Jooyong Yi

LOFT (Lab of Software), UNIST

A Bug's Life: From Its Detection through Patching to Verification

Jooyong Yi

LOFT (Lab of Software), UNIST

Contents

Part 1: Bug Hunting

White, Grey, and Black-box Fuzzing

- PC-satisfying inputs

- Mutating “interesting” inputs

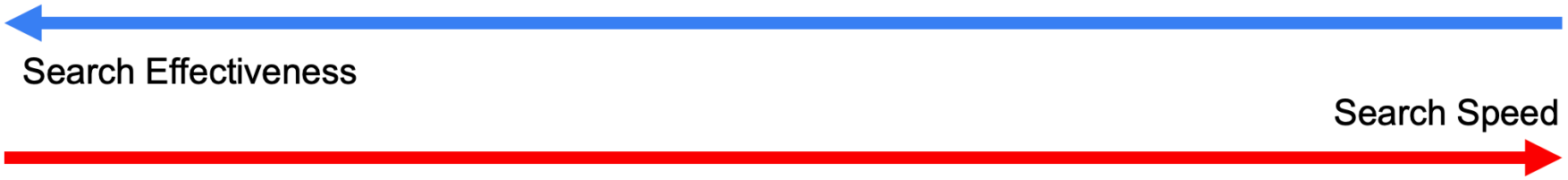
- Random inputs

White, Grey, and Black-box Fuzzing

- PC-satisfying inputs

- Mutating “interesting” inputs

- Random inputs



Grey-box Fuzzing is Not Enough

+

Grey-box Fuzzing is Not Enough

+

Path Condition

Fast Execution

Our New Approach

- Performs symbolic execution without using symbolic execution.
- Lightweight Concolic Testing via Path-Condition Synthesis for Deep Learning Libraries (ICSE 2025)
    - Sehoon Kim, Yonghyeon Kim, Dahyeon Park, Yuseok Jeon, Jooyong Yi, Mijung Kim

Key Intuition

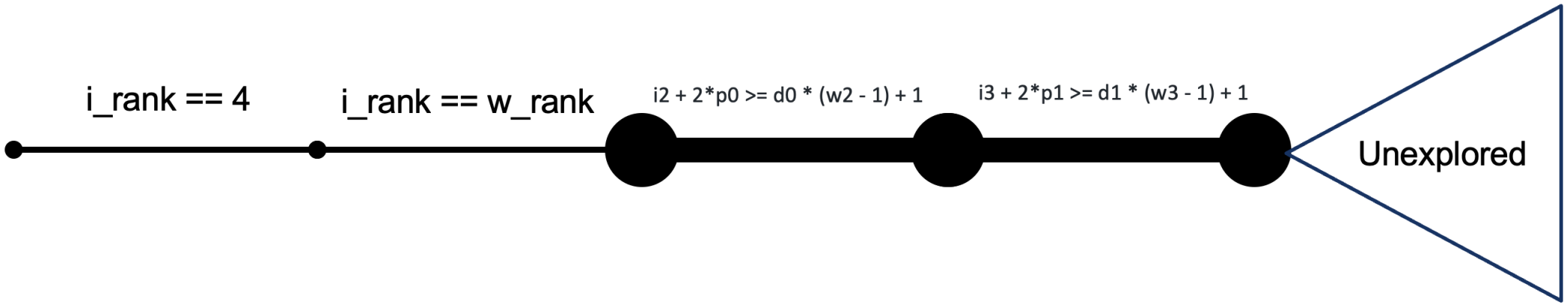
- Fuzzing generates numerous inputs.
- We can infer approximate path conditions from these inputs.

Is an Approximate PC Useful?

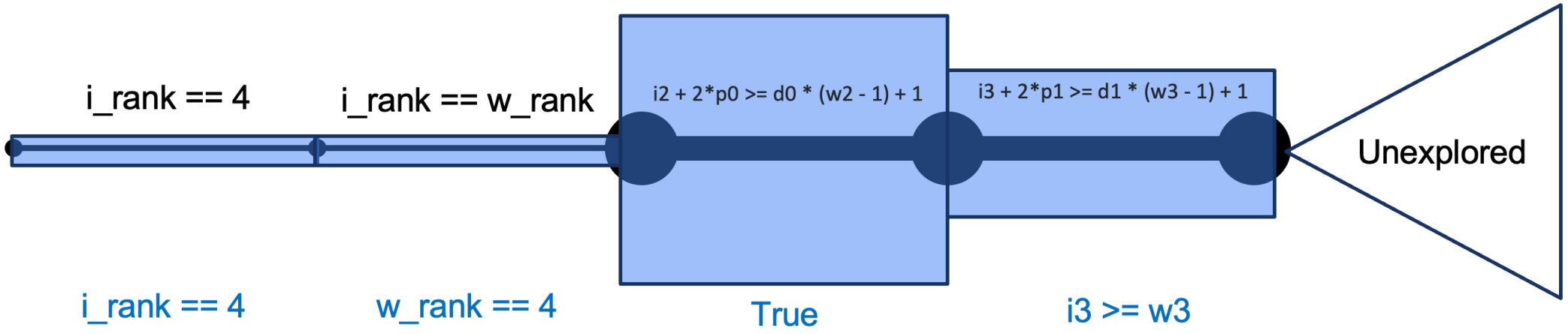
```
void conv2d(Tensor input, Tensor weight, int* padding, int* dilation) {
  TORCH_CHECK(input.dim() == 4);
  TORCH_CHECK(input.dim() == weight.dim());
  bool kh_correct = input.size(2) + 2*padding[0] >= dilation[0] * (weight.size(2) - 1) + 1;
  bool kh_correct = input.size(3) + 2*padding[1] >= dilation[1] * (weight.size(3) - 1) + 1;
  if (kh_correct && kw_correct) {
→   compute_conv2d(input, weight, padding, dilation);
    ...
```

Exact PC	Approximate PC
i_rank == 4 ∧ i_rank == w_rank	i_rank == 4 ∧ w_rank == 4
∧ i2 + 2*p0 >= d0 * (w2 - 1) + 1	∧ True
∧ i3 + 2*p1 >= d1 * (w3 - 1) + 1	∧ i3 >= w3

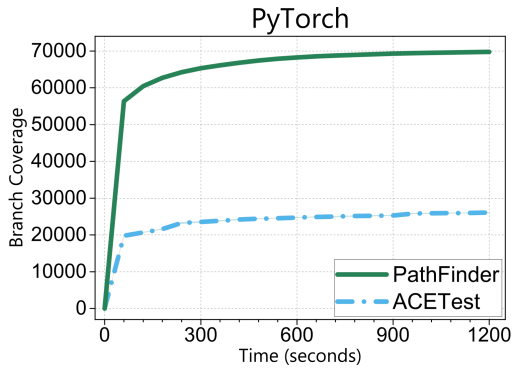
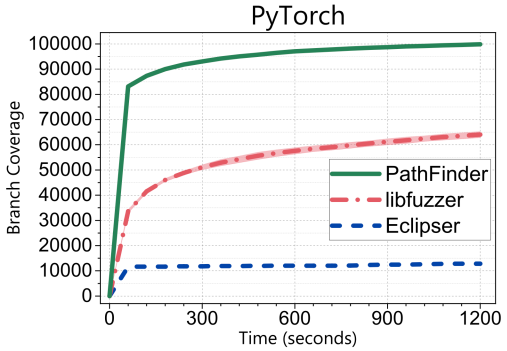
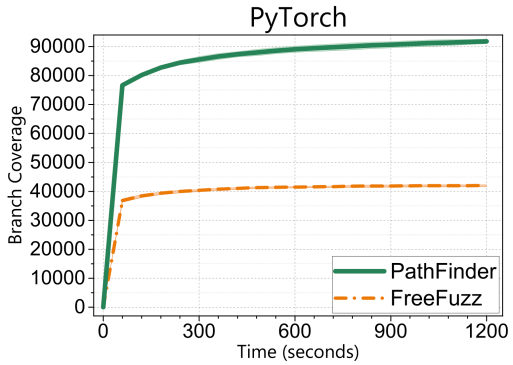
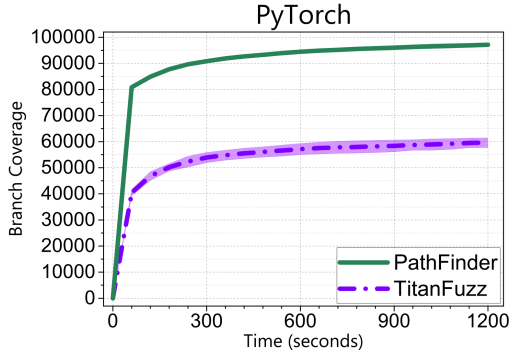
Path Exploration Using Approximate PCs



Path Exploration Using Approximate PCs

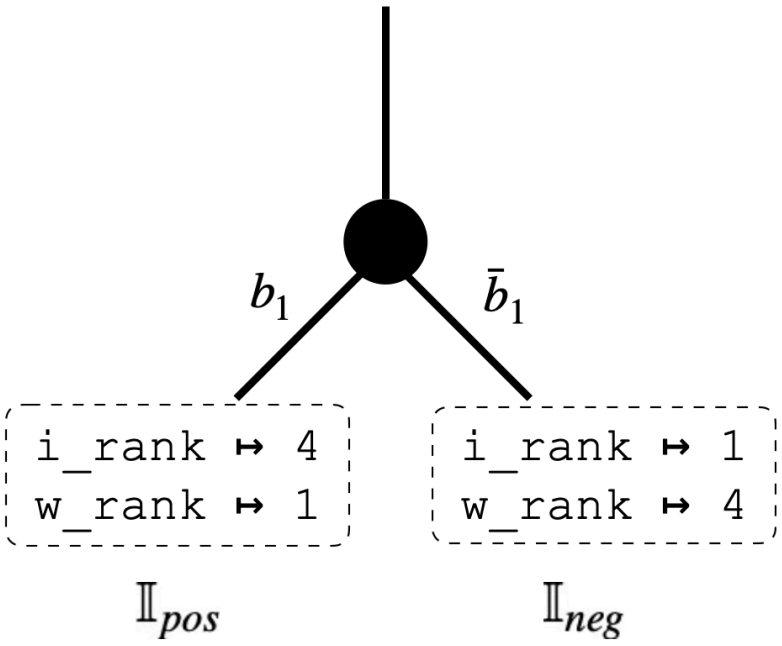


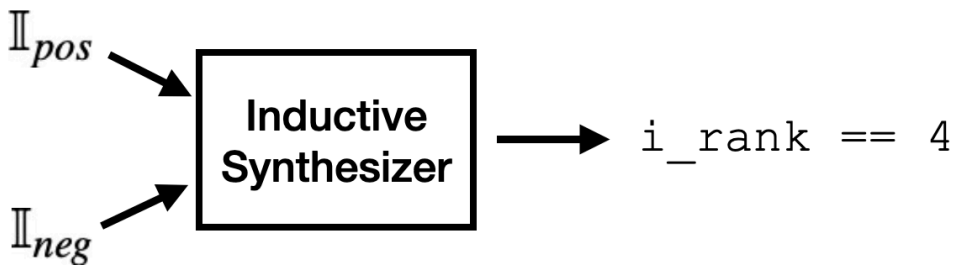
Preview of the Results

White	Grey	Black	Black
			

How to Infer Approximate PCs?

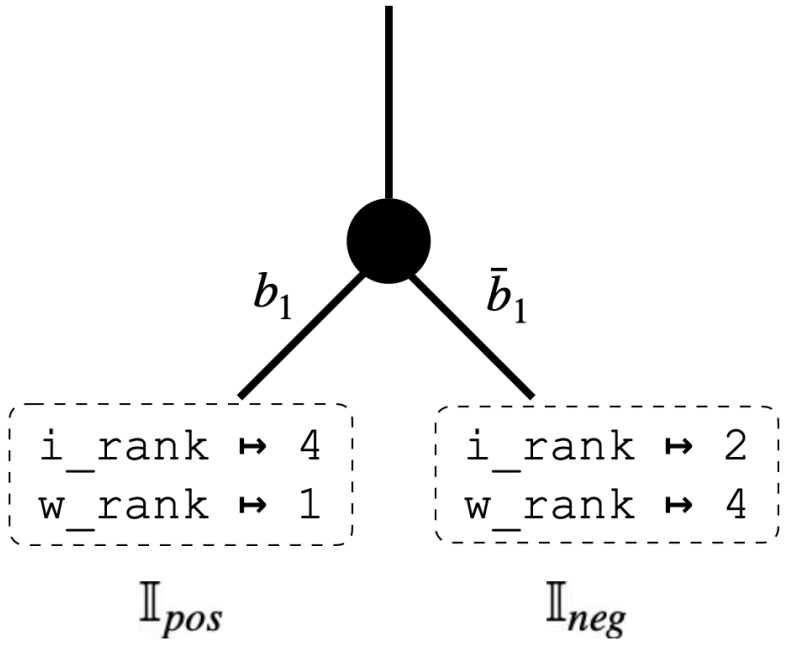
```
void conv2d(Tensor input, Tensor weight, int* padding, int* dilation) {
  TORCH_CHECK(input.dim() == 4); // b1
    ...
```

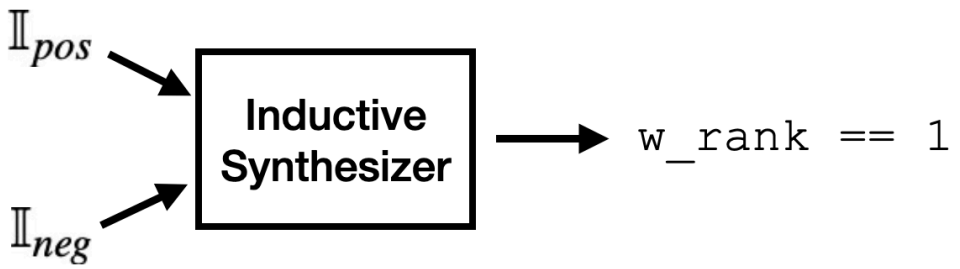




What If the Inferred PC Is Incorrect?

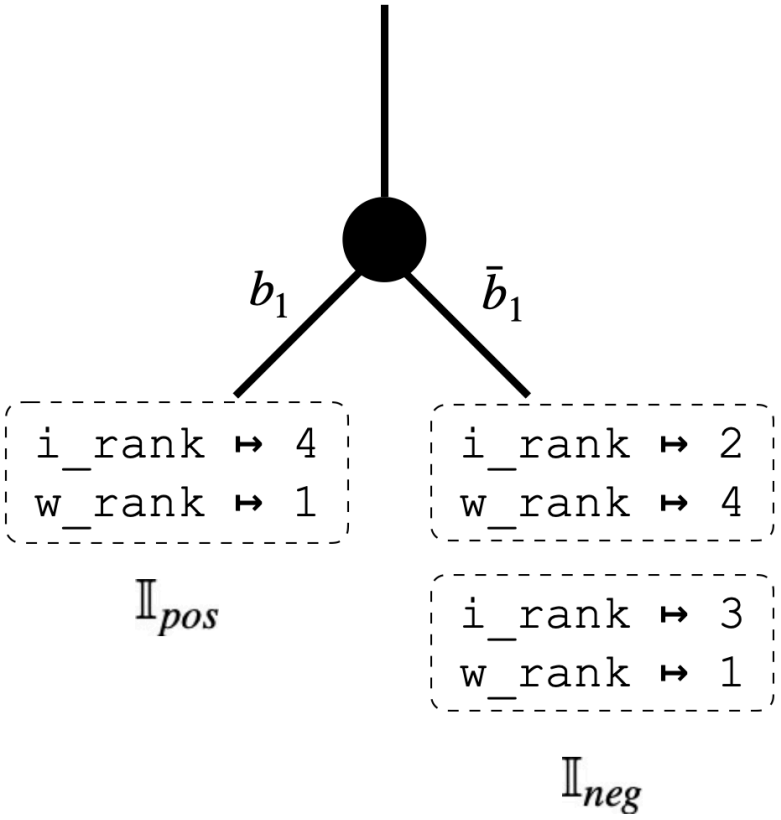
```
void conv2d(Tensor input, Tensor weight, int* padding, int* dilation) {
  TORCH_CHECK(input.dim() == 4); // b1
    ...
```

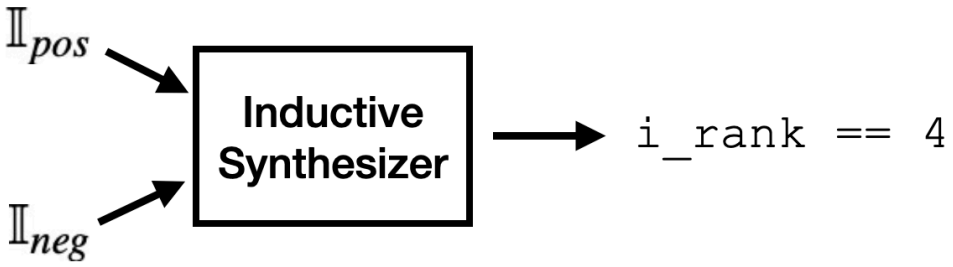




Counter-Example-Guided Condition Refinement

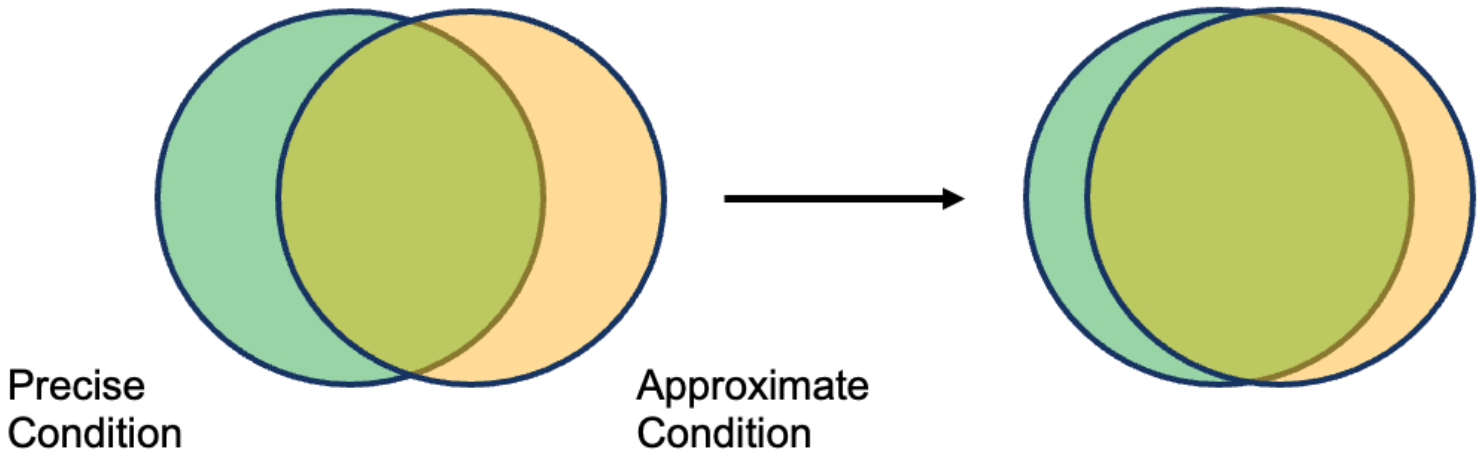
```
void conv2d(Tensor input, Tensor weight, int* padding, int* dilation) {
  TORCH_CHECK(input.dim() == 4); // b1
    ...
```



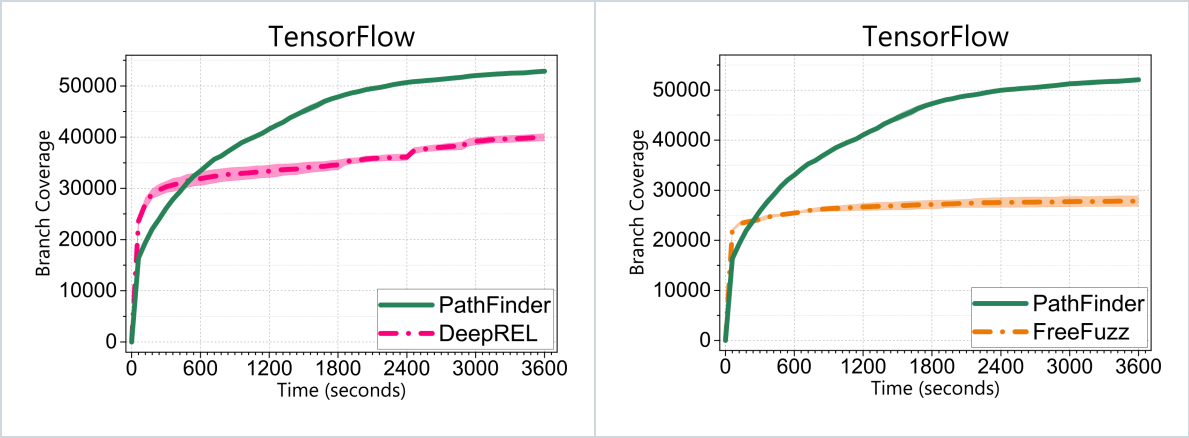


Counter-Example-Guided Condition Refinement

- As the exploration proceeds, the path conditions become more precise.



Tensorflow Results



Bug Finding Results

DL Library	Total	Confirmed	Fixed
PyTorch	43	41	23
TensorFlow	18	18	9
Total	61	59	32

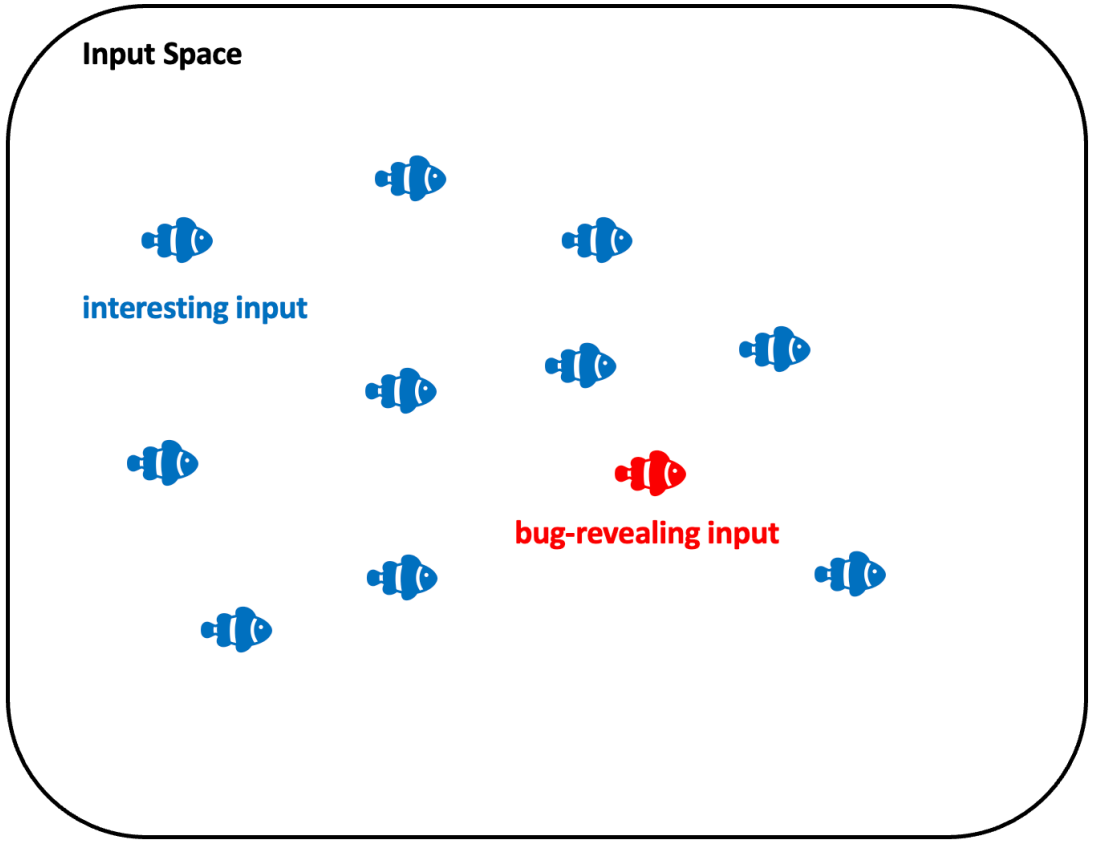
Why DL Libraries?

- Our approach is general.
- Input space of DL libraries is manageable.

Part 2: Patch Hunting

Automated Program Repair from Fuzzing Perspective

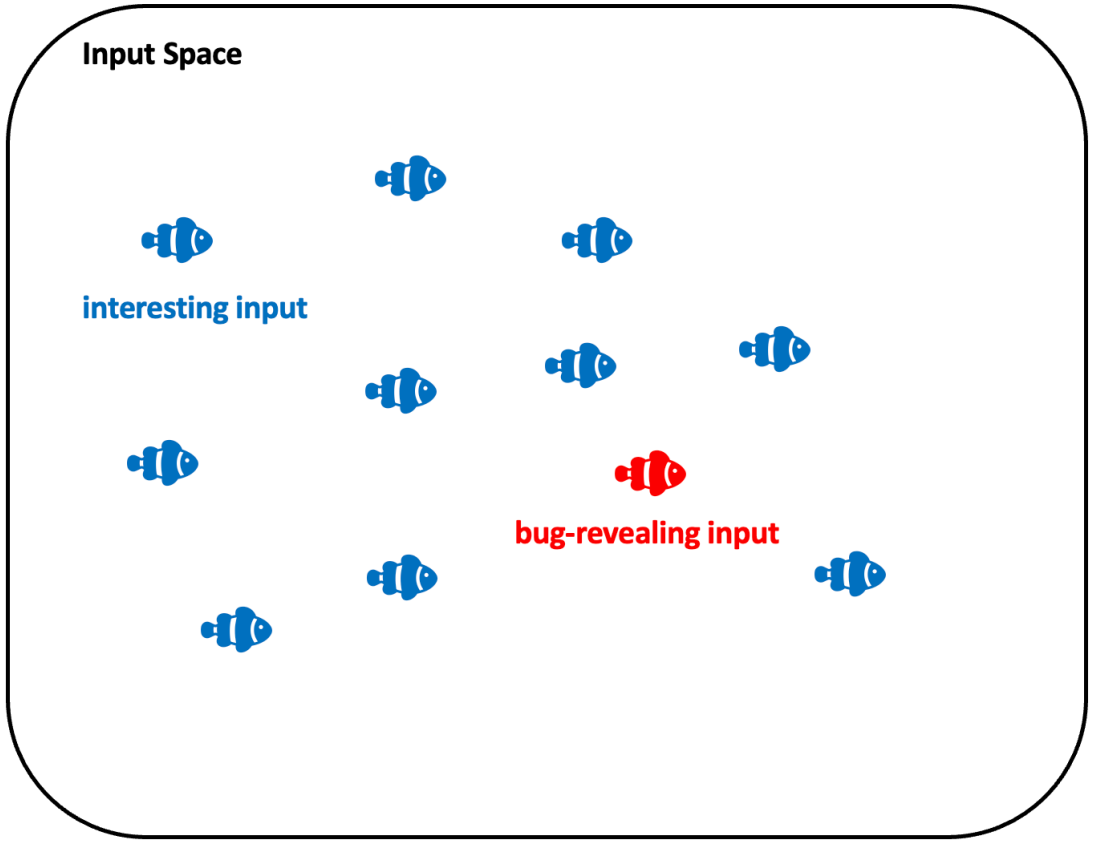
- ISSTA 2023
  - YoungJae Kim, Seungheon Han, Askar Yeltayuly Khamit, Jooyong Yi



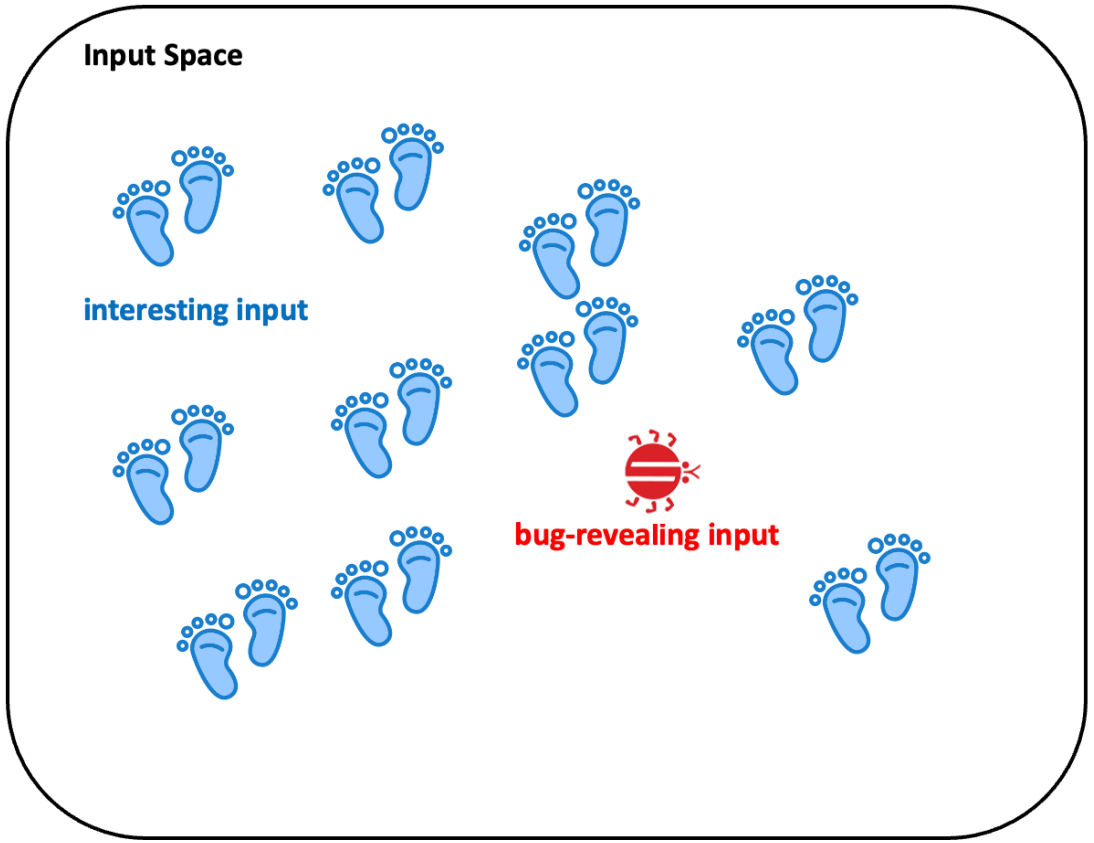
Fuzzing

≈

process of searching for interesting inputs



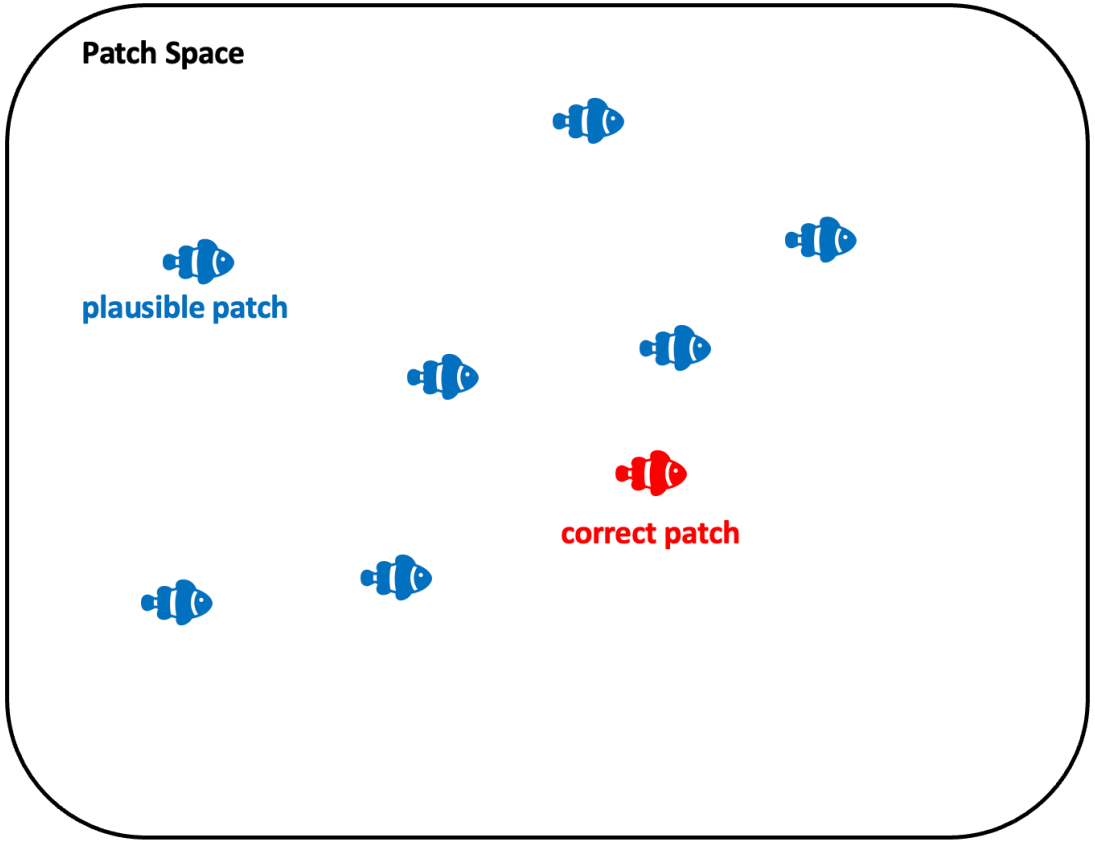
- The location of a bug-revealing input is unknown a priori.
- As more interesting inputs are found, some of them may reveal a new bug.



Fuzzing

≈

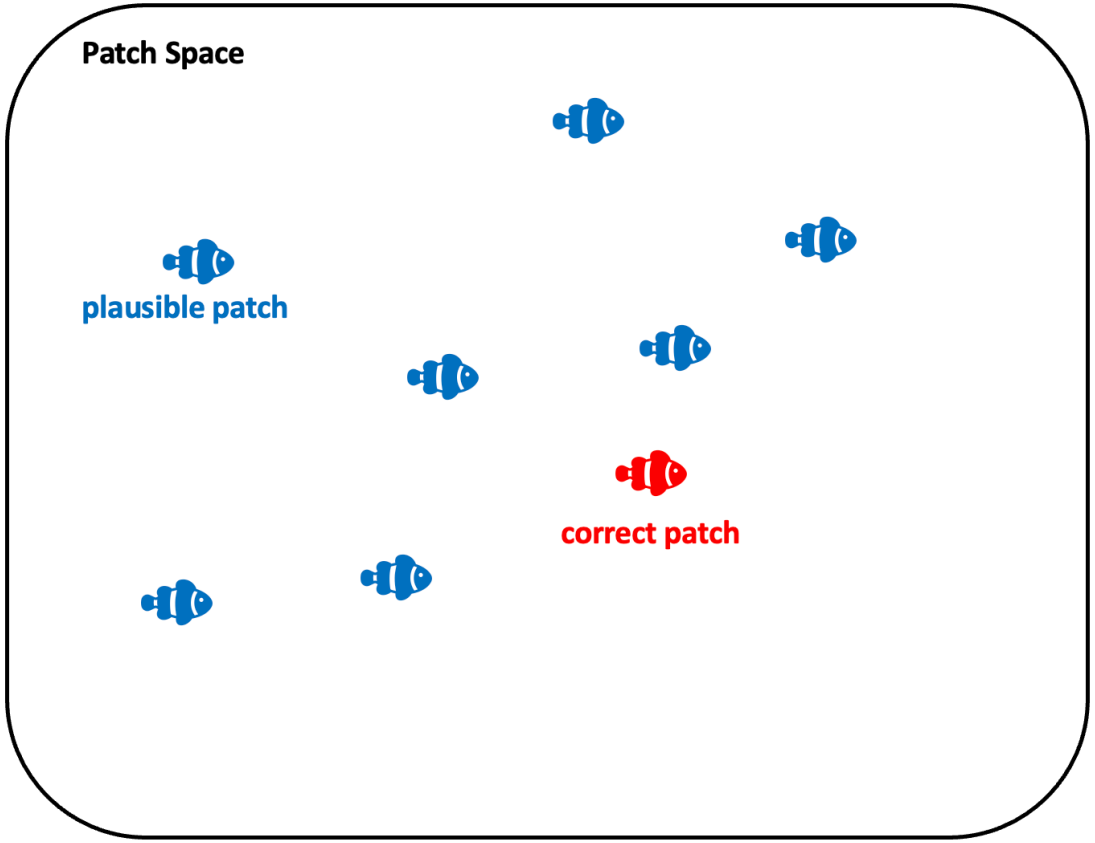
process of following footprints of interesting inputs in pursuit of a bug-revealing input



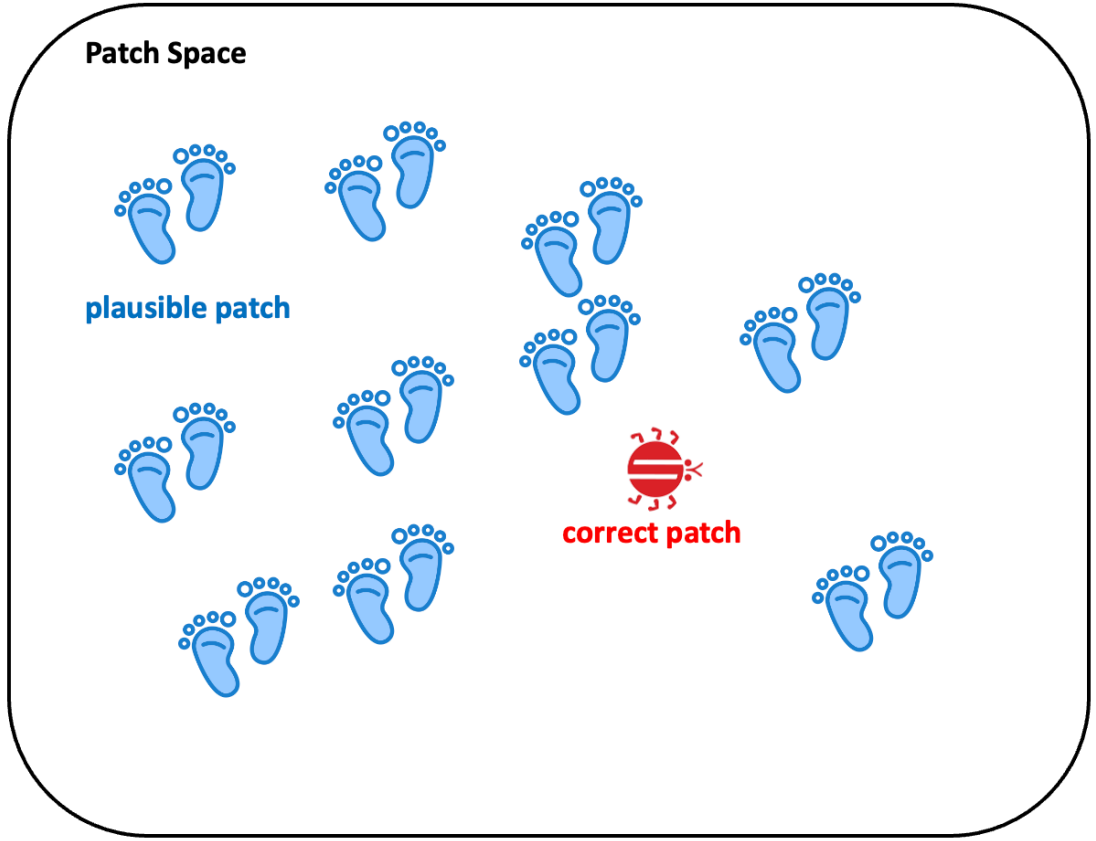
APR

≈

process of searching for plausible patches



- The location of a correct patch is unknown a priori.
- As more plausible patches are found, some of them may be correct.

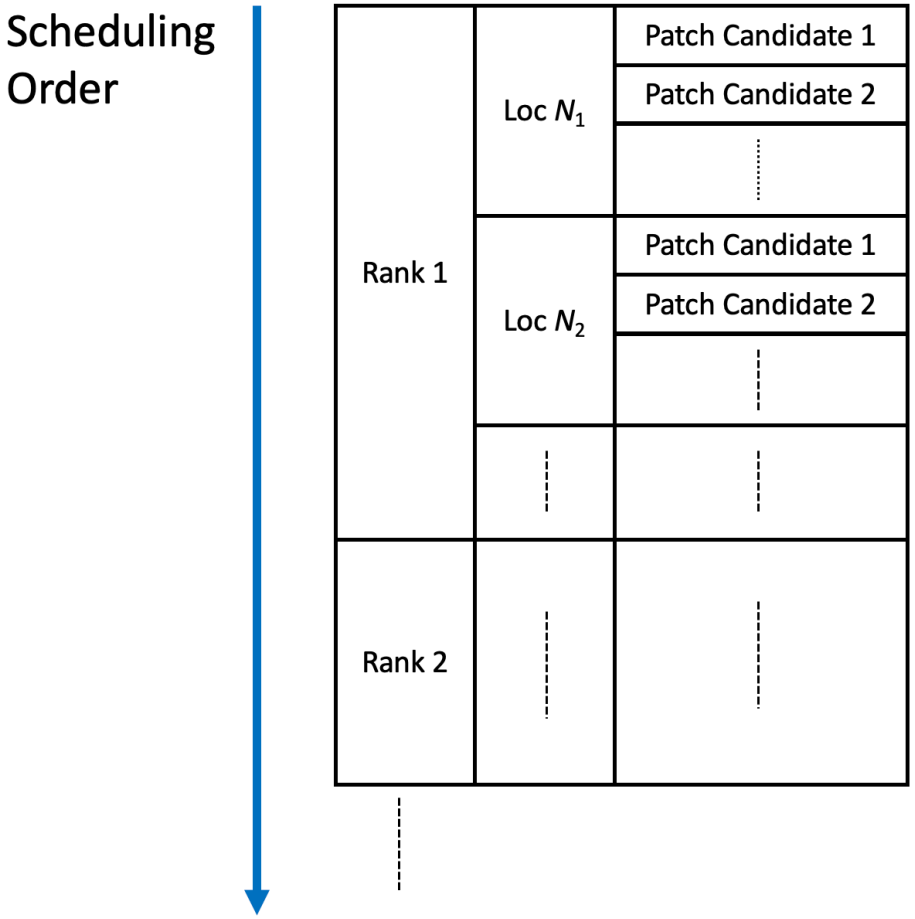


APR

≈

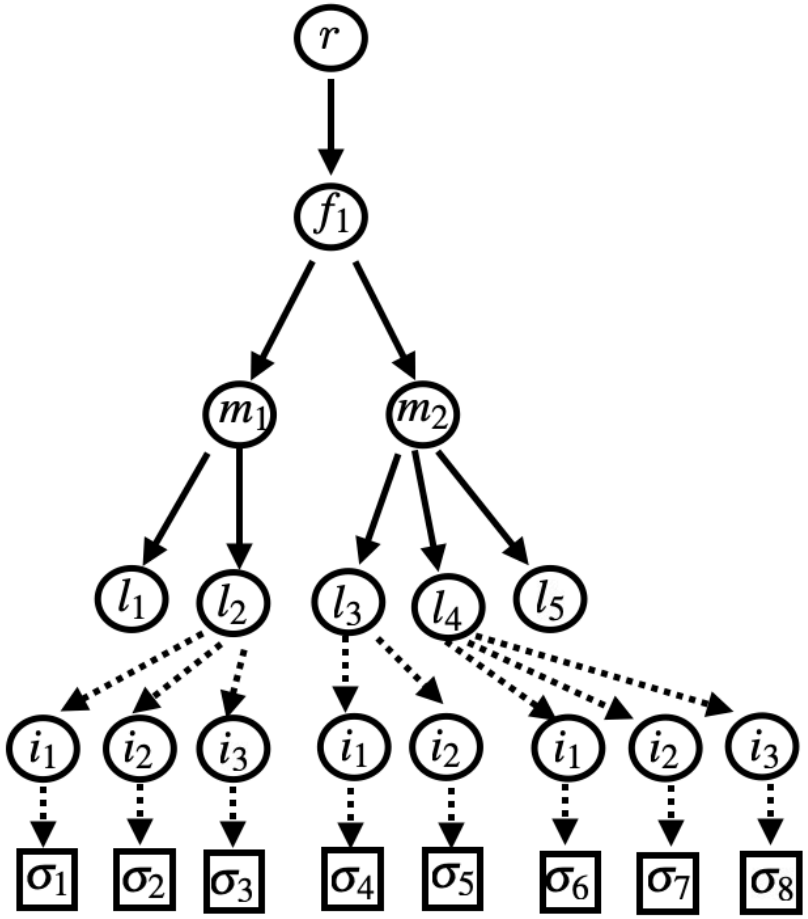
process of following footprints of plausible patches in pursuit of a correct patch

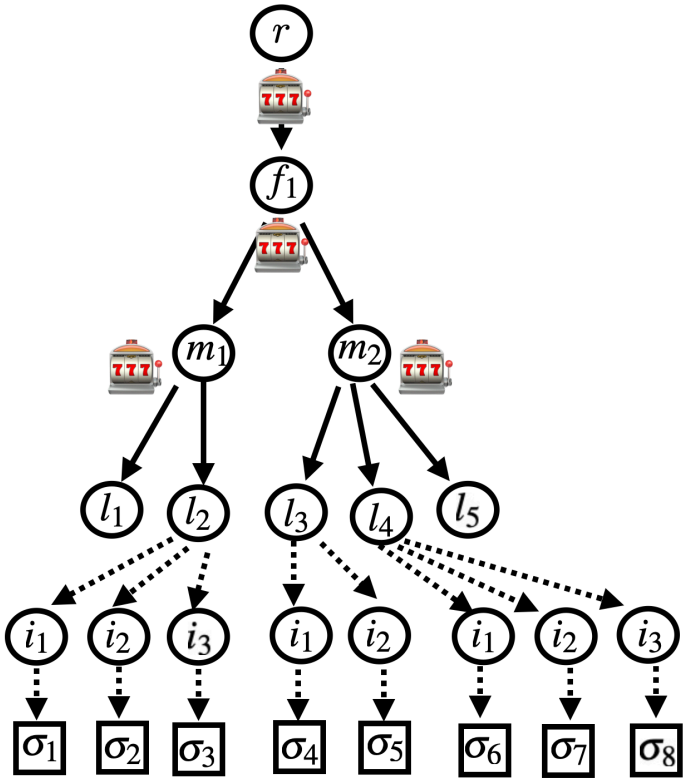
Standard Patch Scheduling Algorithm



Evaluation Results of Our Approach

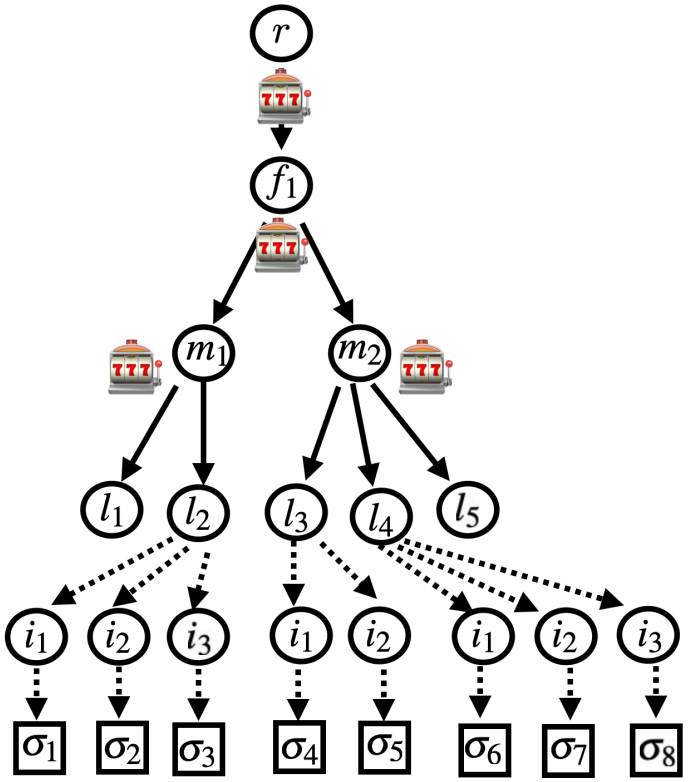
Patch Space





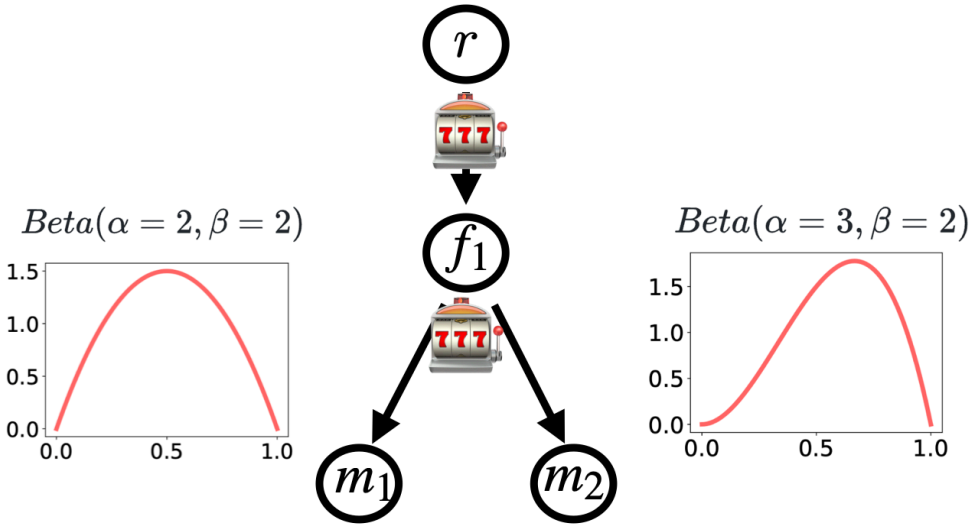
Multi-Armed Bandit Problem

- At each layer, we need to choose one "arm" to pull.
- A reward is given if an "interesting" patch is found.
  - A patch σ is considered *interesting* when the program patched with σ passes one of the tests that previously failed.
- Our goal is to maximize the total reward over time.



Bernoulli Bandit Problem

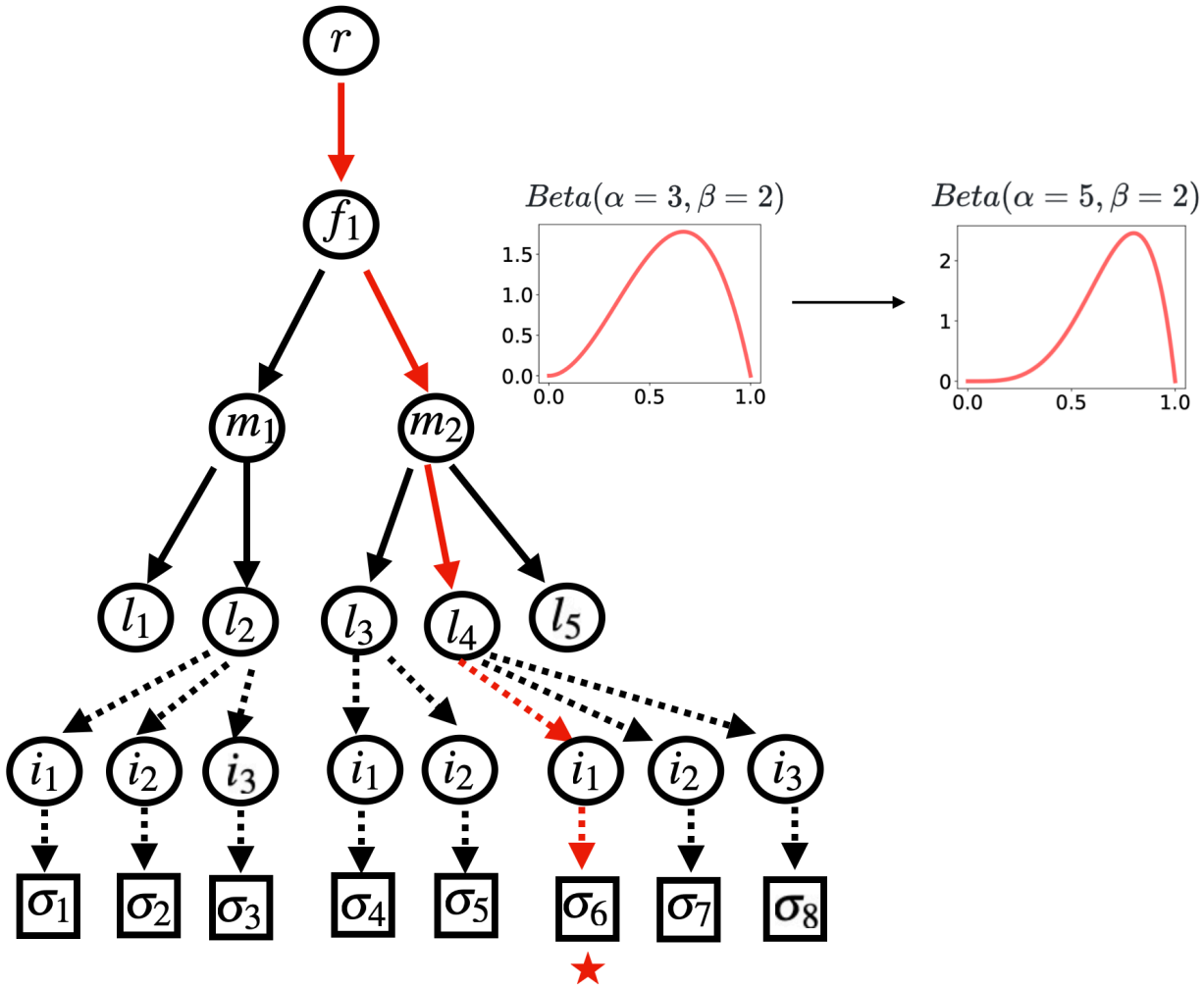
each arm k produces a reward of $\begin{cases} 1 \text{ with probability } \theta_k \\ 0 \text{ with probability } 1 - \theta_k \end{cases}$

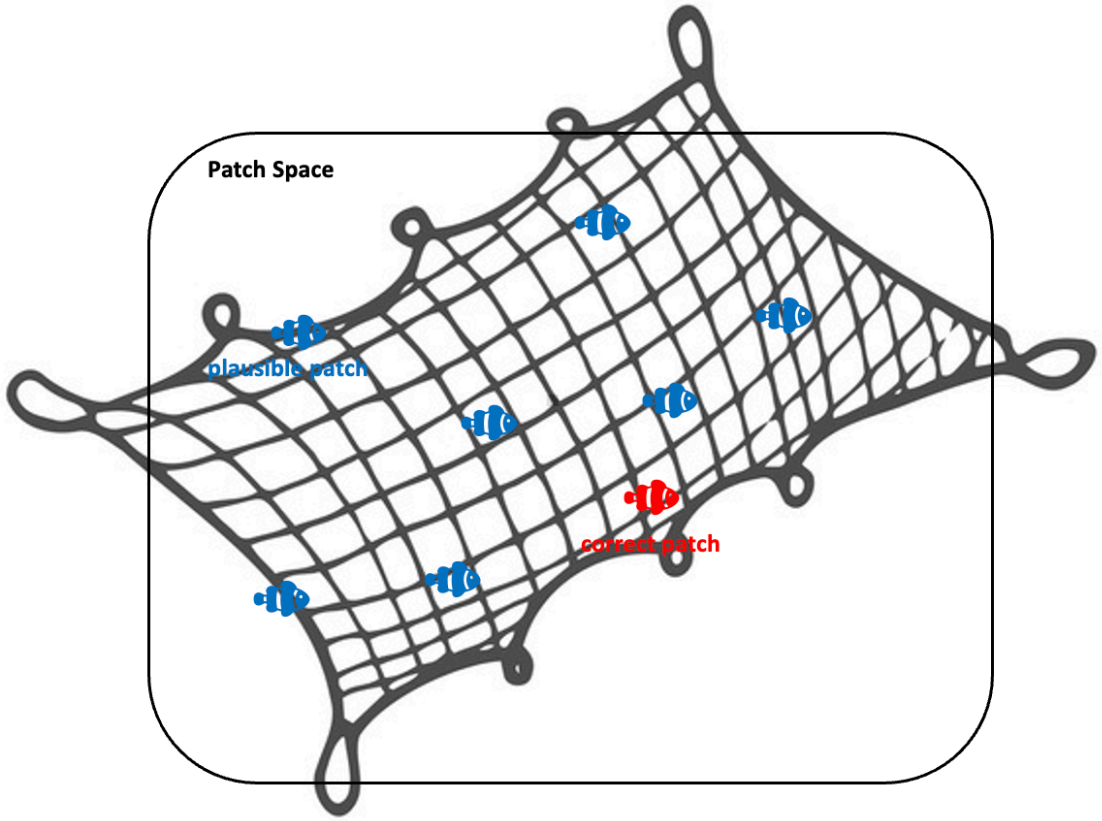


Thompson Sampling Algorithm

1. Sampling: for each arm k, sample θ_k from $Beta(\alpha_k, \beta_k)$
2. Selection: select the arm with the highest sampled θ_k
3. Update: update $Beta(\alpha_k, \beta_k)$

Updating θ_k

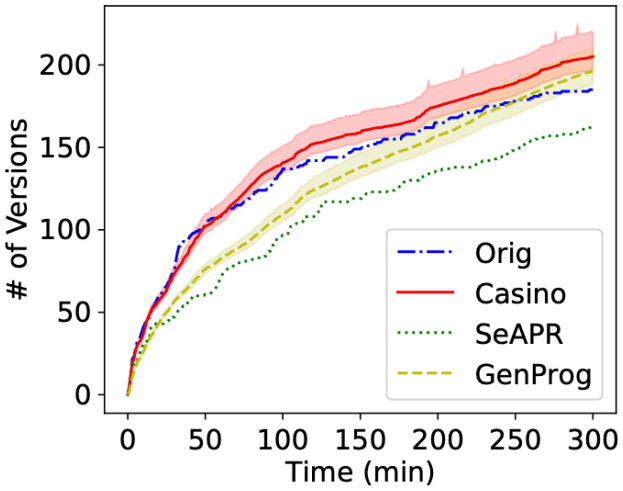
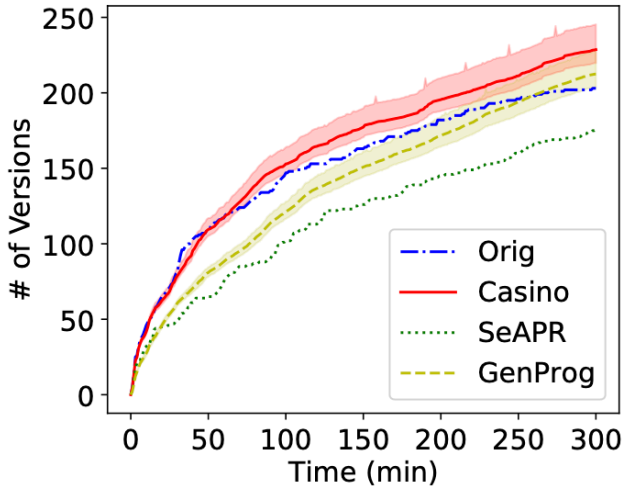


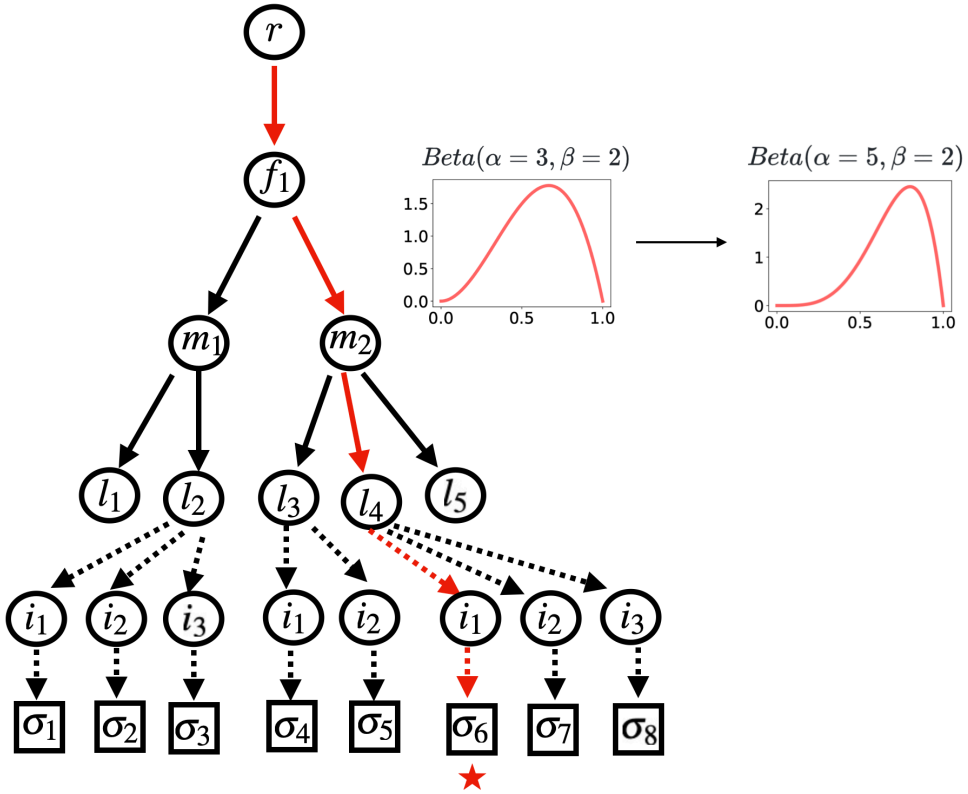


Does Our Approach Fix More Bugs Correctly?

1. Catch plausible patches
2. Rank them using a patch ranking technique

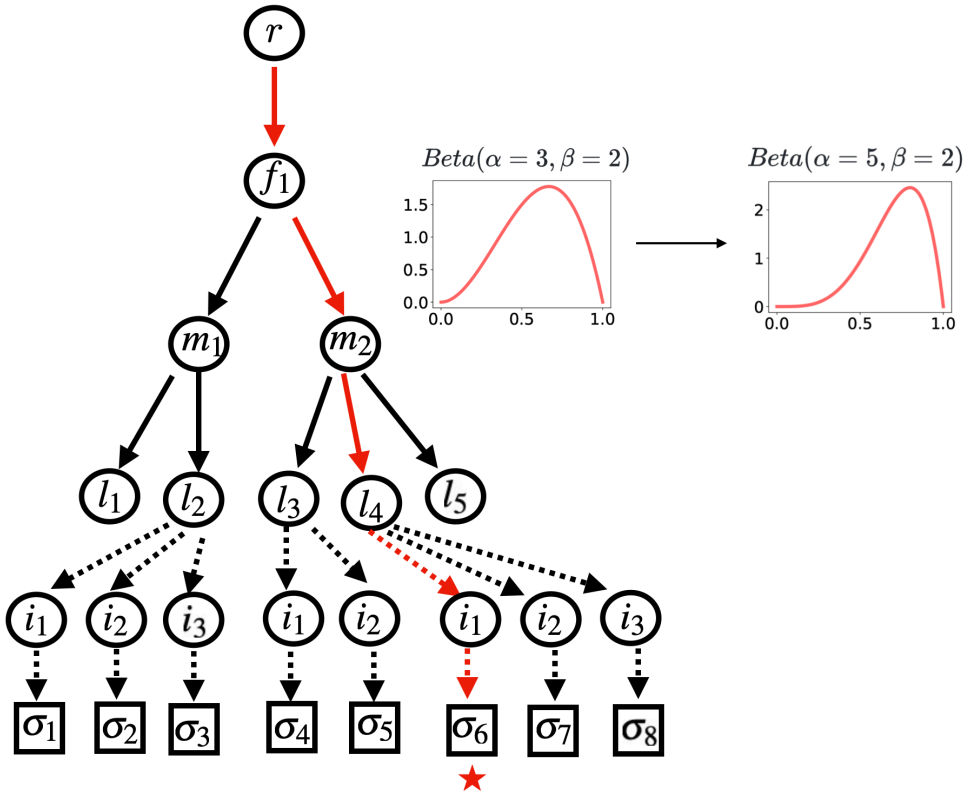
Results on Recalling Correct Patches

Top 1	Top 5
	



Reflection

- Update the distribution when an interesting patch is found: a black-box approach



Reflection

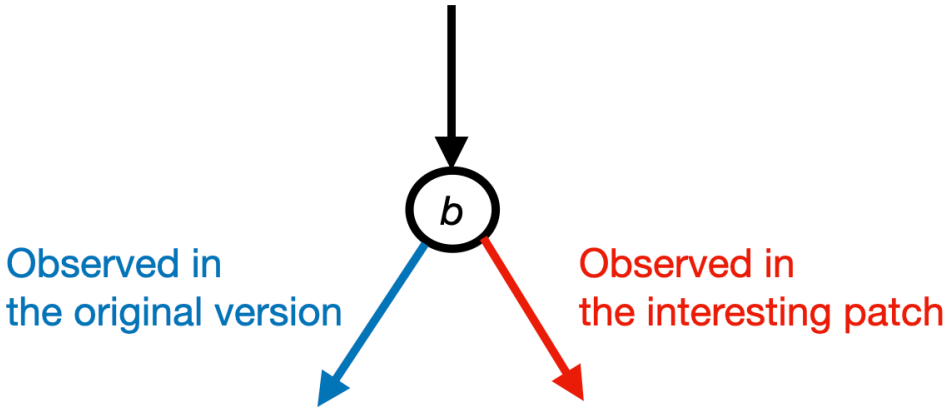
- Update the distribution when an interesting patch is found: a black-box approach
- Can we invent a grey-box approach that performs better than the black-box approach?

Enhancing the Efficiency of Automated Program Repair via Greybox Analysis

- ASE 2024
  - YoungJae Kim, Yechan Park, Seungheon Han, Jooyong Yi

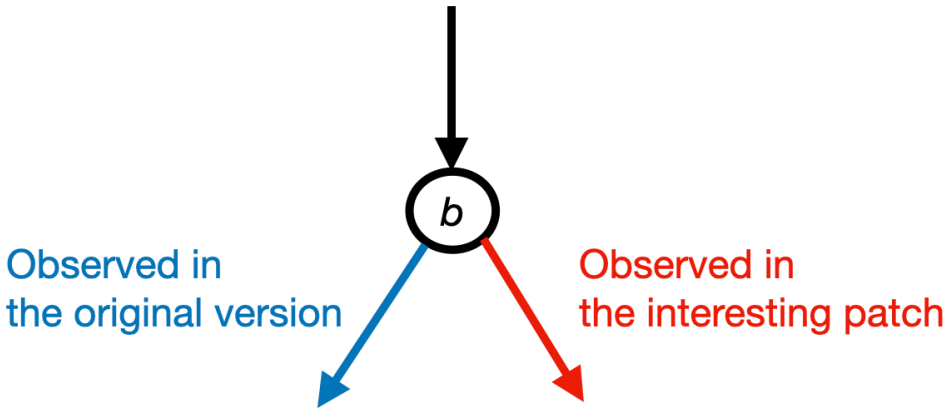
Two Key Questions

1. What to observe?
2. How to guide the search based on the observation?



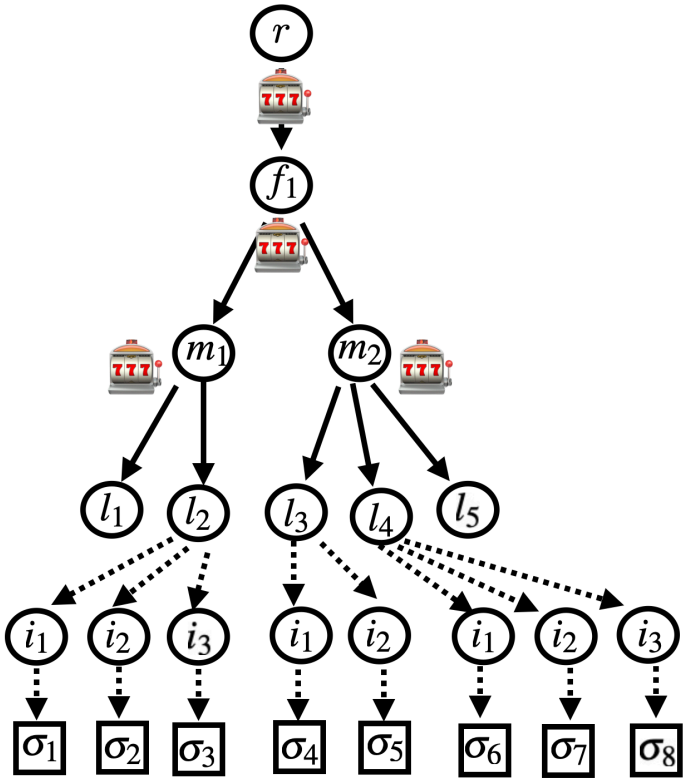
What to Observe?

- Critical branch: a branch whose hit count changes before and after an interesting patch is applied



What to Observe?

- Critical branch: a branch whose hit count changes before and after an interesting patch is applied
  - Positive critical branch: a critical branch whose hit count increases
  - Negative critical branch: a critical branch whose hit count decreases

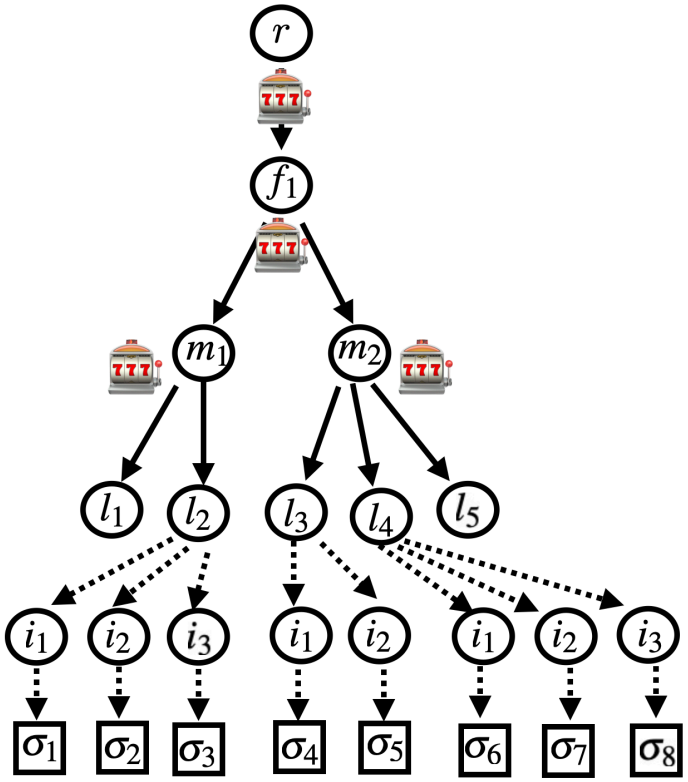


How to Guide the Search?

- We again traverse the patch-space tree using the multi-armed bandit model.
- We choose an edge that is more likely to lead to a patch candidate that **behaves similarly** to the interesting patches found earlier during the repair process.

Count-based Similarity of Patch Behavior

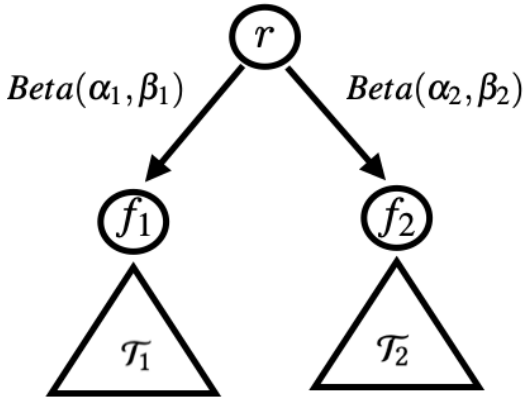
- Suppose that an interesting patch $\sigma_\star$ is found earlier during the repair process.
- Assume $\sigma_\star$ involves a positive critical branch b.
- Patch σ is considered similar to an interesting patch $\sigma_\star$ if
  - the hit count of b increases after applying σ



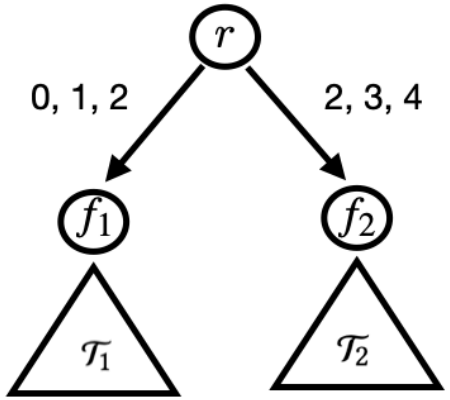
Our Greybox Guidance Policy

- We choose an edge that is more likely to lead to a patch candidate that **shows count-based similarity** to the interesting patches found earlier during the repair process.

Blackbox vs Greybox



Blackbox

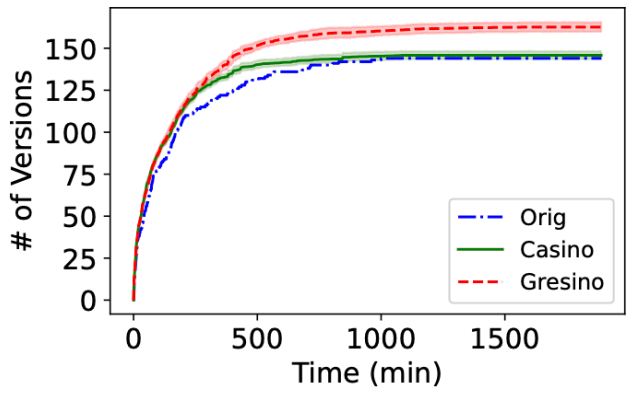
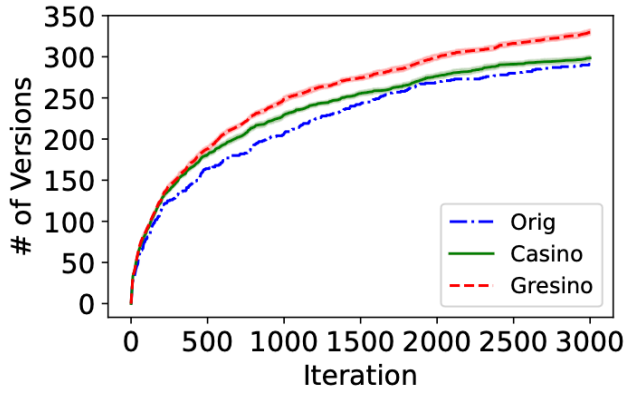


Greybox

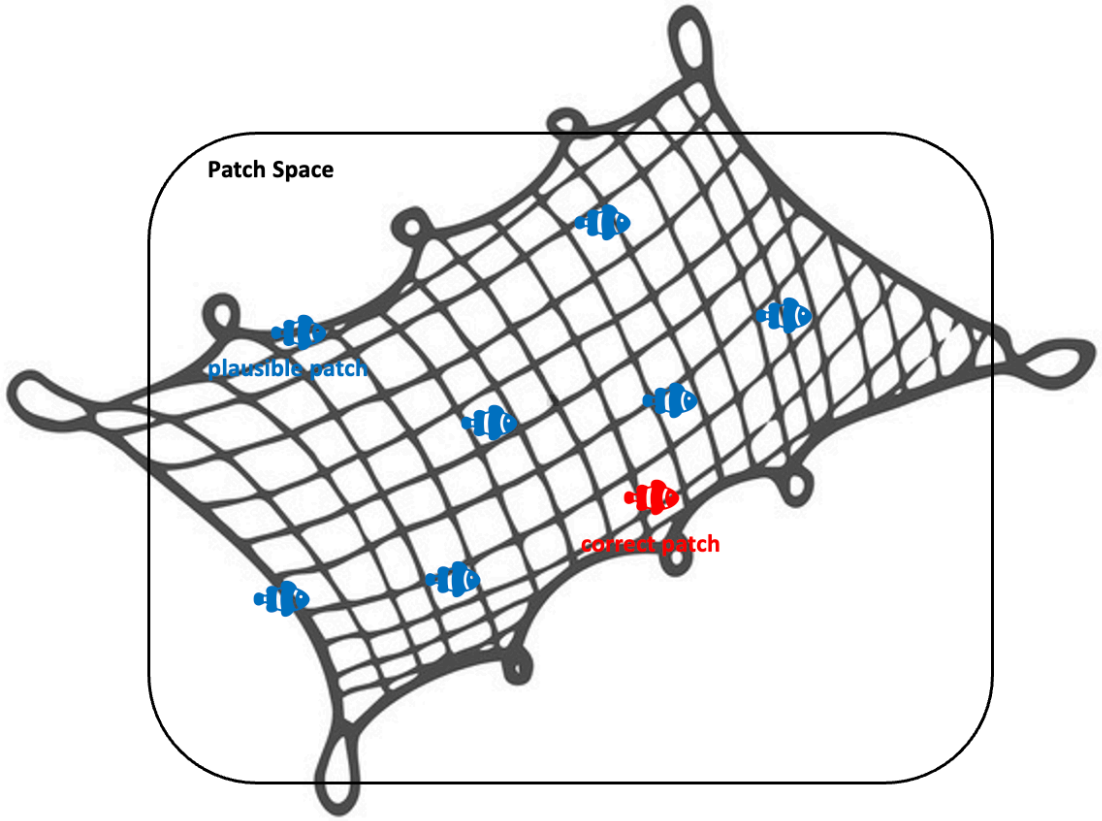
$b_\star$ index	branch ID	+/-	Distribution
0	ID_0	+	$Beta(\alpha_0, \beta_0)$
1	ID_1	+	$Beta(\alpha_1, \beta_1)$
2	ID_2	-	$Beta(\alpha_2, \beta_2)$
3	ID_3	+	$Beta(\alpha_3, \beta_3)$
4	ID_4	-	$Beta(\alpha_4, \beta_4)$

Evaluation (D4J v1.2; 10 times repetitions)

Results on Recalling Correct Patches

Top 1	Top 5
	

Part 3: Patch Verification



Cast a Wide Net

APR

≈

process of searching for plausible patches

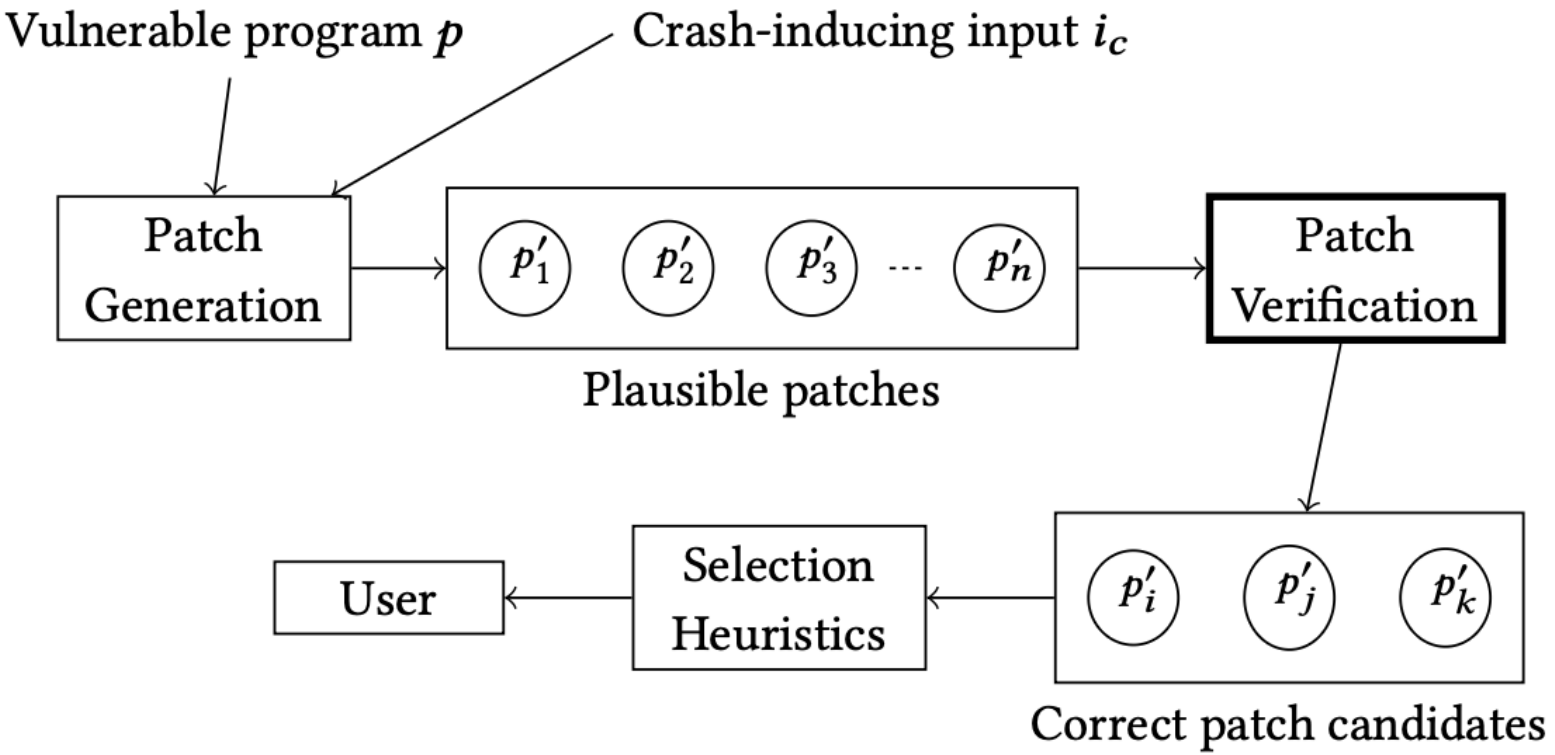
+

select a correct patch

SymRadar: PoC-Centered Bounded Verification for Vulnerability Repair

- ICSE 2026
  - Seungheon Han, YoungJae Kim, Yeseung Lee, Jooyong Yi

Automated Vulnerability Repair



Problem Statement

- Is a given function safe or vulnerable?

Existing Approaches

Static Analysis

Static Analysis

- Modern static analyzers are neither sound nor complete.
  - They may determine a vulnerable function as safe.
  - They may determine a safe function as vulnerable.

Deep Learning

Deep Learning

- Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection (ISSTA 2025)

Our Approach

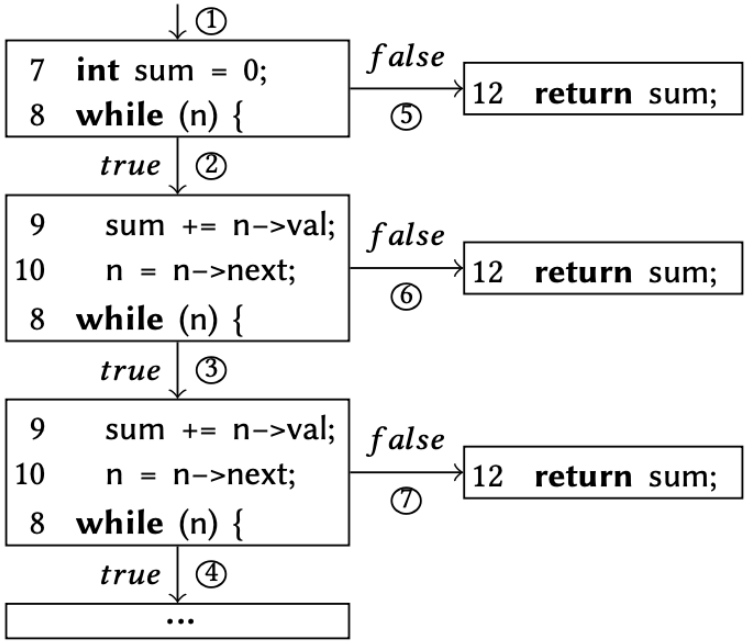
1. Bounded verification via symbolic execution
2. Function-level verification to avoid the reachability problem
3. PoC-centered verification

Under-Constrained Symbolic Execution

```
struct node {
  int data;
  struct node* next;
};

int listSum(struct node* n) {
  int sum = 0;
  while (n) {
    sum += n->data;
    n = n->next;
  }
  return sum;
}
```

Under-Constrained Symbolic Execution



Paths:

π_1: ① → ⑤

π_2: ① → ② → ⑥

π_3: ① → ② → ③ → ⑦

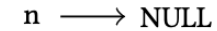
Path conditions:

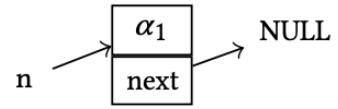
n = NULL

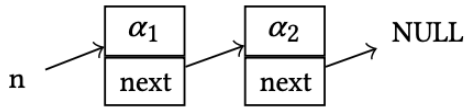
n ≠ NULL ∧
n = &node1 ∧
node1.next = NULL

n ≠ NULL ∧
n = &node1 ∧
node1.next ≠ NULL ∧
node1.next = &node2 ∧
node2.next = NULL

Symbolic inputs:



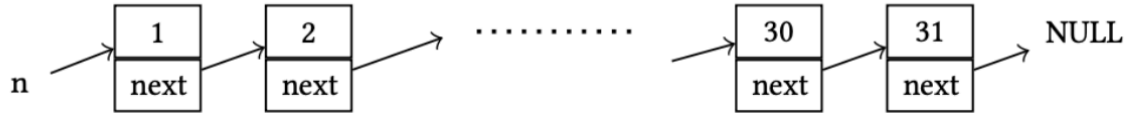




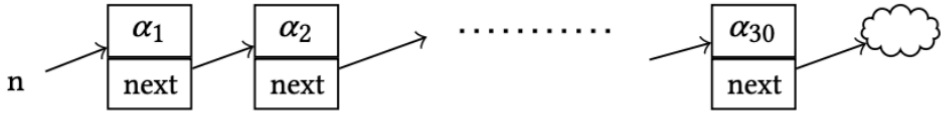
Limitation of UC-SE

```
int listSum(struct node* n) {
  int sum = 0;
  int i = 1; // added
  while (n) {
    sum += n->data;
    n = n->next;
    i *= 2; // added
  }
  g = arr[i]; // added
  return sum;
}
```

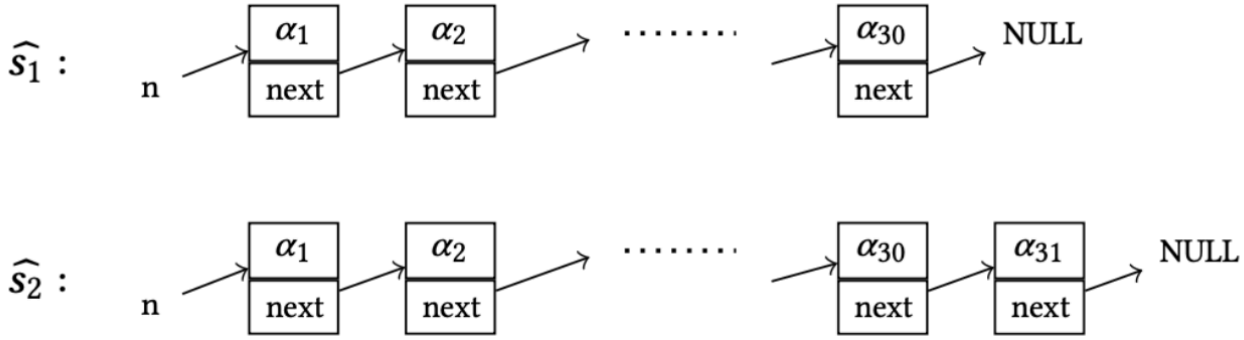
PoC-Centered Bounded Patch Verification



Step 1: Taking a concrete snapshot

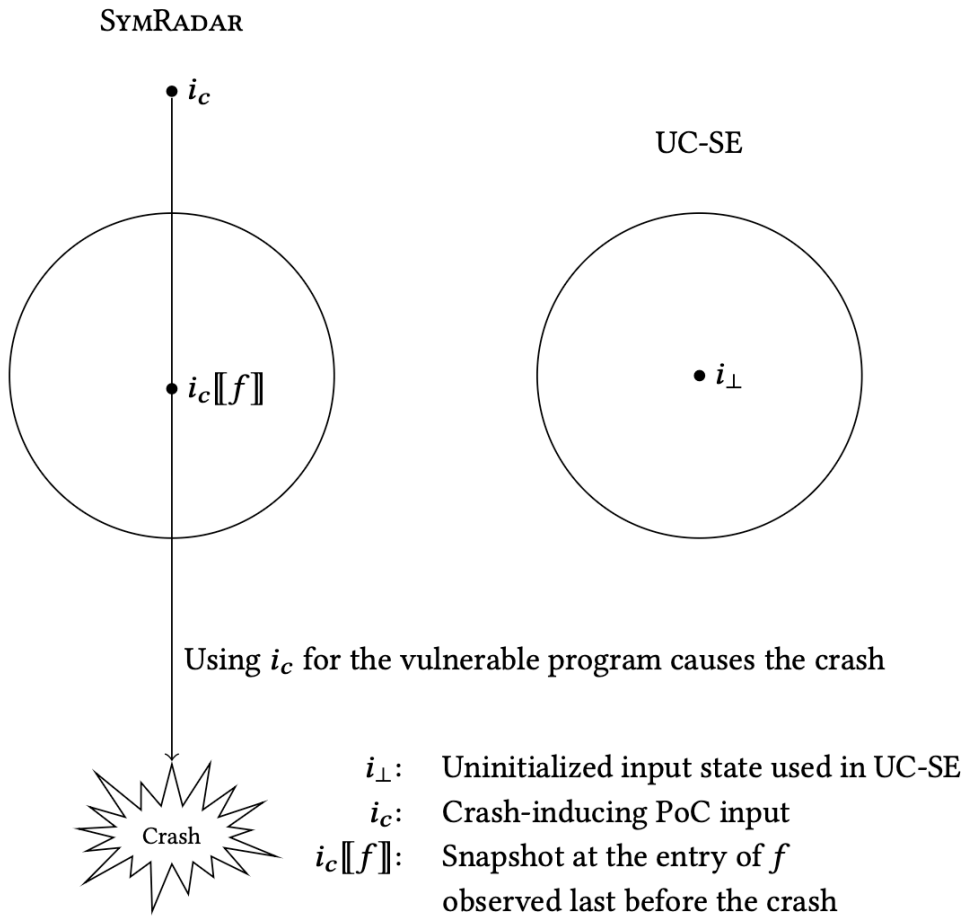


Step 2: Constructing the abstract snapshot

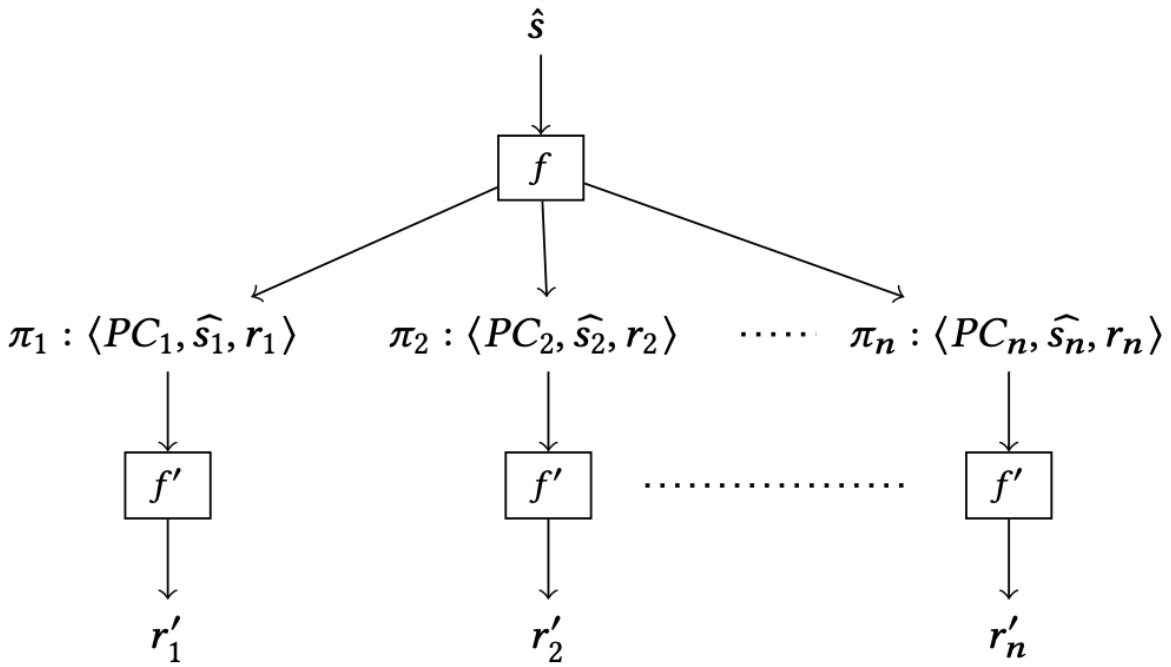


Step 3: Performing patch verification

UC-SE vs. SymRadar

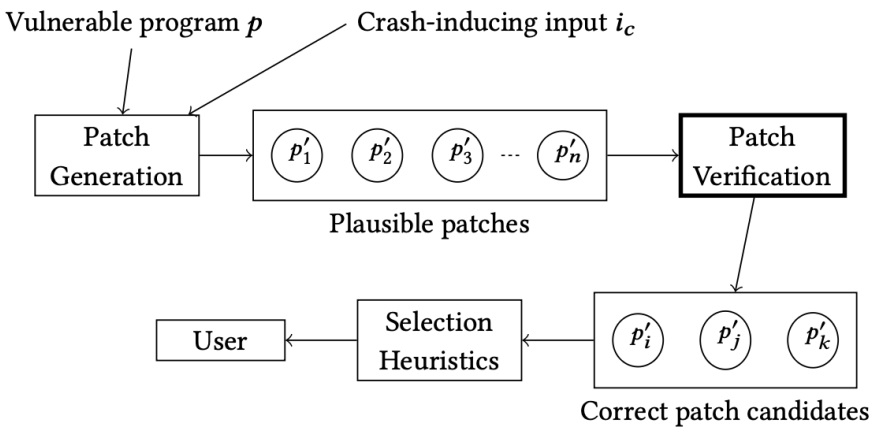


Patch Verification



Patch Classification Rubric

Case	r_i	r_i'	Condition	Classification
1	Crash	Normal	-	Safe
2	Crash C	Crash	$C = C_{PoC}$	Unsafe
3	Crash C	Crash	$C \neq C_{PoC}$	Safe
4	Normal	Crash	$C = C_{PoC}$	Unsafe
5	Normal	Crash	$C \neq C_{PoC}$	Safe
6	Normal	Normal	-	Check regression



Key Requirements for Patch Verification

1. Detecting as many incorrect plausible patches as possible (high specificity)
2. Preserving as many correct patches as possible (high recall)

Evaluation (3,3037 patches generated from CPR)

Tool	Recall	Specificity	Balanced Accuracy
SymRadar (Ours)	100%	78%	89%
CPR	96%	8%	52%
UC-KLEE	88%	57%	73%
Spider	77%	59%	67%
VulnFix	62%	66%	64%

Evaluation (90 patches generated from San2Patch)

Tool	Recall	Specificity	Balanced Accuracy
SymRadar (Ours)	100%	74%	87%
UC-KLEE	100%	52%	76%
Spider	48%	83%	65%
VulnFix	63%	28%	45%

Is Our Heuristic Effective?

Case	r_i	r_i'	Condition	Classification
1	Crash	Normal	-	Safe
2	Crash C	Crash	$C = C_{PoC}$	Unsafe
3	Crash C	Crash	$C \neq C_{PoC}$	Safe
4	Normal	Crash	$C = C_{PoC}$	Unsafe
5	Normal	Crash	$C \neq C_{PoC}$	Safe
6	Normal	Normal	-	Check regression

Is Our Heuristic Effective?

Heuristic	Recall	Specificity	Balanced Accuracy
Yes	100%	78%	89%
No	73%	80%	76%

Runtime Performance

- SymRadar detects
  - 90% of incorrect patches within 3 minutes.
  - 95% of incorrect patches within 2.3 hours.

Authors

Part 1: How do we hunt bugs?

Sehoon Kim
Yonghyeon Kim
Dahyeon Park
Yuseok Jeon
Jooyong Yi
Mijung Kim

Part 2: How do we hunt patches?

YoungJae Kim
Seungheon Han
Askar Yeltayuly Khamit
Yechan Park
Jooyong Yi

Part 3: How do we verify patches?

Seungheon Han
YoungJae Kim
Yeseung Lee
Jooyong Yi