧ (α ≤ β+10)

2nd iteration

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {

      z  = twice (y);

      if ( z == x ) {

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

Concrete State	Symbolic State
	Solve: (2*β = α) ∧ (α > β+10) **Solution: α=22, β=11**
x = 2, y = 1, z=2	x = α, y = β, z = 2*β *PC*: (2*β = α) ∧ **(α ≤ β+10)**

2nd iteration

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {
      z  = twice (y);

      if ( z == x ) {

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

Concrete State	Symbolic State
x = 22, y = 11	x = α, y = β
	PC: true

3rd iteration

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {

      z  = twice (y);
      if ( z == x ) {

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

Concrete State

Symbolic State

x = 22, y = 11, z = 22

x = α, y = β, z =2*β

PC: true

3rd iteration

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {

      z  = twice (y);

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

Concrete State

Symbolic State

x = 22, y = 11, z = 22

x = α, y = β, z =2*β

PC: 2*β = α

3rd iteration

Concolic Testing

```
int twice (int v) {
      return 2 × v;
}

void main (int x, int y) {

      z  = twice (y);

      if ( z == x ) {

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

Concrete State

Symbolic State

error-triggering input

x = 22, y = 11,
z = 22

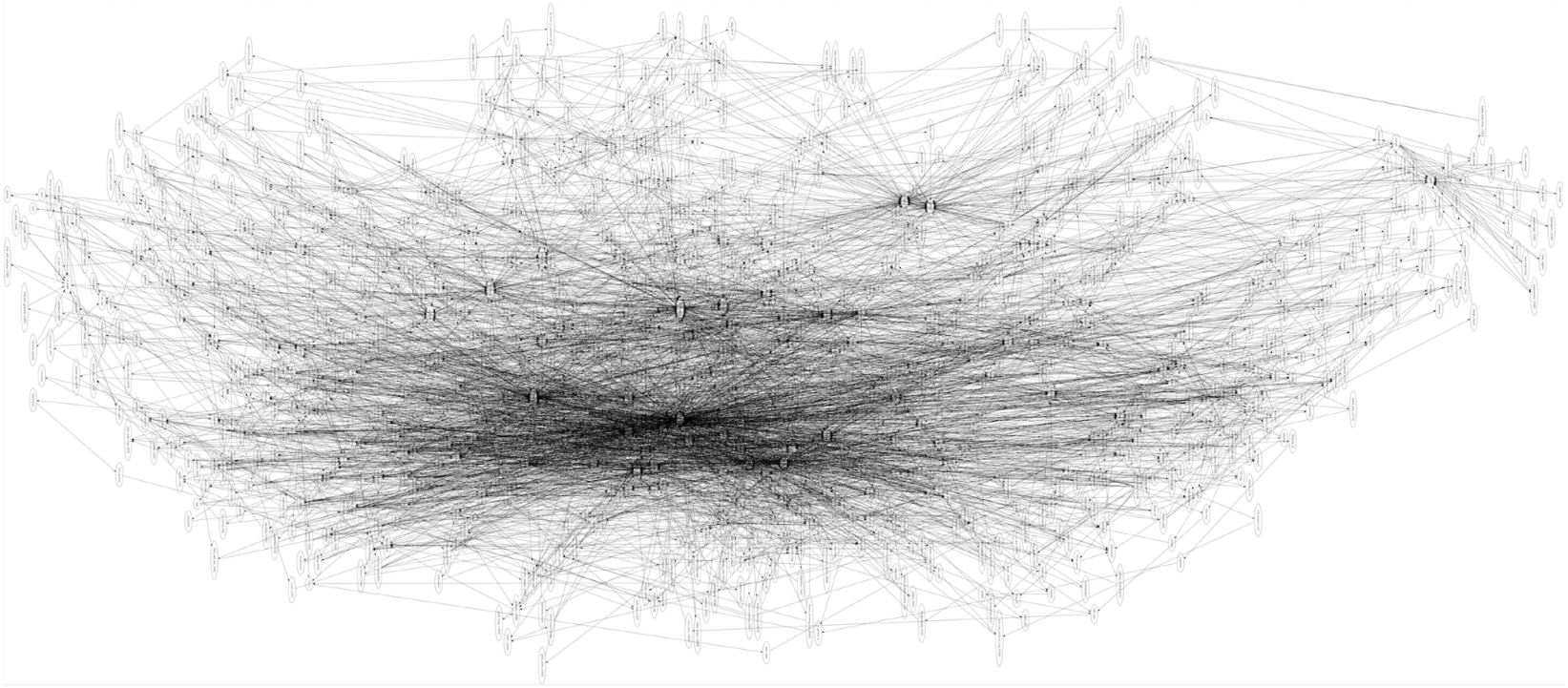
$x = \alpha, y = \beta, z = 2*\beta$

PC: $(2*\beta = \alpha) \wedge (\alpha > \beta+10)$

3rd iteration

Challenge

- Path Explosion
  - # of execution paths: $2^{\text{\# of branches}}$
    - ex) grep-2.2(3,836) : $2^{3,836}$ paths (worst case)
  - Exploring all paths is impossible.

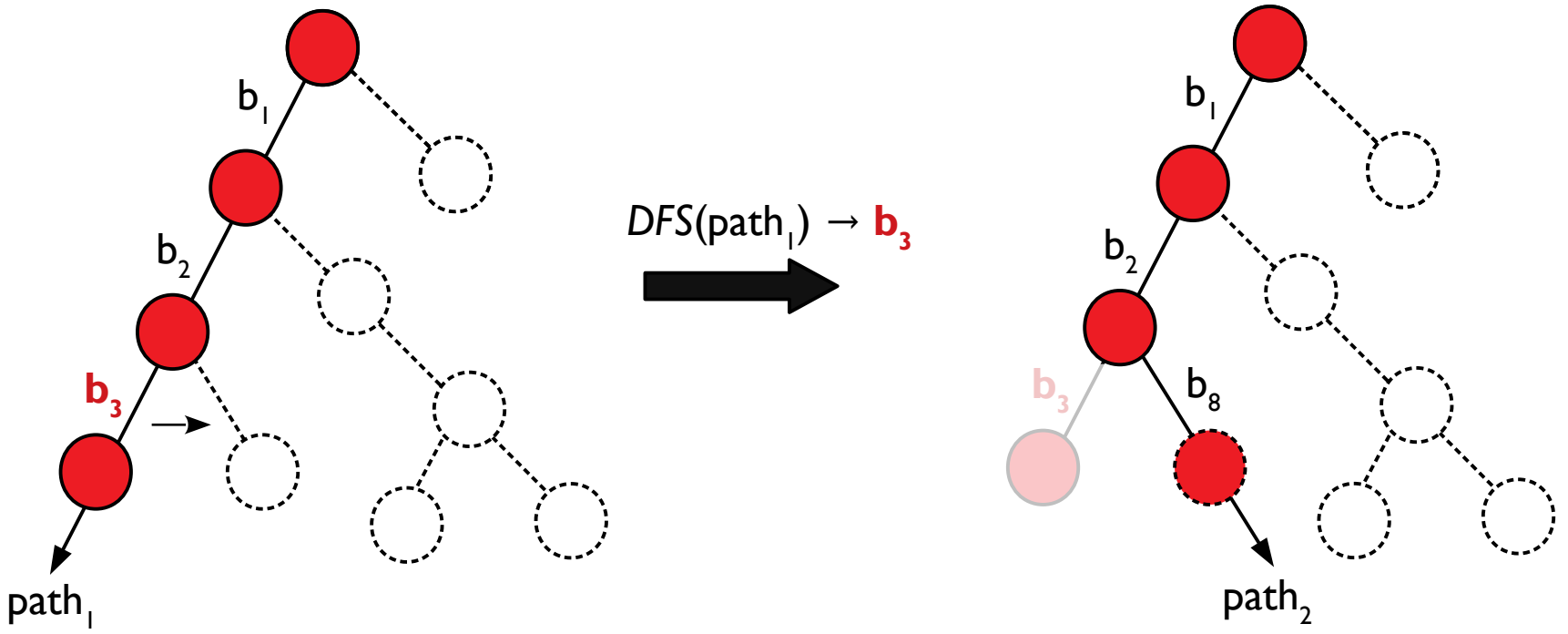


Diverse Solutions

- Goal: Mitigate the path-explosion.

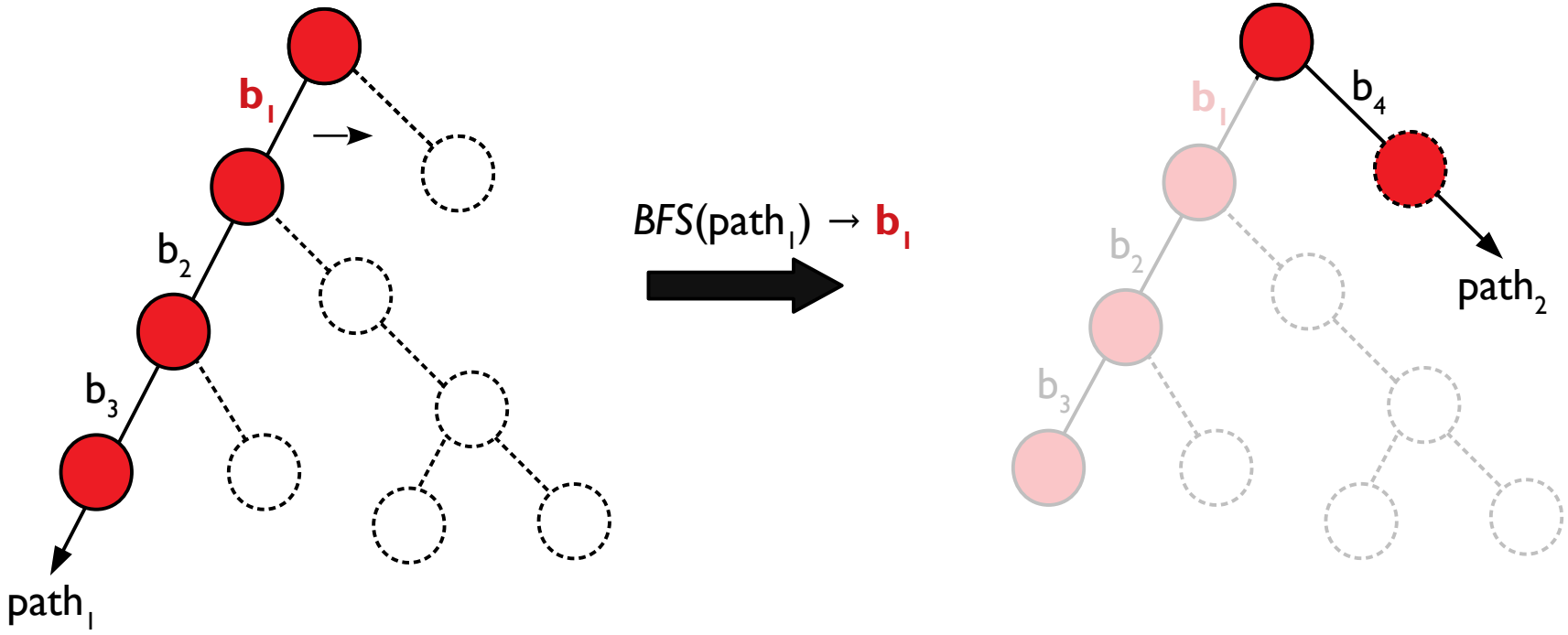
Search Heuristic

- Select branches that are likely to maximize code coverage.
- Numerous search heuristics have been proposed.
  - DFS, BFS, Random, Generational, CFDS, CGS, etc



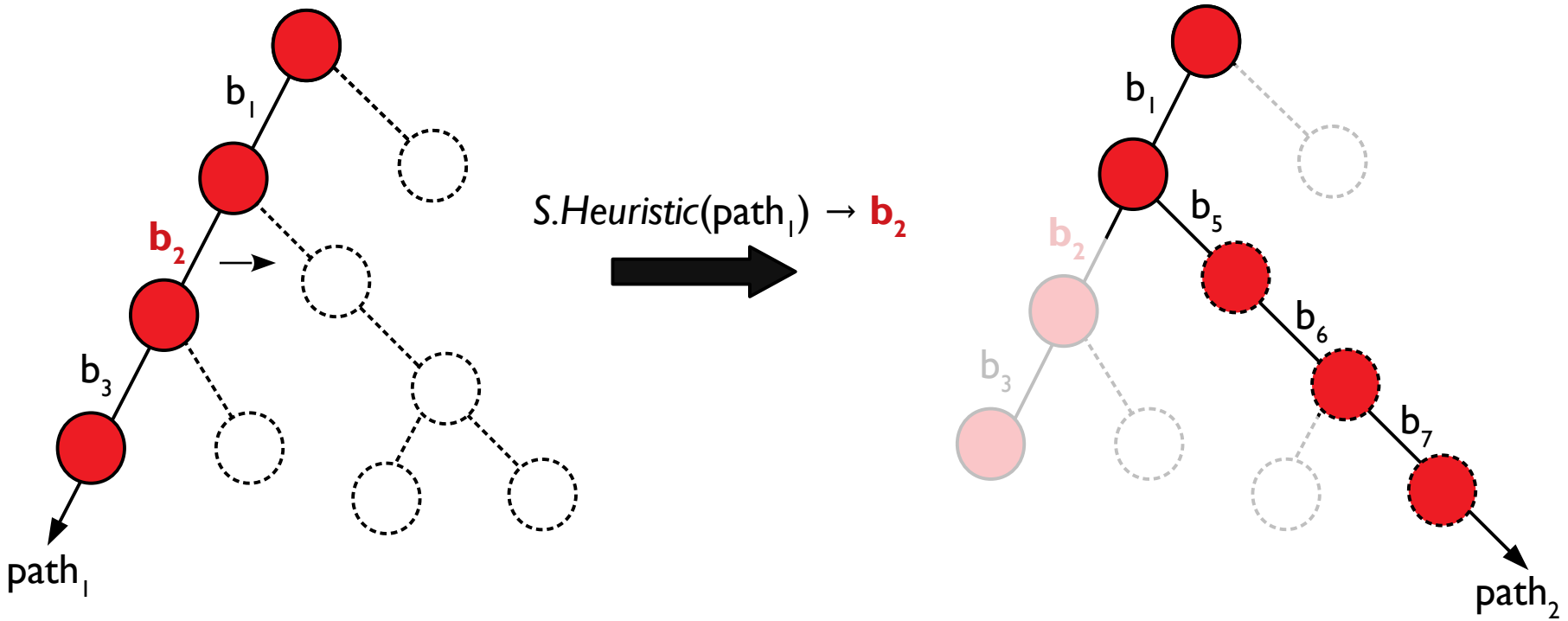
Search Heuristic

- Select branches that are likely to maximize code coverage.
- Numerous search heuristics have been proposed.
  - DFS, BFS, Random, Generational, CFDS, CGS, etc



Search Heuristic

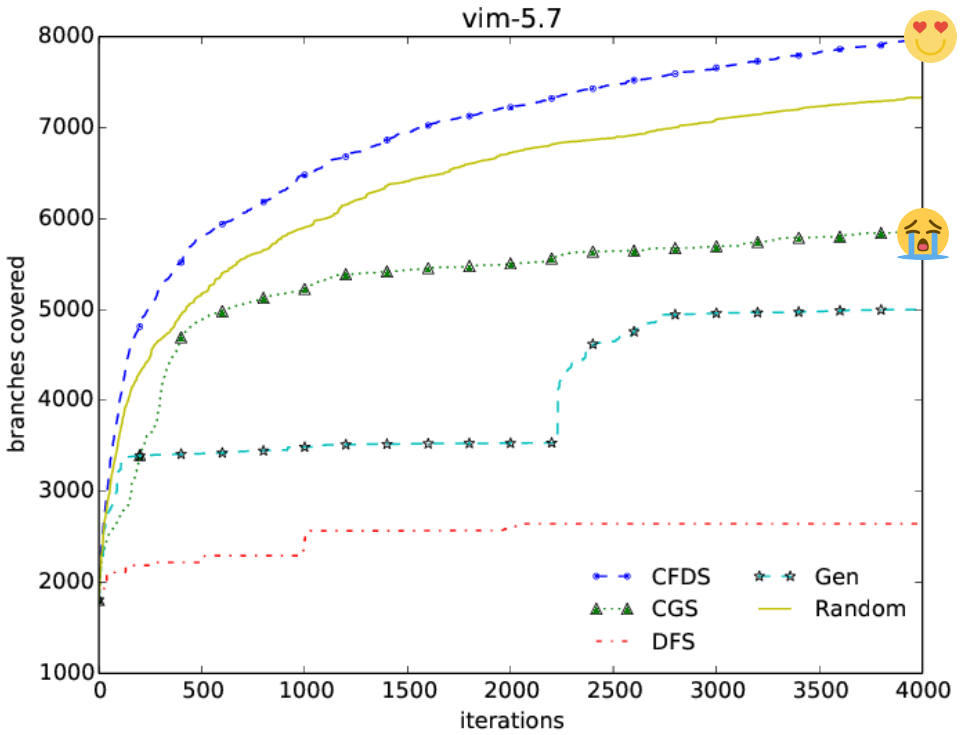
- Select branches that are likely to maximize code coverage.
- Numerous search heuristics have been proposed.
  - DFS, BFS, Random, Generational, CFDS, CGS, etc

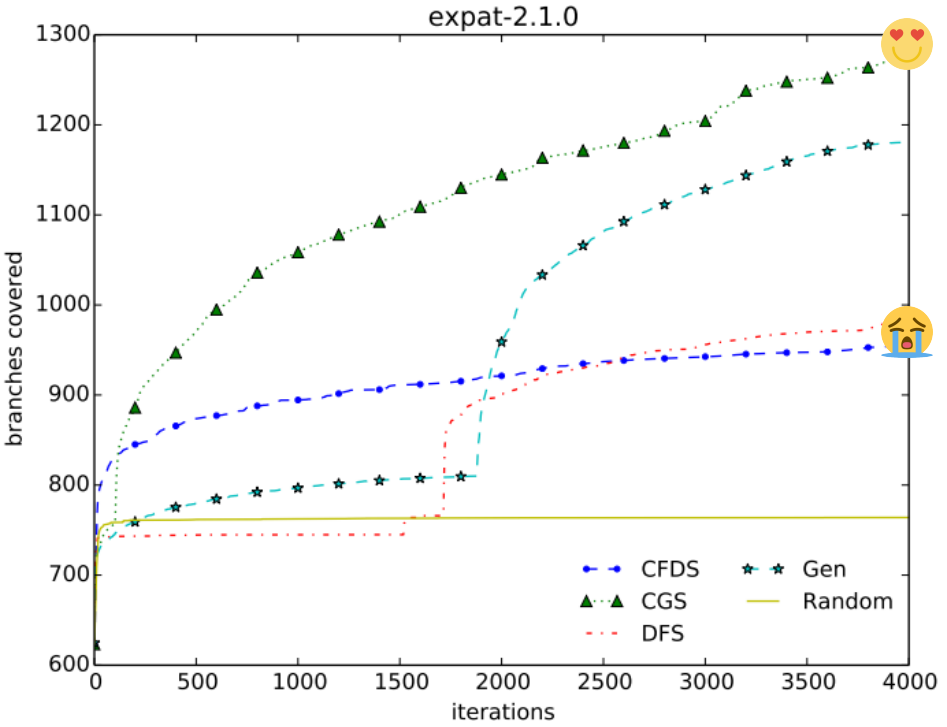


Open-Source Tool: ParaDySE

Motivation

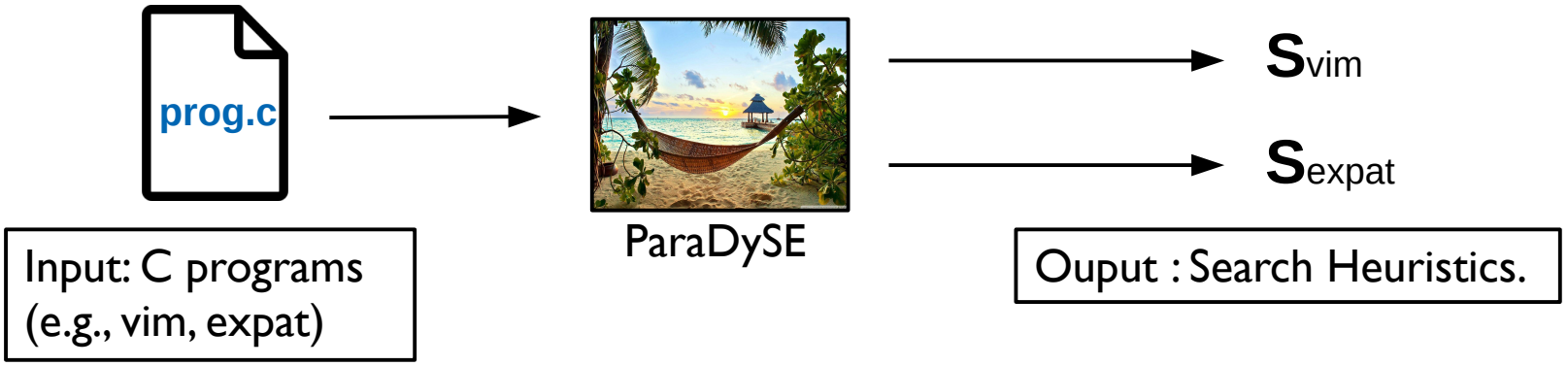
- No existing heuristics consistently achieve high coverage.
- Designing new heuristic is highly nontrivial.
  - Search Heuristic → (ICSE, FSE, ASE, NDSS, ...)





Goal

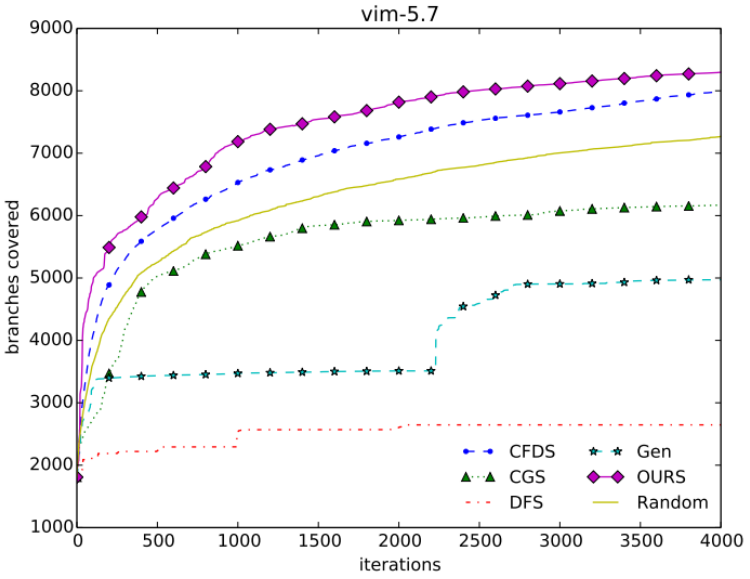
- Automatically Generating Search Heuristics

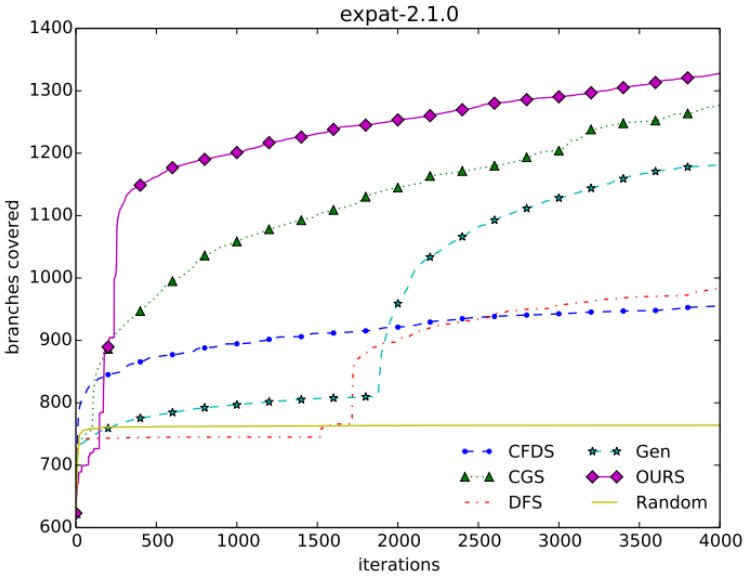


- Key ideas
  - Machine learning + Concolic Testing.
    - Parameterized Search Heuristic.
    - Effective Parameter Search Algorithm.

Effectiveness

- Considerable increase in branch coverage.

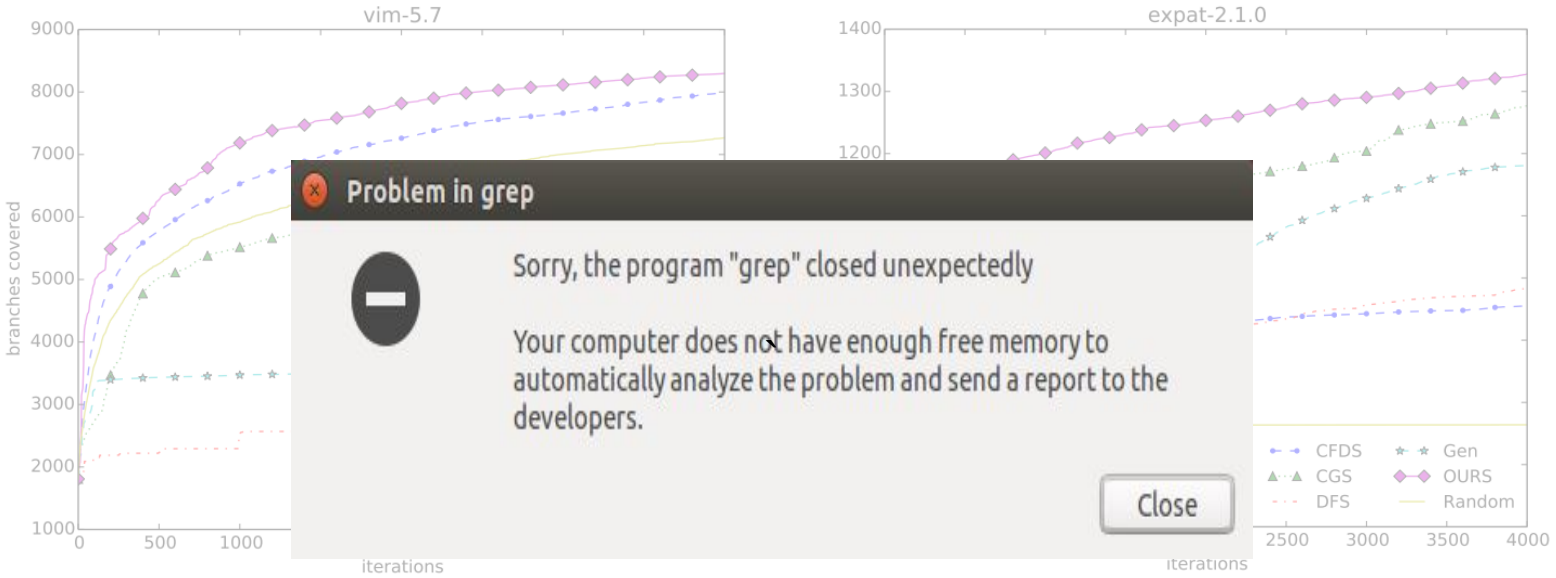




- Found the real-world performance bugs.
  - gawk-3.0.3: ./gawk 'V0\\n^000000000000070000000' file
  - grep-2.2: ./grep '\(\)\1\+**' file
    - Trigger the error in grep-3.1 (the latest version)

Effectiveness

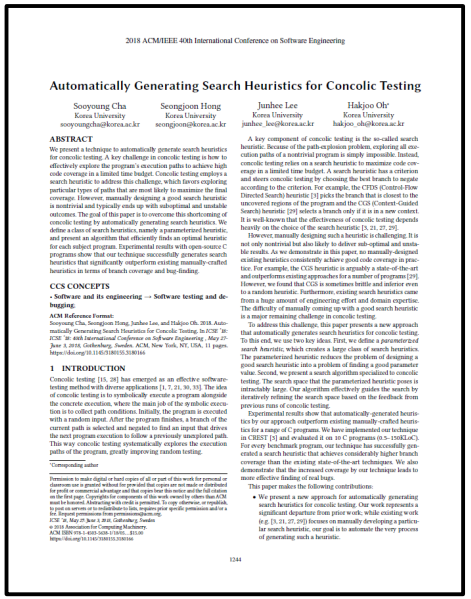
- Considerable increase in branch coverage.



- Found real-world performance bugs.
  - gawk-3.0.3: ./gawk 'V0\\n^000000000000070000000' file
  - grep-2.2: ./grep '\(\)\1\+**' file
    - Trigger the error in grep-3.1 (the latest version)

In academia

- Make the tool publicly available. (Recently)



2018 ACM/IEEE 40th International Conference on Software Engineering

Automatically Generating Search Heuristics for Concolic Testing

Soyoung Cha, Korea University, soyoungcha@korea.ac.kr
Seongjoon Hong, Korea University, seongjoon@korea.ac.kr
Junhee Lee, Korea University, junhee_lee@korea.ac.kr
Hakjoo Oh*, Korea University, hakjoo_oh@korea.ac.kr

ABSTRACT

We present a technique to automatically generate search heuristics for concolic testing. A key challenge in concolic testing is how to effectively explore the program's execution paths to achieve high code coverage in a limited time budget. Concolic testing employs a search heuristic to address this challenge, which favors exploring particular types of paths that are most likely to maximize the final coverage. However, manually designing a good search heuristic is nontrivial and typically ends up with suboptimal and unstable outcomes. The goal of this paper is to overcome this shortcoming of concolic testing by automatically generating search heuristics. We define a class of search heuristics, namely a parameterized heuristic, and present an algorithm that efficiently finds an optimal heuristic for each subject program. Experimental results with open-source C programs show that our technique successfully generates search heuristics that significantly outperform existing manually-crafted heuristics in terms of branch coverage and bug-finding.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging.**

ACM Reference Format:

Soyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180166

1 INTRODUCTION

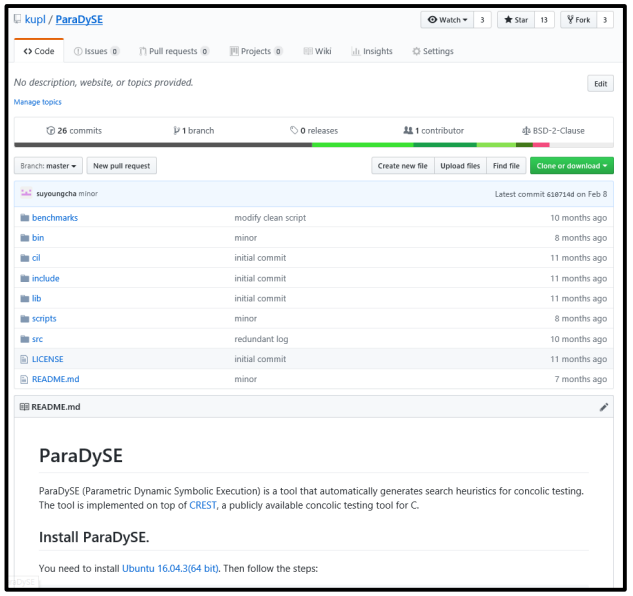
Concolic testing [15, 28] has emerged as an effective software-testing method with diverse applications [1, 7, 21, 30, 35]. The idea of concolic testing is to symbolically execute a program alongside the concrete execution, where the main job of the symbolic execution is to collect path conditions. Initially, the program is executed with a random input. After the program finishes, a branch of the current path is selected and negated to find an input that drives the next program execution to follow a previously unexplored path. This way concolic testing systematically explores the execution paths of the program, greatly improving random testing.

*Corresponding author

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180166

90% ↑



kupl / ParaDySE

Watch 3 | Star 13 | Fork 3

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

No description, website, or topics provided. Edit

Manage topics

26 commits | 1 branch | 0 releases | 1 contributor | BSD-2-Clause

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

suyoungcha minor — Latest commit 610714d on Feb 8

benchmarks	modify clean script	10 months ago
bin	minor	8 months ago
cil	initial commit	11 months ago
include	initial commit	11 months ago
lib	initial commit	11 months ago
scripts	minor	8 months ago
src	redundant log	10 months ago
LICENSE	initial commit	11 months ago
README.md	minor	7 months ago

README.md

ParaDySE

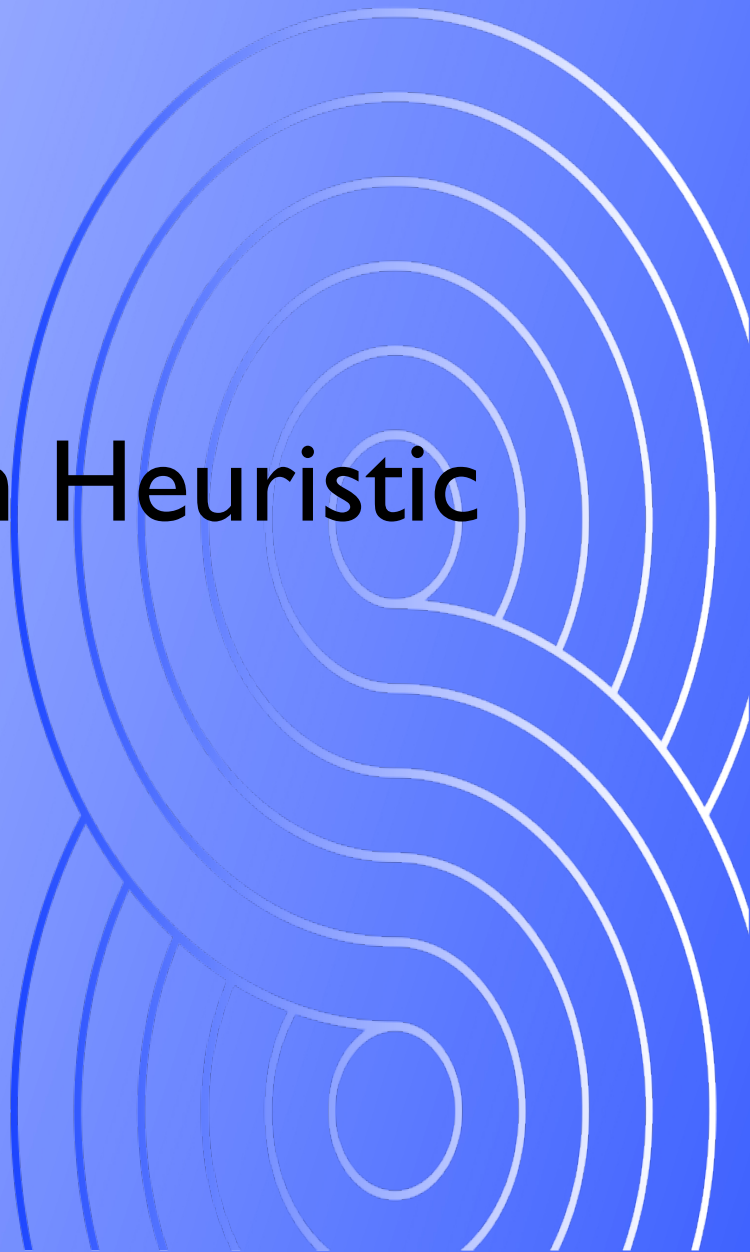
ParaDySE (Parametric Dynamic Symbolic Execution) is a tool that automatically generates search heuristics for concolic testing. The tool is implemented on top of CREST, a publicly available concolic testing tool for C.

Install ParaDySE.

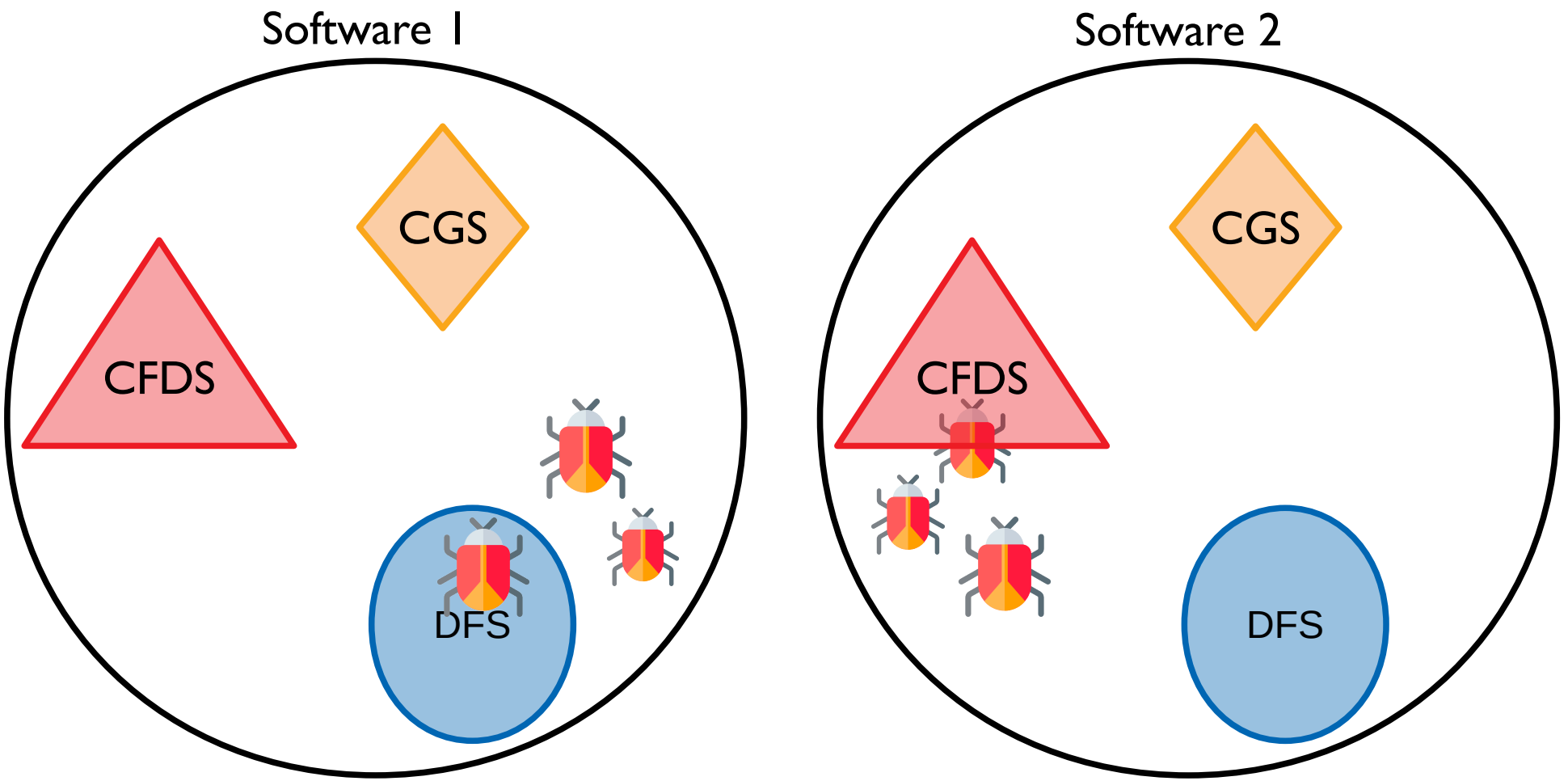
You need to install Ubuntu 16.04.3(64 bit). Then follow the steps:

- A good opportunity to use the latest techniques.

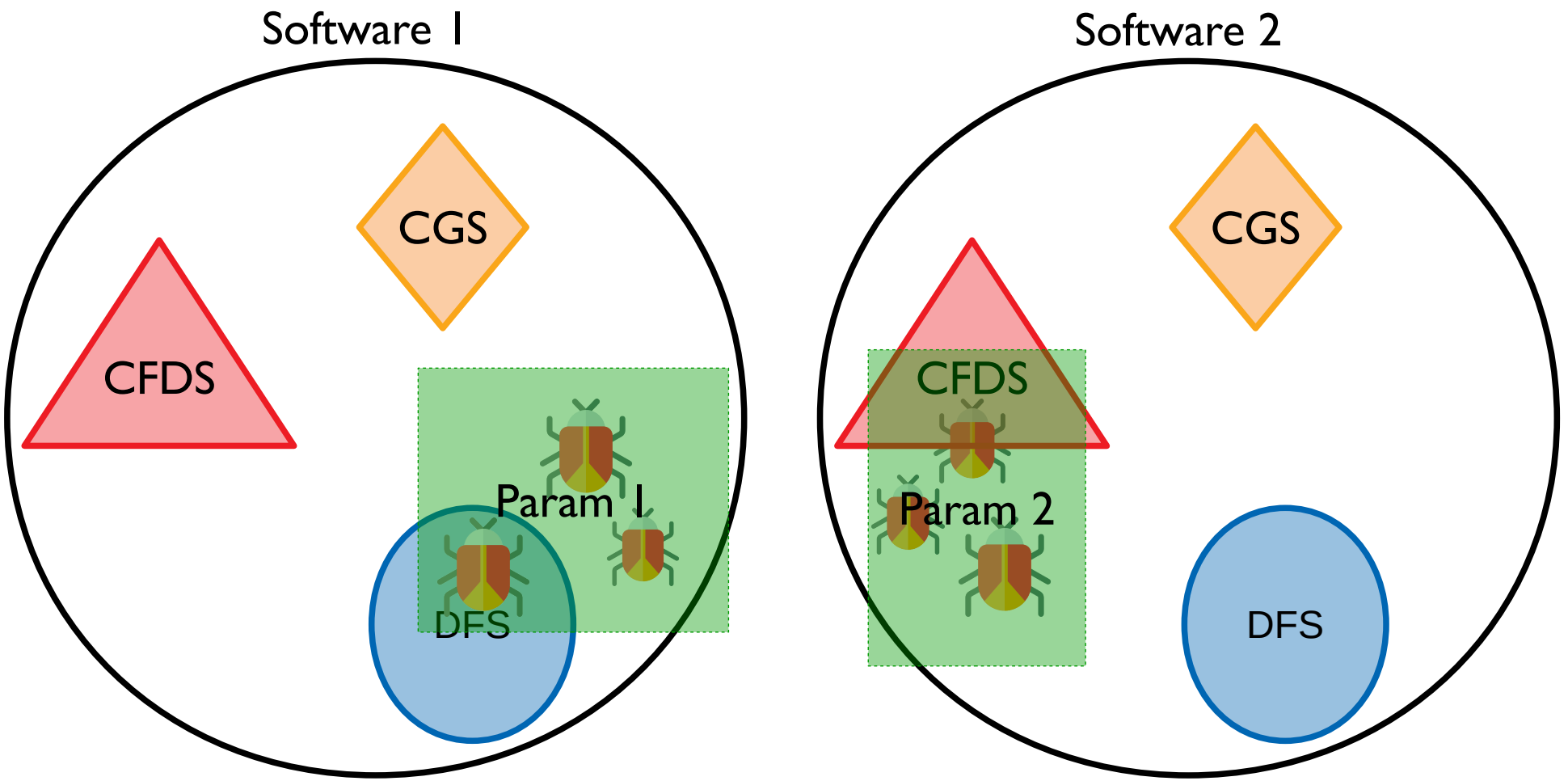
1. Parameterized Search Heuristic



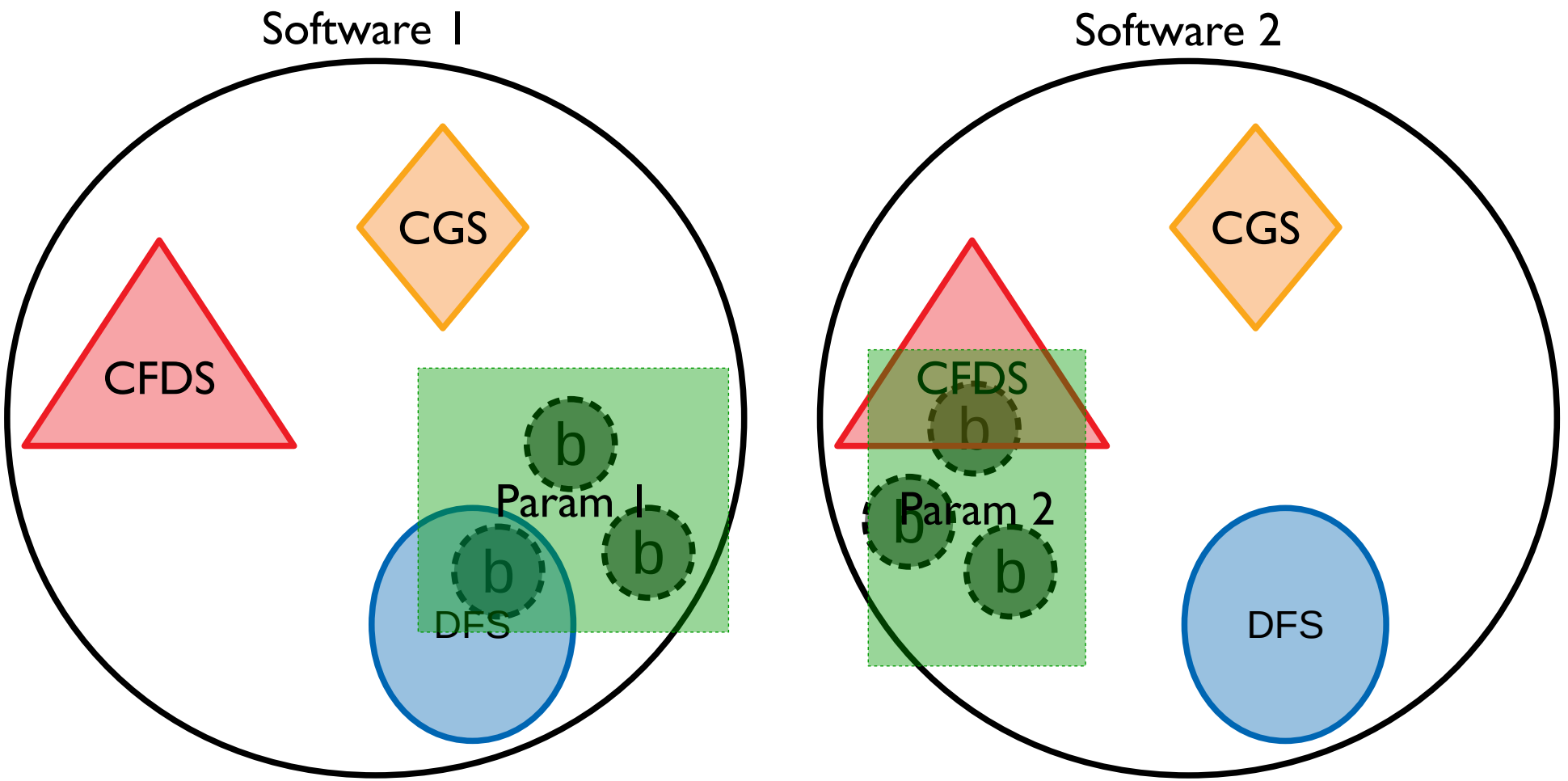
Existing Search Heuristics



Parameterized Search Heuristic

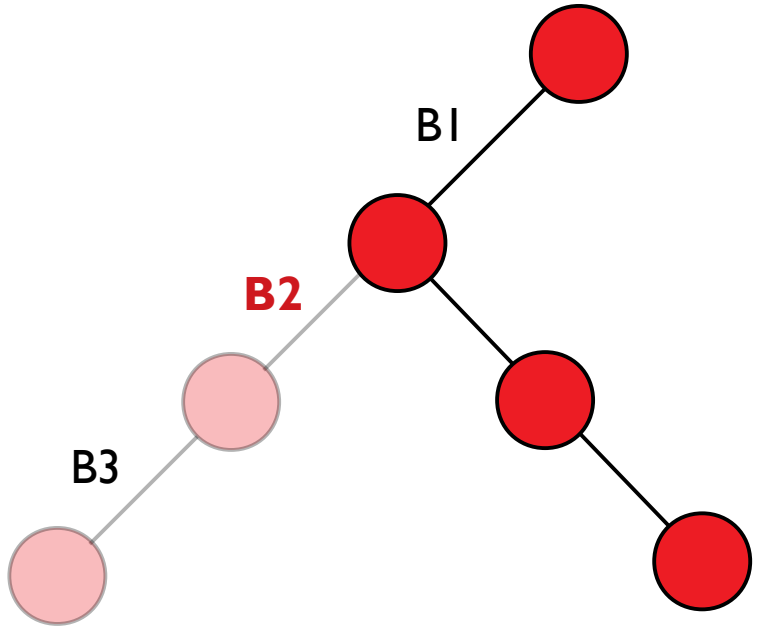


Parameterized Search Heuristic



Parameterized Search Heuristic

- Search Heuristic$_{\theta}$: Path → Branch
  - Generating a "good search heuristic" → Finding a "good parameter θ"



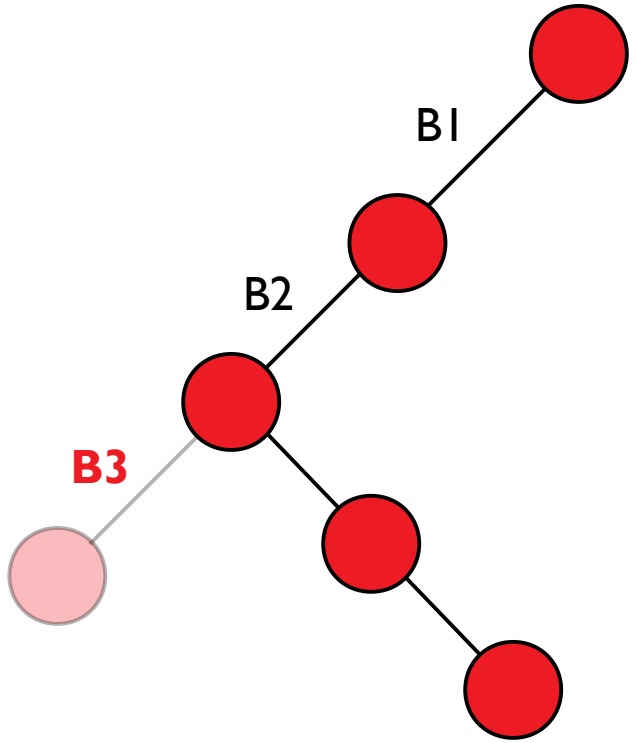
$score_{\theta}(B1) = 0.1$

$score_{\theta}(B2) = 0.7$

$score_{\theta}(B3) = -0.5$

Parameterized Search Heuristic

- Search Heuristic$_{\theta}$: Path → Branch
  - Generating a "good search heuristic" → Finding a "good parameter θ"



$score_{\theta 2}(B1) = 0.3$

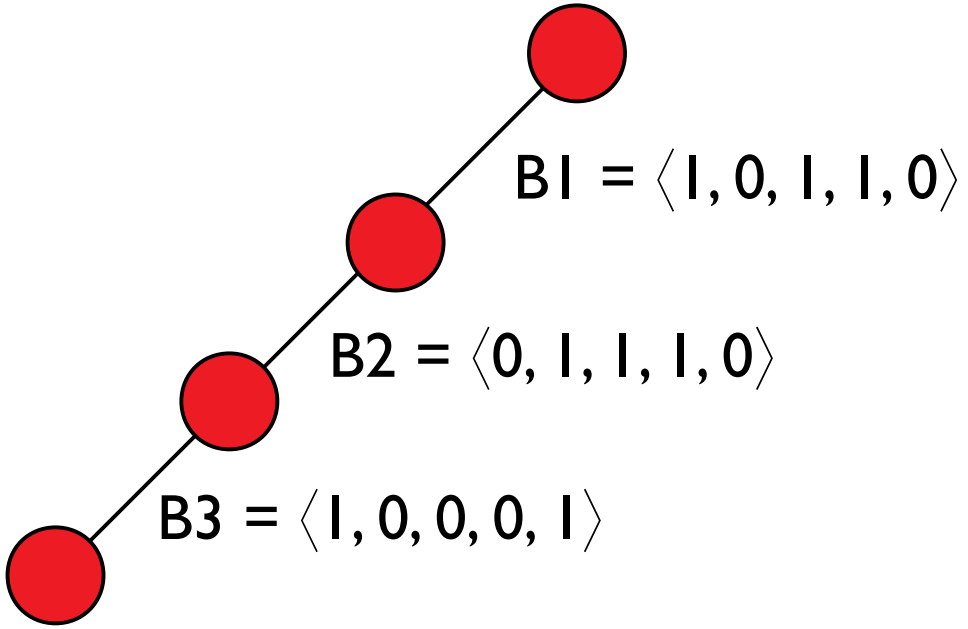
$score_{\theta 2}(B2) = -0.2$

$score_{\theta 2}(B3) = 0.9$

Parameterized Search Heuristic

(1). Represent branches as feature vectors

- A feature : a boolean predicate on branches.
  - ex1) the branch in main function ?
  - ex2) true branch of a case statement ?



Parameterized Search Heuristic

- Design 40 features.
  - 12 static features
    - extracted without execution.

      (e.g., true branch of a loop)
  - 28 dynamic features
    - Extracted at runtime.

      (e.g., branch newly covered in the previous execution)

#	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
34	branch negated in the last 30 executions
35	branch in the function that has the largest number of uncovered branches
36	the opposite branch belongs to unreached functions (top 10% of the largest func.)
37	the opposite branch belongs to unreached functions (top 20% of the largest func.)
38	the opposite branch belongs to unreached functions (top 30% of the largest func.)
39	the opposite branch belongs to unreached functions (# of branches > 10)
40	branch inside the most recently reached function

Parameterized Search Heuristic

(2). Scoring

- The parameter : a k-dimension vector.

$$\theta = \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle$$

- Linear combination of feature vector and parameter
  - $Score_\theta(B1) = \langle 1, 0, 1, 1, 0 \rangle \cdot \langle -0.5, 0{,}1, 0.4, 0.2, 0 \rangle = 0.1$
  - $Score_\theta(B2) = \langle 0, 1, 1, 1, 0 \rangle \cdot \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle = 0.7$
  - $Score_\theta(B3) = \langle 1, 0, 0, 0, 1 \rangle \cdot \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle = -0.5$

(3). Choosing the branch with the highest score

- B2

2. Parameter Search Algorithm

Parameter Search Algorithm

- Finding good parameters is crucial.
- Naive algorithm based on random sampling.

$\theta_1 = \langle -0.5, 0.1, 0.4, 0.2, 0 \rangle \rightarrow$ Coverage(519)

$\theta_2 = \langle -0.9, 0.5, 0.9, -0.2, 1.0 \rangle \rightarrow$ Coverage(423)

...

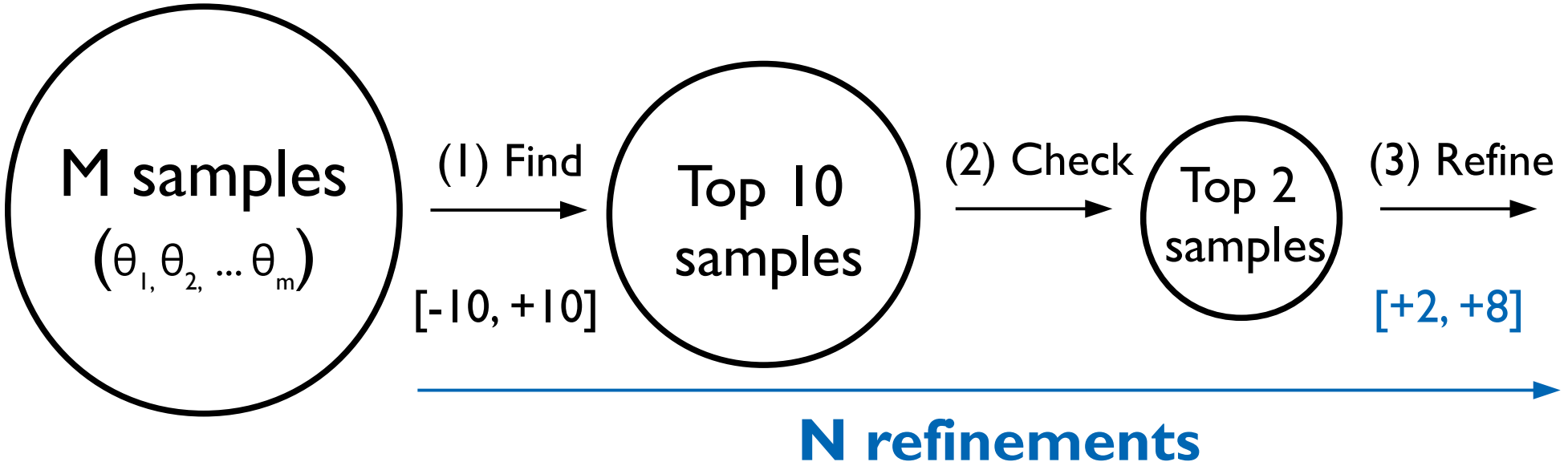
$\theta_n = \langle 0.7, -0.2, -0.9, -0.9, 0.3 \rangle \rightarrow$ Coverage(782)

Timeout → θ_{22} (best)

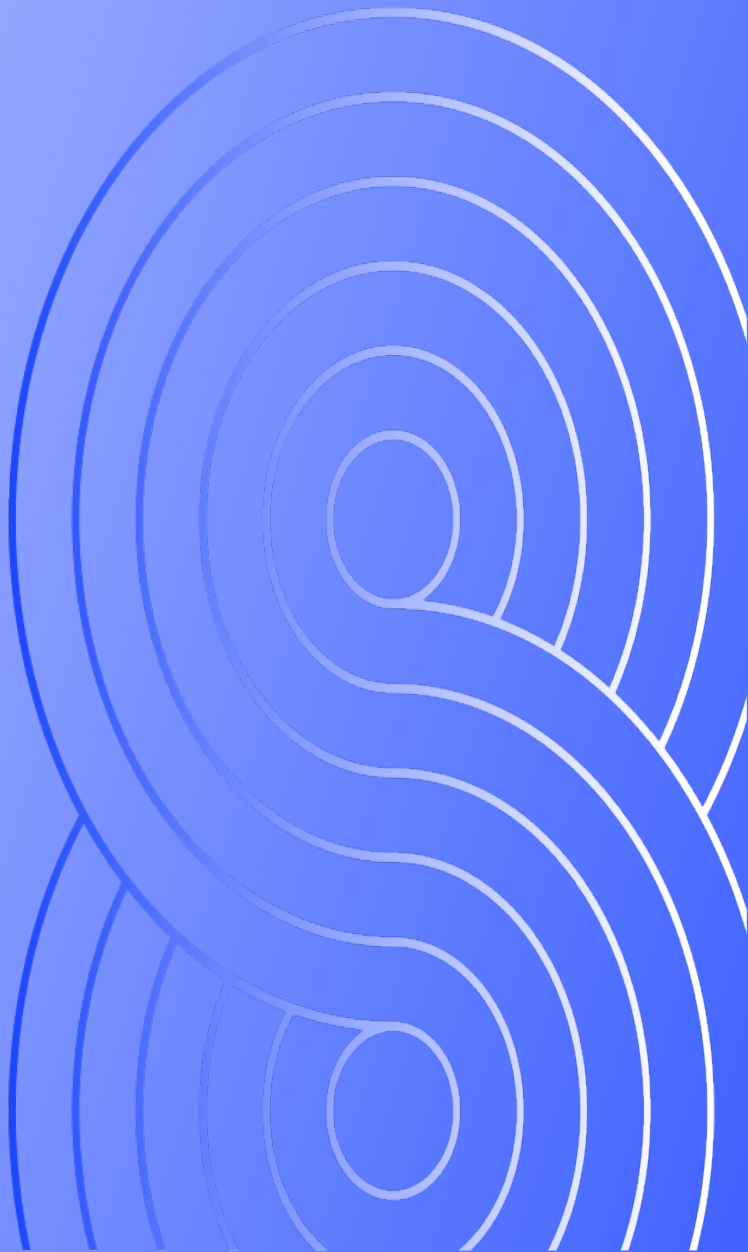
  - Failed to find good parameters.
    - Search space is intractably large.
    - Performance variation in concolic testing.

Parameter Search Algorithm

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



Experiments



Experiments

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

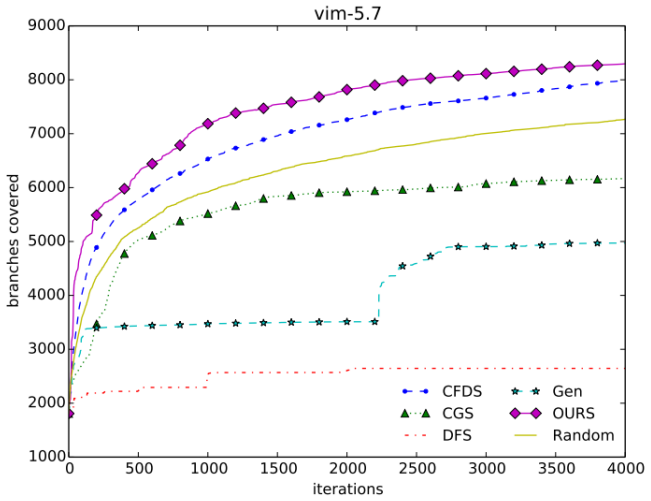
Program	# Total branches	LOC
vim-5.7	35,464	165K
gawk-3.0.3	8,038	30K
expat-2.1.0	8,500	49K
grep-2.2	3,836	15K
sed-1.17	2,656	9K
tree-1.6.0	1,438	4K
cdaudio	358	3K
floppy	268	2K
kbfiltr	204	1K
replace	196	0.5K

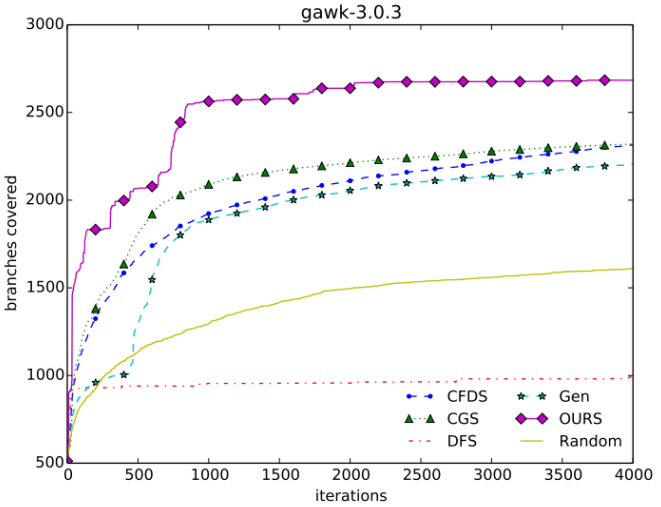
Evaluation Setting

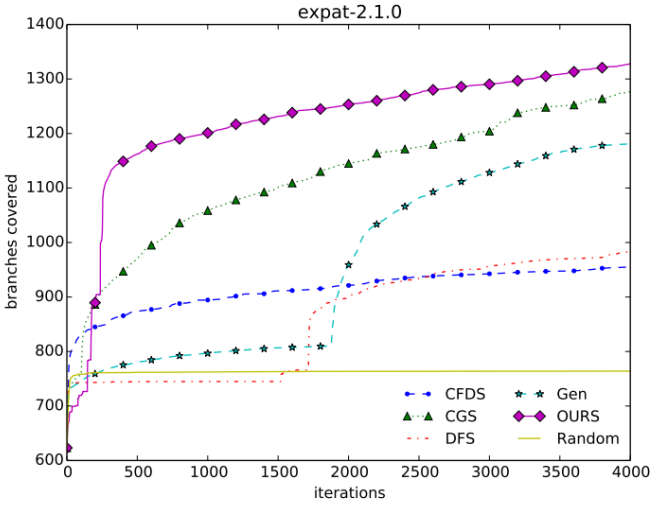
- The same initial inputs
- The same testing budget (4,000 executions)
- Average branch coverage for 100 trials (50 for vim)
  - 1 trial = 4,000 executions

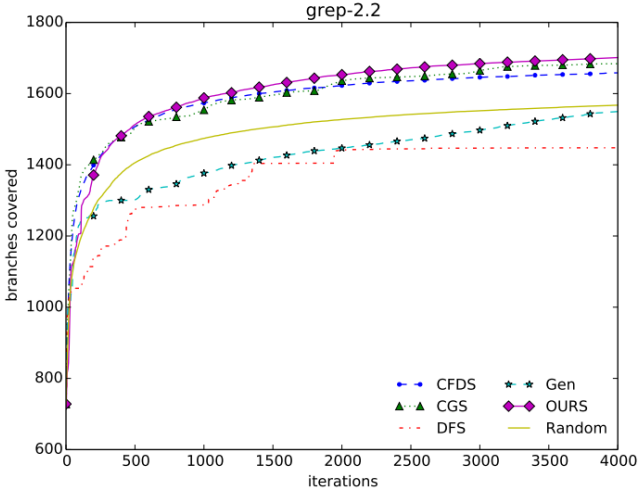
Effectiveness

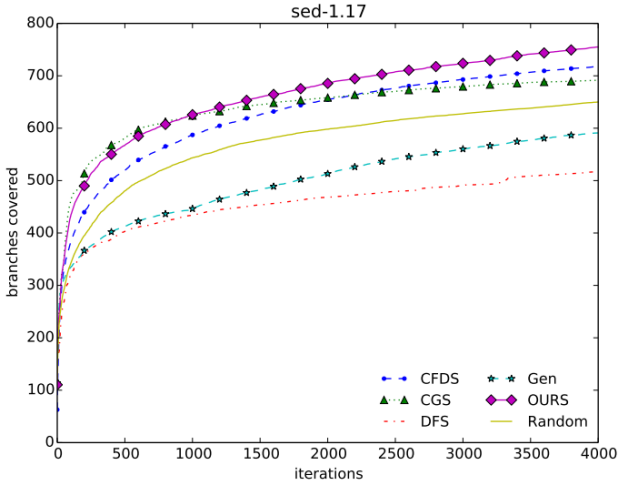
- Average branch coverage (6 Smiles 😍)

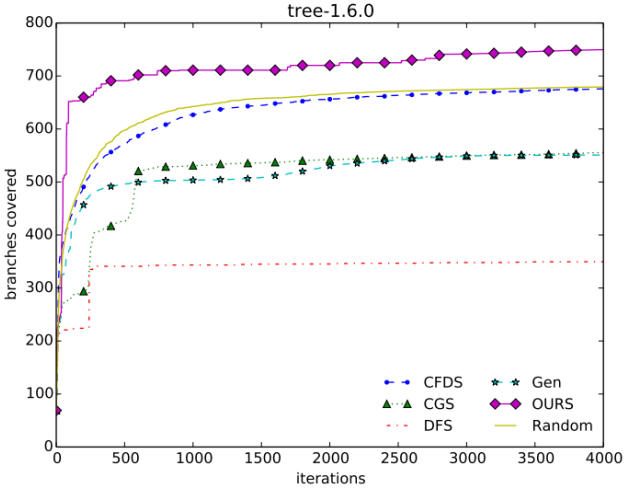






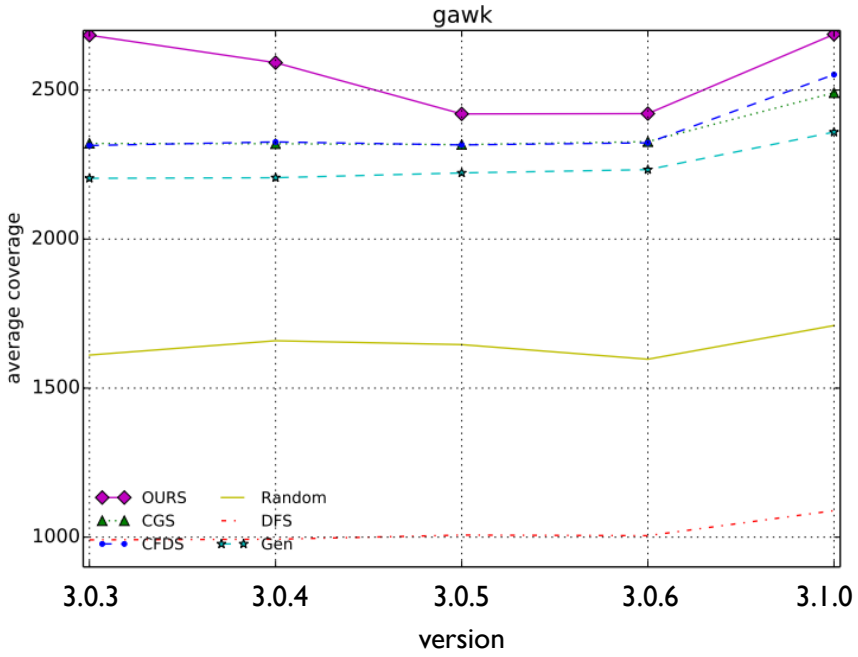




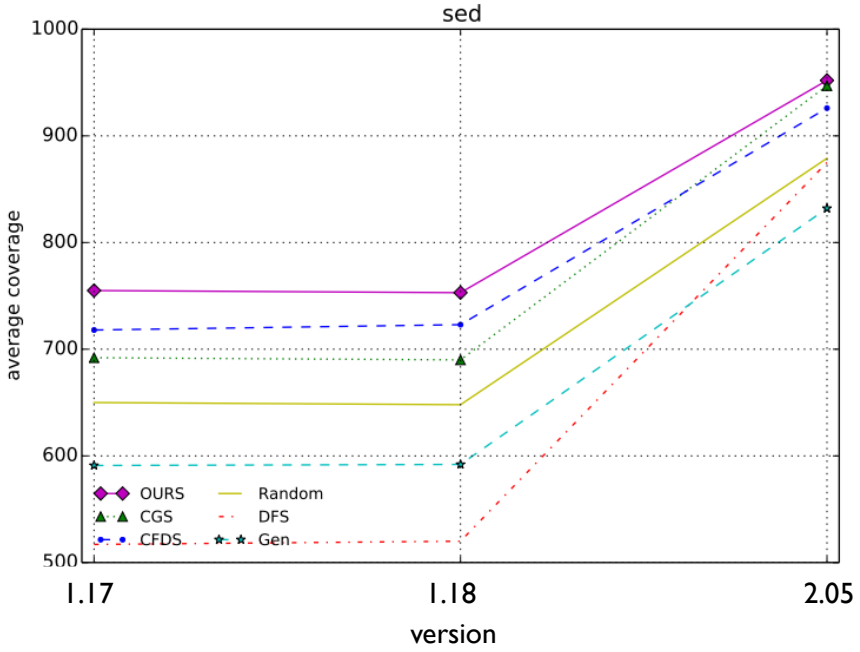


Effectiveness

- Reusable over multiple subsequent programs



(4 Years)



(A Year)

- Time for obtaining the heuristics (with 20 cores)
  - vim-5.7(24 h), expat-2.1.0(10h), grep-2.2(5h), tree-1.6.0(3h)

Important Features

Rank	Benchmarks					
	vim	gawk	expat	grep	sed	tree
1	# 15	**# 10**	# 27	# 14	# 13	# 36
2	# 18	# 13	# 30	# 40	# 2	# 15
3	# 35	# 12	# 23	# 24	# 29	# 5
4	# 40	# 38	# 31	# 1	# 3	# 25
5	# 31	# 14	# 4	# 30	# 8	# 40
6	# 7	# 9	# 9	# 38	# 30	# 9
7	# 13	# 35	# 8	# 32	# 35	# 13
8	# 3	# 31	# 15	# 17	# 6	# 39
9	# 12	# 4	# 25	# 31	# 21	# 30
10	# 10	# 33	# 7	# 29	# 16	# 22

Top 10 positive features

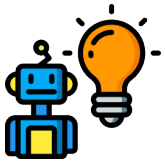
Rank	Benchmarks					
	vim	gawk	expat	grep	sed	tree
1	# 17	# 26	# 39	# 20	# 11	**# 10**
2	# 11	# 8	# 35	# 39	# 32	# 35
3	# 34	# 16	# 33	# 22	# 19	# 6
4	# 33	# 29	# 37	# 25	# 40	# 24
5	# 22	# 3	# 38	# 26	# 38	# 7
6	# 21	# 6	# 2	# 19	# 18	# 12
7	# 26	# 22	# 24	# 27	# 5	# 23
8	# 25	# 11	# 22	# 21	# 20	# 2
9	# 37	# 19	# 10	# 33	# 34	# 27
10	# 20	# 28	# 32	# 37	# 26	# 11

Top 10 negative features

- No winning feature always belongs to top 10 features.
- Depending the program, the role of feature changes. (feature 10)

Search Heuristic should be adaptively tuned for each program.

Take away

- Q1. 'Concolic Testing' 은 어떤 기술인가?
  - SW 의 오류검출을 위한 하나의 똑똑한 SW 테스팅 기법.
- Q2. 'ParaDySE' 는 어떤 도구인가?
  - 기계학습을 통해 SW 테스팅 성능을 향상시킨 오픈소스 툴.
- Q3. 마지막으로 하고 싶은 말은?
  -  > 

Thank You