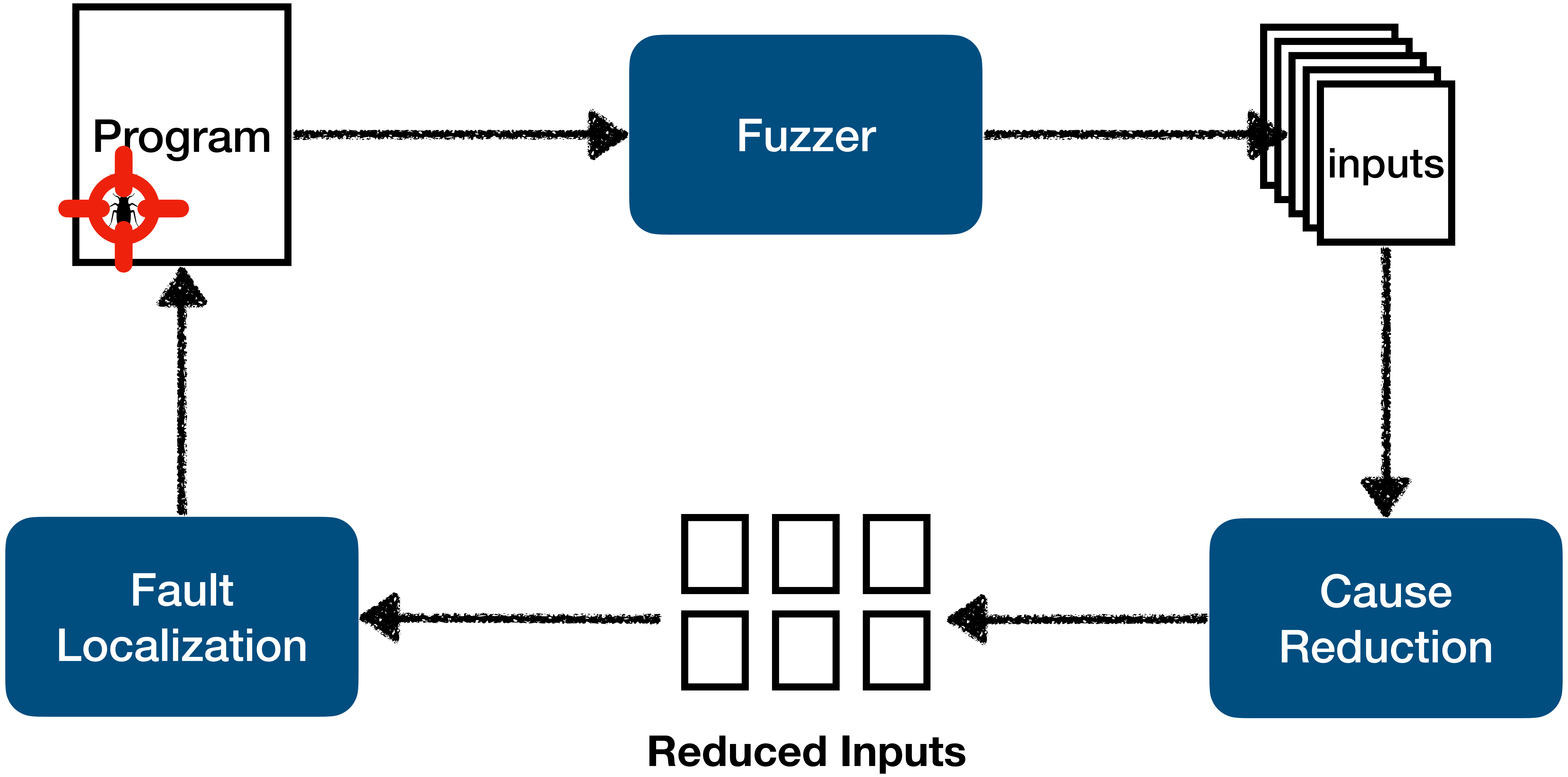
IS893: Advanced Software Security

5. Fault Localization

Kihong Heo

KAIST

Overview


Program
Fuzzer
inputs
Cause
Reduction
Reduced Inputs
Fault
Localization

Fault Localization

- Given 1+ failing inputs and 0+ passing inputs, find a list of suspicious locations
- Two dominant approaches:
  - Statistical: “executing faulty statements is likely to lead to a failure”
  - Logical: “minimal sub-formula making a safety condition UNSAT”

```
1: x = input(); // read “-1”
2: if (x < 0) {
3:   y = x + 1;
4:   z = x + 2;
5: } else {
6:   y = x + 2;
7:   z = x + 2;
8: }
9: k = y;
10: r = z / k;    // crash
```

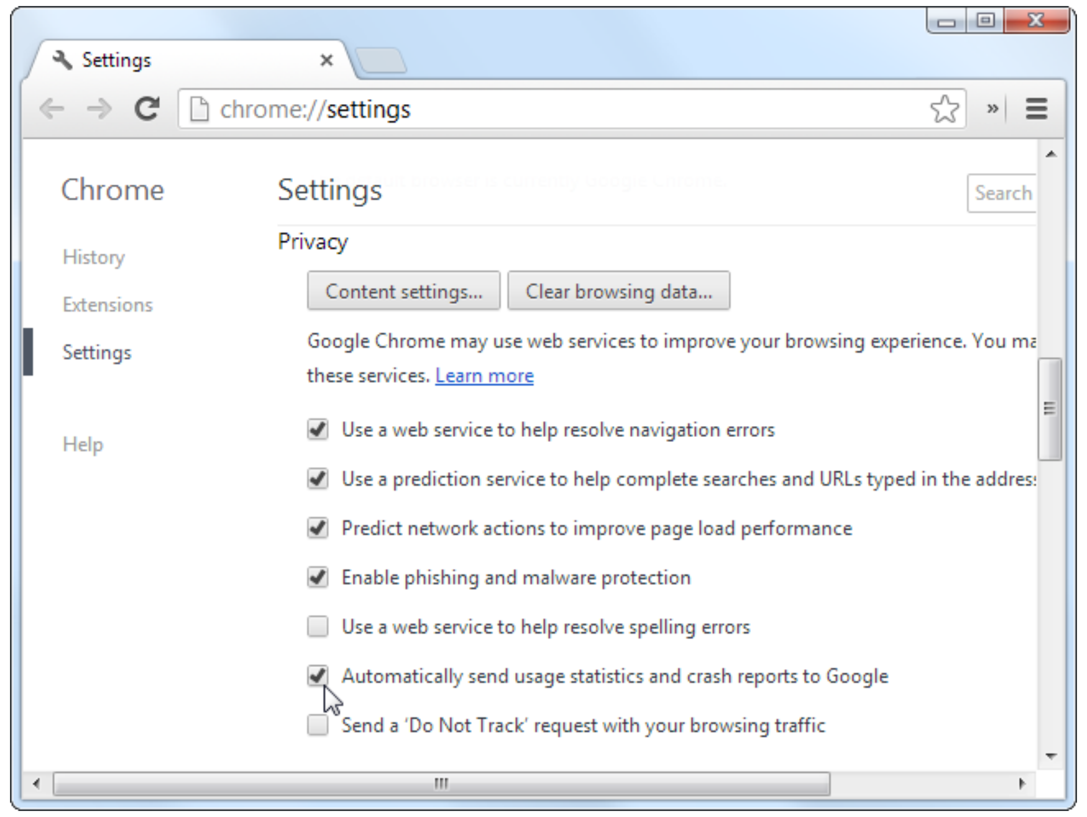
$$x = -1 \land x < 0 \land y_1 = x + 1 \land z_1 = x + 2 \land (x < 0\ ?\ k = y_1 : k = y_2) \land k \neq 0 : \text{UNSAT}$$

$$\Downarrow$$

$$\underbrace{x = -1}_{\text{input}} \land \underbrace{y_1 = x + 1}_{\text{line 3}} \land \underbrace{(x < 0\ ?\ k = y_1 : k = y_2)}_{\text{line 2 \& 9}} \land \underbrace{k \neq 0}_{\text{assertion}} : \text{Minimal UNSAT Core}$$

Statistical Fault Localization

- Statistically analyze correlations between certain behaviors and test results
  - Behavior: code coverage, branch conditions, return values, etc
  - So called “statistical debugging”
  - Enabled by a large amount of test runs from fuzzers, crowdsource, etc



Steps

1. **Instrument** the target program to capture “interesting” behaviors

2. **Observe** the behaviors and results for each test run

3. **Analyze** behavioral changes in successful vs. failing runs

Interesting Behavior

- Assume that any interesting behavior is represented as a predicate **P**
  - On a program state at a particular program point

Branch Conditions

```
observe(p != 0);
if (p) {
  …
} else {
  …
}
```

Return Values

```
fd = fopen(…);
observe(fd > 0);
```

Scala Relationships

```
observe(i < j);
…
observe(i > 42);
```

Pointer Relationships

```
observe(p == q);
…
observe(p != null);
```

… and many others depending on the problem

Statistical Reasoning

- A large amount of information about many predates in a program
  - E.g., 300K predicates for UNIX bc with 30K test runs*
- Which predicate is more relevant to bugs?
  - Most of them are not predictive of anything

*Liblit et al., Scalable Statistical Bug Isolation, *PLDI*, 2005

Measures

- How likely is failure when predicate **P** is observed to be true?

F(P) = # failing runs where P is observed to be true
S(P) = # successful runs where P is observed to be true

$$\text{Failure(P)} = Pr(\text{Crash} \mid \text{P observed to be true}) = \frac{F(P)}{F(P) + S(P)}$$

Hypothesis: **P** is suspicious if Failure(**P**) is high

```
f = …;
if (f == NULL) {
  *f;
}
```

Problem

- "Correlation is not causation"

F(P) = # failing runs where P is observed to be true
S(P) = # successful runs where P is observed to be true

```
f = …;
if (f == NULL) {
  x = 0;
  *f;
}
```

Failure(f == NULL) = 1.0
Failure(x == 0) = 1.0

Fact: High Failure(**P**) does not mean **P** is the cause of a bug

More Context

- Intuition:
  If **P** is correlated with the failure, regardless of the truth value, then less important

- How likely is failure when predicate **P** is observed?
  (not necessarily its truth value)

F(P observed) = # failing runs where P is observed ~~to be true~~ S(P observed) = # successful runs where P is observed ~~to be true~~

$$\text{Context(P)} = \mathit{Pr}(\text{Crash} \mid \mathbf{P} \text{ observed } \cancel{\text{to be true}}) = \frac{\text{F}(\mathbf{P} \text{ observed})}{\text{F}(\mathbf{P} \text{ observed}) + \text{S}(\mathbf{P} \text{ observed})}$$

New Measure

- How much does **P** being true increase the probability of failure over simply reaching the line where **P** is sampled?

Increase(P) = Failure(P) - Context(P)

```
f = …;
if (f == NULL) {
  x = 0;
  *f;
}
```

// Suppose 1 failing run and 2 passing runs

Failure(f == NULL) = 1.0
Context(f == NULL) = 0.33
Increase(f == NULL) = 0.67

Failure(x == 0) = 1.0
Context(x == 0) = 1.0
Increase(x == 0) = 0.0

"Increase(P) = 1" means high correlation with failing runs
"Increase(P) = 0" means P is an invariant (true for all runs)
"Increase(P) = -1" means high correlation with successful runs

Example

```
void main() {
  int z;
  for (int i = 0; i < 3; i++) {
    char c = getc();
    if (c == 'a')
      z = 0;
    else
      z = 1;
    assert(z == 1);
  }
}
```

Inputs: {"bba", "bbb"}

	Increase
c == 'a'	0.5
c != 'a'	0.0
i < 3	0.0
i >= 3	-0.5

Simple Algorithm

- Discard predicates having Increase(P) <= 0
  - E.g., bystander predicates, predicates correlated with success
- Sort remaining predicates by Increase(P)

Real World Example: bc

```
void more_array() {
  …
  /* Copy the old arrays. */
  for (indx = 1; index < old_count; indx++)
    arrays[indx] = old_ary[indx];
  /* Initialize the new elements. */
  for (; index < v_count; indx++)
    arrays[indx] = NULL;
  …
}
```

Interesting behavior: scala relationship

	Increase
1	indx > scale
2	indx > use_math
3	indx > opterr
4	indx > next_func
5	indx > i_base
…	…

v_count is a wrong bound

Problem: Multiple Bugs

- Real programs often have multiple unknown bugs
- The effects of bugs may be interrelated
- High Increase values are still good indicators for debugging?
  - Maybe not, because of redundancies
  - E.g., predicates of common bugs may dominates

New Goal

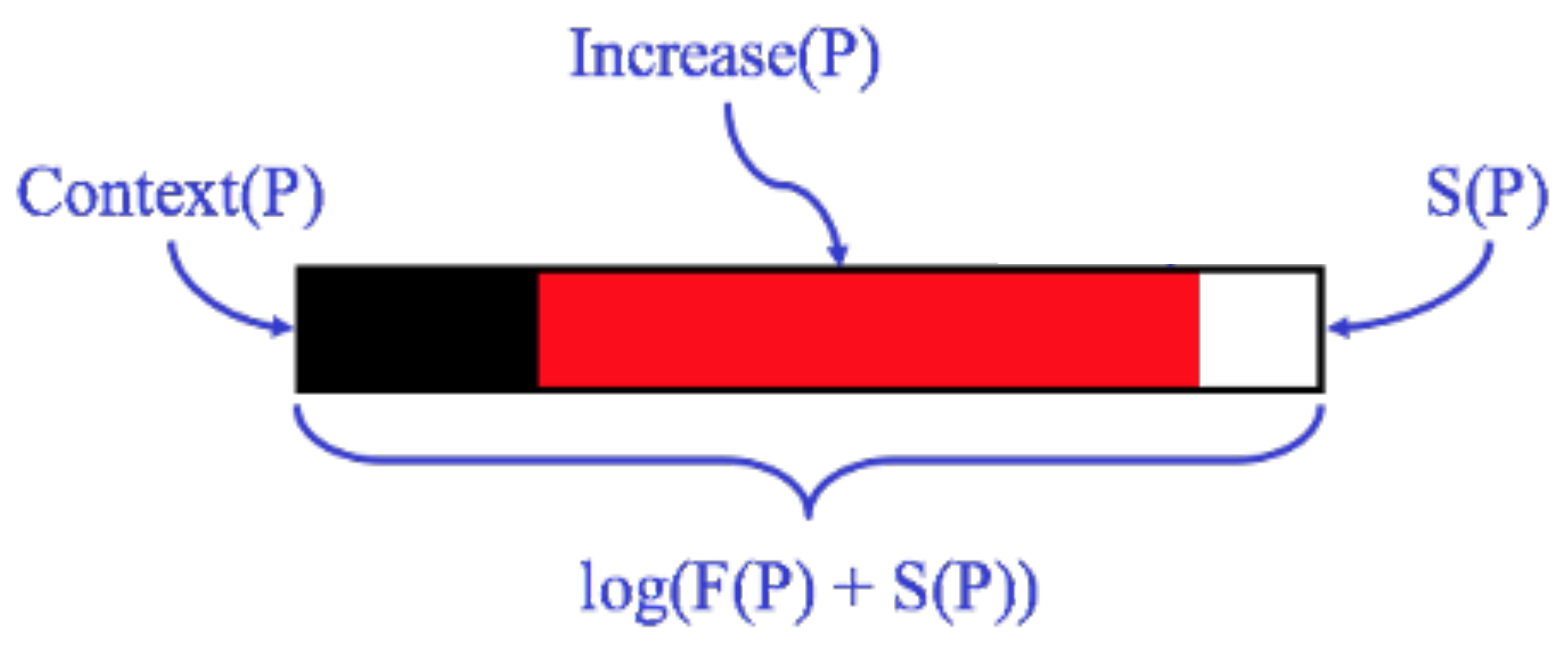
- Find the best predictor for each bug w/o prior knowledge of #bugs
  - Enable us to focus on the bug
- Sorted by the importance of the bugs
  - Enable us to prioritize efforts to the bugs that affect the most users
- How? Intuition: simulate the way humans fix bugs
  - Find the most important bug; fix it; and repeat

New Algorithm

1. Compute various measures (e.g., Increase, F, etc) and rank the predicates
2. Pick the top-ranked predicate **P**
3. Discard all runs where **P** is true:
    - simulate fixing the bug corresponding to **P**
    - reduces rank of correlated predicates:
      (predicators of other bugs will rise)
4. Goto 1

Measures

- Is Increase(P) still a good measure?
- Let us organize all measures in compact visual: bug thermometer



Long bar: the predicate was observed many times

Importance

- Ranking by Increase(P)

Increase(P) = Failure(P) - Context(P)
Failure(P) = *Pr*(Crash | P observed to be true)
Context(P) = *Pr*(Crash | P observed)

(Report from >5000 failing runs)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.065	0.935 ± 0.019	0	23	23	`((*(fi + i)))->this.last_token < filesbase`
	0.065	0.935 ± 0.020	0	10	10	`((*(fi + i)))->other.last_line == last`
	0.071	0.929 ± 0.020	0	18	18	`((*(fi + i)))->other.last_line == filesbase`
	0.073	0.927 ± 0.020	0	10	10	`((*(fi + i)))->other.last_line == yy_n_chars`
	0.071	0.929 ± 0.028	0	19	19	`bytes <= filesbase`
	0.075	0.925 ± 0.022	0	14	14	`((*(fi + i)))->other.first_line == 2`
	0.076	0.924 ± 0.022	0	12	12	`((*(fi + i)))->this.first_line < nid`
	0.077	0.923 ± 0.023	0	10	10	`((*(fi + i)))->other.last_line == yy_init`
.... 2732 additional predictors follow						

High Increase(P) scores tend to report predicates with high Failure and low Context
=> if observed to be true, very likely crash, but mostly observed to be false
=> indicate very special cases of more general bugs

Importance

Increase(P) = Failure(P) - Context(P)
Failure(P) = *Pr*(Crash | P observed to be true)
Context(P) = *Pr*(Crash | P observed)

- Ranking by F(P)

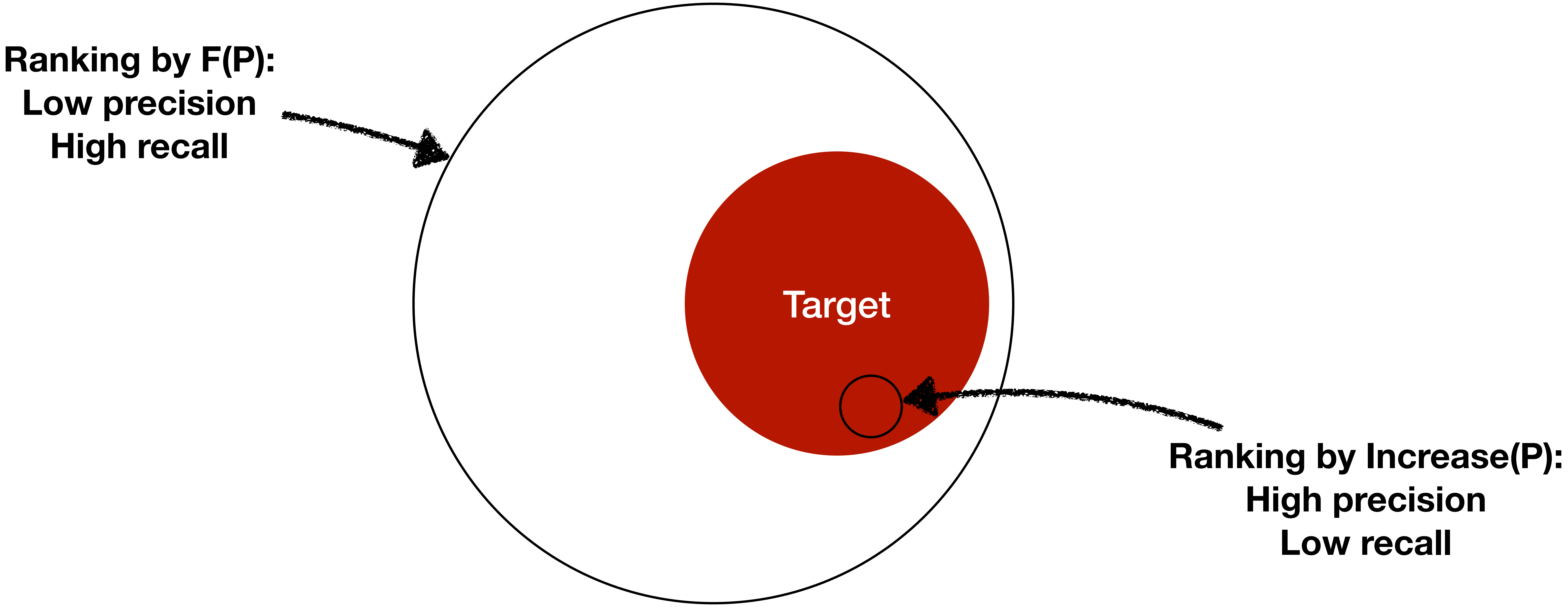
(Report from >5000 failing runs)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.007 ± 0.012	22554	5045	27599	`files[filesindex].language != 15`
	0.176	0.007 ± 0.012	22566	5045	27611	`tmp == 0 is FALSE`
	0.176	0.007 ± 0.012	22571	5045	27616	`strcmp != 0`
	0.176	0.007 ± 0.013	18894	4251	23145	`tmp == 0 is FALSE`
	0.176	0.007 ± 0.013	18885	4240	23125	`files[filesindex].language != 14`
	0.176	0.008 ± 0.013	17757	4007	21764	`filesindex >= 25`
	0.177	0.008 ± 0.014	16453	3731	20184	`new value of M < old value of M`
	0.176	0.261 ± 0.023	4800	3716	8516	`config.winnowing_window_size != argc`

. . . . 2732 additional predictors follow . . . .

High F(P) scores tend to report predicates with low Failure and high Context
=> if observed to be true, more likely pass
=> indicate very general predicates covering many different bugs as well as correct behaviors

Analogy


Ranking by F(P):
Low precision
High recall
Target
Ranking by Increase(P):
High precision
Low recall

Harmonic Mean

- Standard solution to achieve both high precision and high recall

$$\text{Importance}(P) = \frac{2}{1/\text{Increase}(P) + 1/F(P)}$$

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.824 ± 0.009	0	1585	1585	`files[filesindex].language > 16`
	0.176	0.824 ± 0.009	0	1584	1584	`strcmp > 0`
	0.176	0.824 ± 0.009	0	1580	1580	`strcmp == 0`
	0.176	0.824 ± 0.009	0	1577	1577	`files[filesindex].language == 17`
	0.176	0.824 ± 0.009	0	1576	1576	`tmp == 0 is TRUE`
	0.176	0.824 ± 0.009	0	1573	1573	`strcmp > 0`
	0.116	0.883 ± 0.012	1	774	775	`((*(fi + i)))->this.last_line == 1`
	0.116	0.883 ± 0.012	1	776	777	`((*(fi + i)))->other.last_line == yyleng`
……… 2732 additional predictors follow ………						

Conclusion

- Which part of the program is wrong? Fault localization!
- A common approach: statistically localize root causes of bugs
    - Collect data by instrumentation and crash reports (fuzzing, crowd, etc)
    - Many metrics to rank “interesting behaviors”
- Applications: guiding manual debugging or automated program repair tools