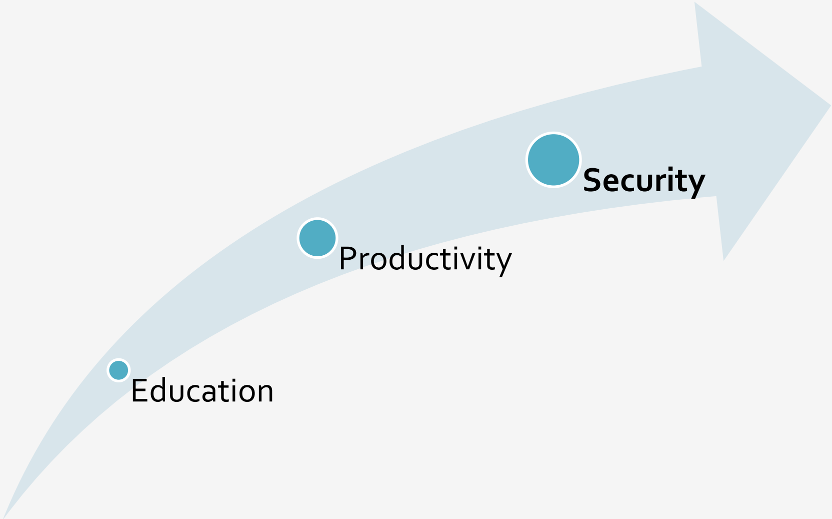
# Overfitting in Program Repair

Abhik Roychoudhury

Professor, National University of Singapore

Director, National Satellite of Excellence in Trustworthy Software, Singapore

# Why?



Education

Productivity

Security

- Maintaining Legacy Software
- Debugging Aid
- Education, Grading in MooCs
- Security Patches
- Self-healing systems, Drones

**Automated Program Repair**
Claire Le Goues, Michael Pradel, Abhik Roychoudhury
Communications of the ACM (CACM),
62(12), December 2019.

# Program Correctness

- “Behind every large program there is a small program waiting to get out”
- C.A.R. Tony Hoare

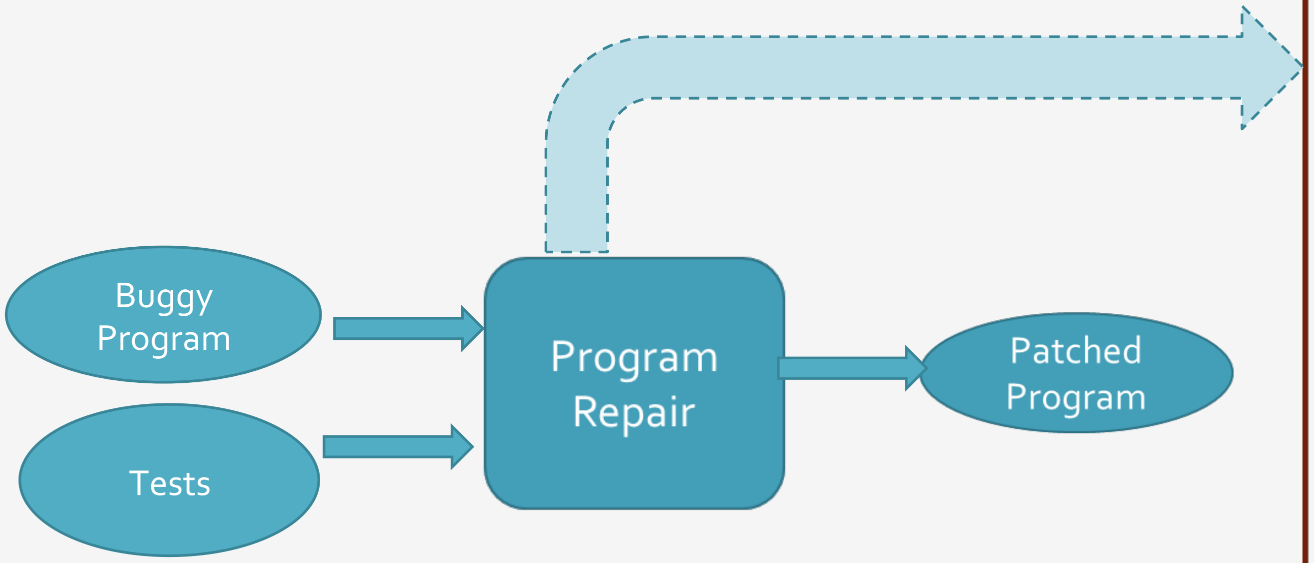
- Behind every large program there is an algorithm waiting to get out
- “Free your mind with mathematics”
- Leslie Lamport

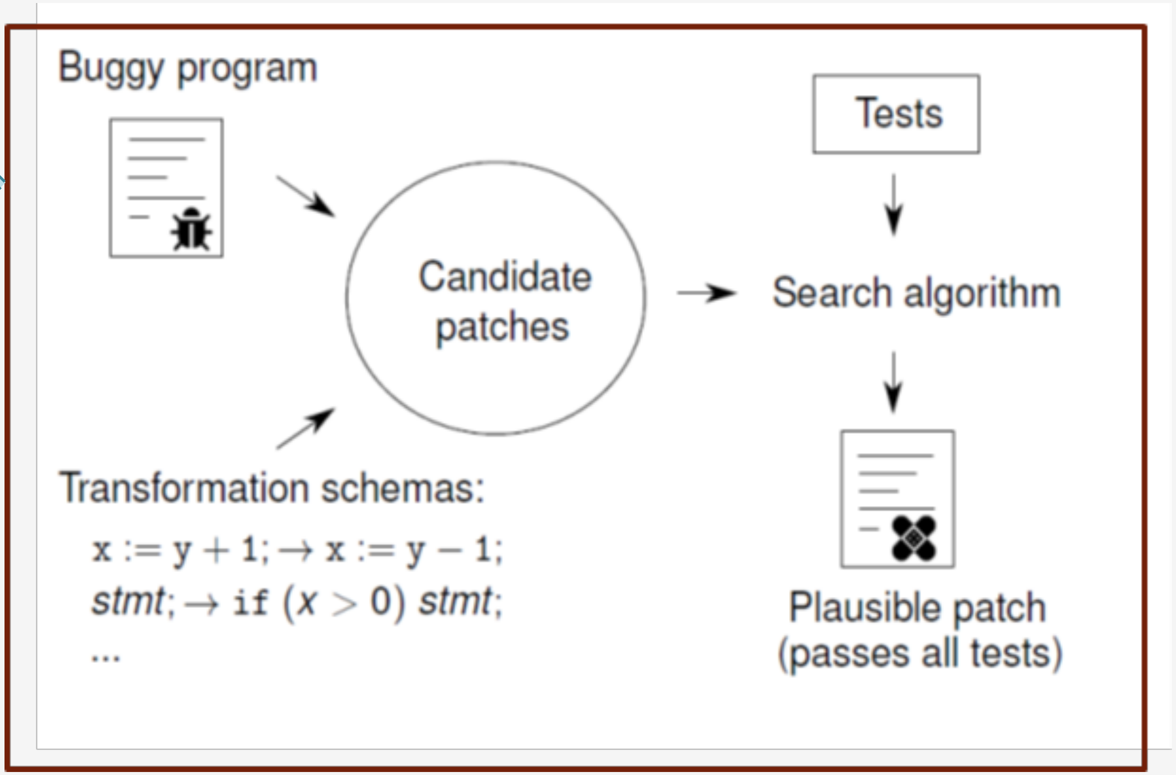
# Formal Specification

- Manual Formal Specification
  - Specify the requirements
- Challenges:
  - Developer education (non CS backgrounds)
  - Developer reluctance
  - Third party code and non monolithic assembly

- Automated Specification inference
  - Only what is ”wrong” in the program
- Challenges:
  - Requires specifying what is correct
  - Limited notions of correctness available e.g. Tests
  - Generalizes beyond tests but subject to **overfitting**.

# Automated Program Repair





Can we generate a program

```
if input1 return output1
else if input2 return output2
else …
```

**Generate and Validate**

# The “right” patch

| Test id | a | b | c | oracle | Pass |
|---|---|---|---|---|---|
| 1 | -1 | -1 | -1 | INVALID | ✓ |
| 2 | 1 | 1 | 1 | EQUILATERAL | ✓ |
| 3 | 2 | 2 | 3 | ISOSCELES | ✓ |
| 4 | 2 | 3 | 2 | ISOSCELES | ✗ |
| 5 | 3 | 2 | 2 | ISOSCELES | ✗ |
| 6 | 2 | 3 | 4 | SCALANE | ✗ |

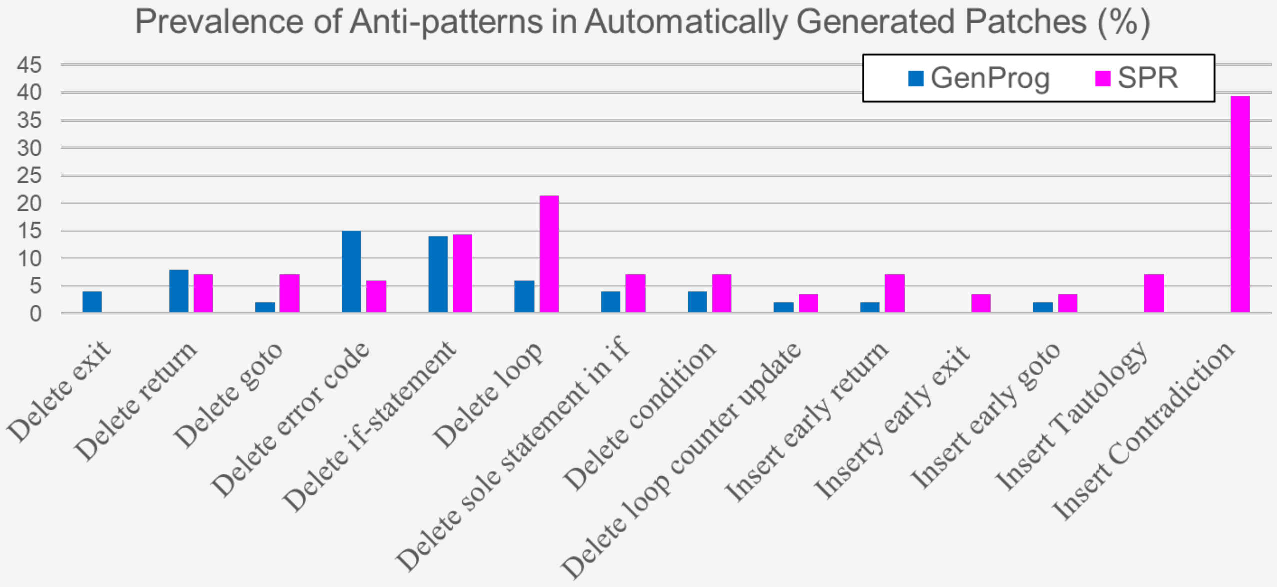
```
1 int triangle(int a, int b, int c){
2     if (a <= 0 || b <= 0 || c <= 0)
3        return INVALID;
4     if (a == b && b == c)
5        return EQUILATERAL;
6     if (a == b || b != c) // bug!
7        return ISOSCELES;
8 return SCALENE;
9 }
```

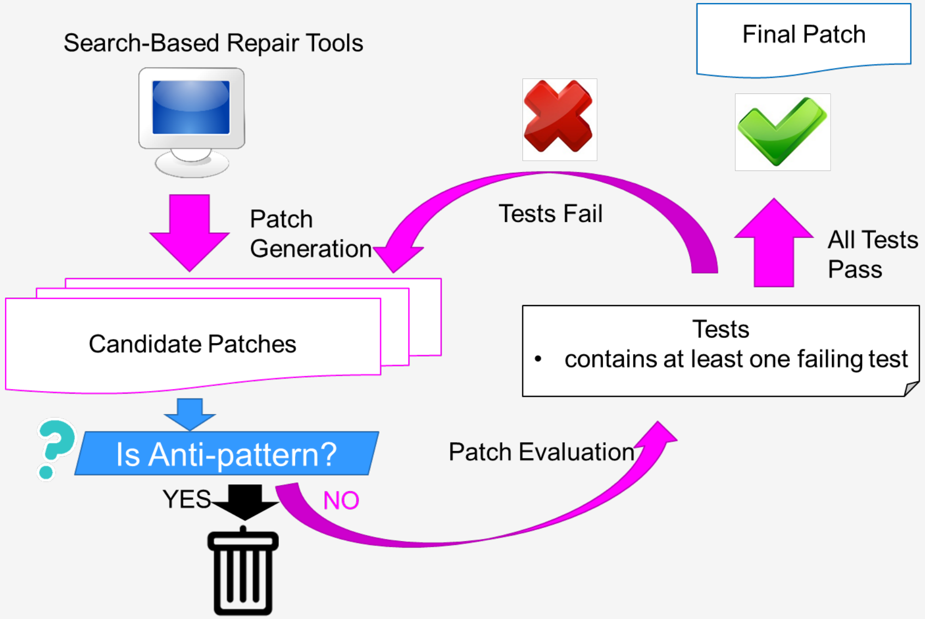
Correct fix
(a == b || b== c || a == c)

Traverse all mutations of line 6 ??

Hard to generate fix since (a ==c) or (c ==a) never appear anywhere else in the program !

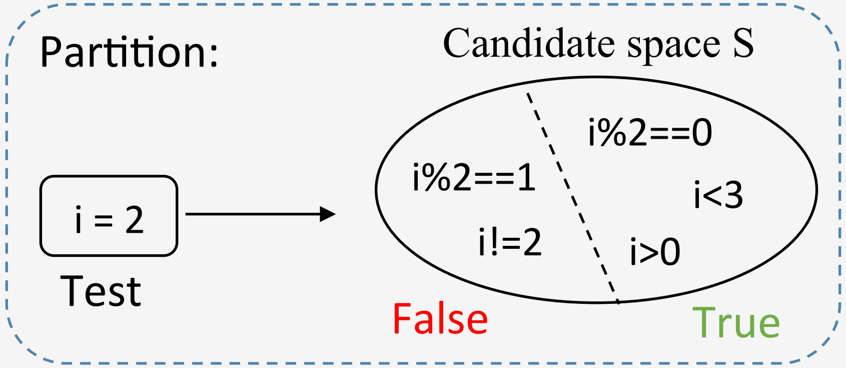
# 1. Combat Overfitting: Antipatterns





# Generate and Validate over Partitions

Test-equivalence Analysis for Automatic Patch Generation, Mechtaev et al. TOSEM, 2018



```
for candidate c ∈$\mathcal{S}$ do
    validate(c)
end
```

(a) Enumerative approach

```
for partition p ∈$\mathcal{S}$ do
    validate(p)
end
```

(b) Test-equivalent partitioning

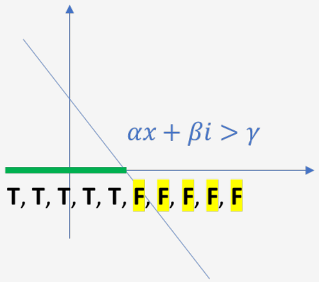
The patch candidates can be evaluated more efficiently.

# Test Equivalence on Patches

```
scanf ("%d" ,&x);
for (i = 0; i < 10; i++)
       if (x - i > 0)
              printf ("1");
       else
              printf ("0");
```

Consider all inequalities $\alpha x\ [\pm]\ \beta i\ [>\geq=\neq]\ \gamma$

| Sequence of values: | Equivalence class (x = 4): |
|---|---|
| {T, T, T, T, T, T, T, T, T, T} | {x > 0, …} |
| {T, T, T, T, T, T, T, T, T, F} | {x – i > -5, …} |
| {T, T, T, T, T, T, T, T, F, T} | EMPTY |
| {T, T, T, T, T, T, T, T, F, F} | {x – i > -4, …} |
| {T, T, T, T, T, T, T, F, T, T} | EMPTY |
| {T, T, T, T, T, T, T, F, T, F} | EMPTY |
| {T, T, T, T, T, T, T, F, F, T} | EMPTY |
| … | |



```
if (tif->tif_rawcc > 0 && tif->tif_rawcc !=orig_rawcc
    && (tif->tif_flags & TIFF_BEENWRITING) != 0
    && ! TIFFFlushData1 ( tif )) {
      TIFFErrorExt (tif -> tif_clientdata, module,
        "Error flushing data before directory write");
  return (0);
}
```

112487 modifications

```
(( tif -> tif_rawcc > 0) && (tif -> tif_rawcc != orig_rawcc ))
  || (tif -> tif_flags & TIFF_BEENWRITING))
```

```
(( tif -> tif_rawcc > 0) || (tif -> tif_rawcc != orig_rawcc ))
  && (tif -> tif_flags & TIFF_BEENWRITING))
```

```
(( tif -> tif_rawcc == 0) && (tif -> tif_rawcc != orig_rawcc ))
  && (tif -> tif_flags & TIFF_BEENWRITING))
```

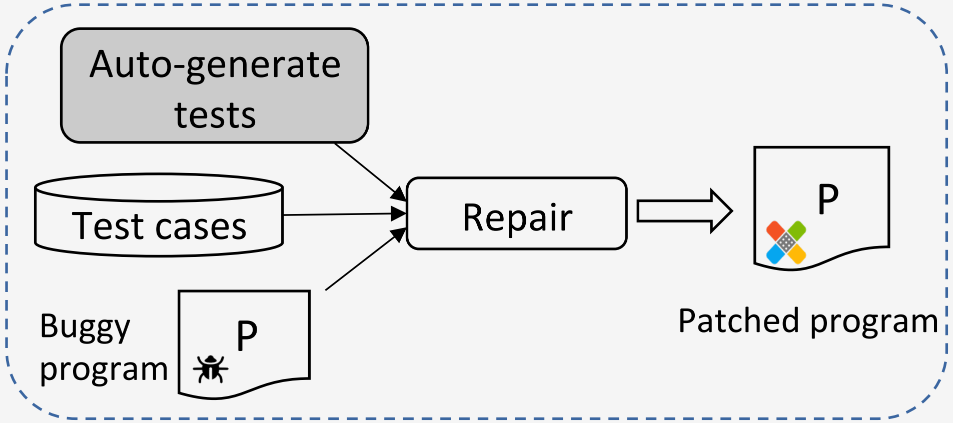
```
(( tif -> tif_rawcc > 0) && (tif -> tif_rawcc != orig_rawcc ))
  && (tif -> tif_flags & TIFF_BEENWRITING)) || (imagedone >= orig_rawcc)
```

```
(( tif -> tif_rawcc > 0) && (tif -> tif_rawcc != orig_rawcc ))
  && (tif -> tif_flags & TIFF_BEENWRITING)) || (tif->tif_flags >= 74)
```

5 eq. classes

# 2. Combat Overfitting: Fuzz Testing

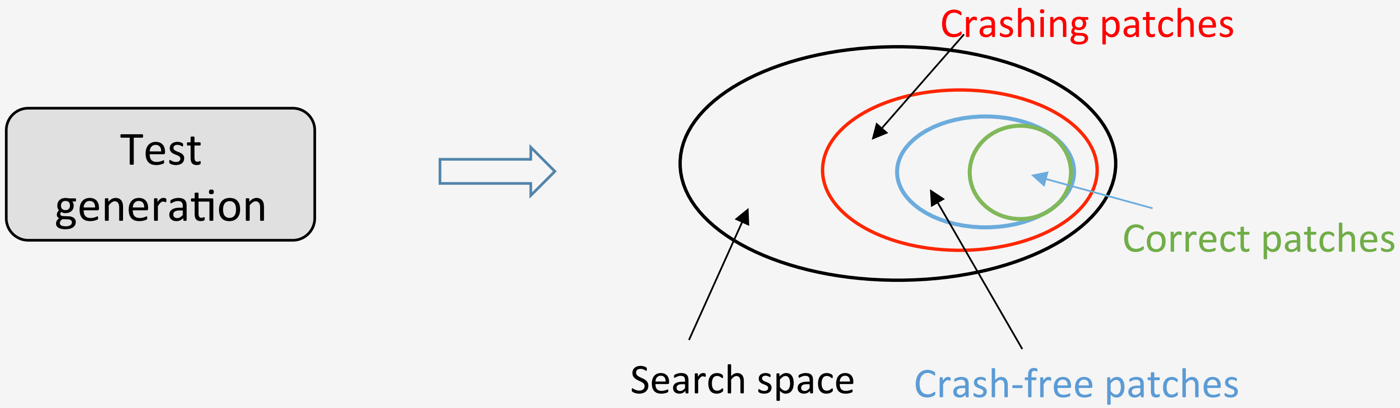
The given test suite can be enhanced by test generation (random, evolutionary algorithm and etc.).



Problems:

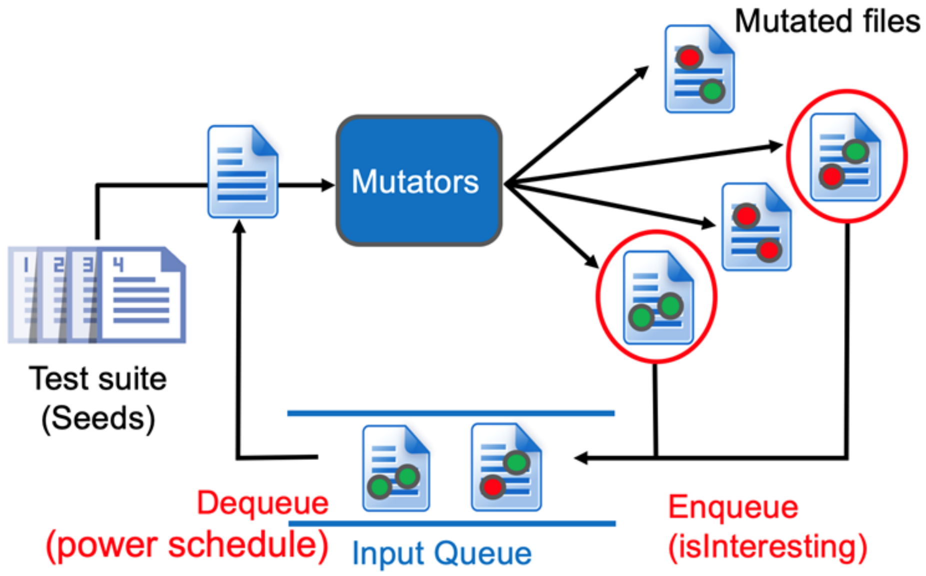
1. The oracles of newly generated tests are usually unknown.
2. Test generation for program repair is inefficient because it has no knowledge about patch candidates.

# Test generation to alleviate over-fitting

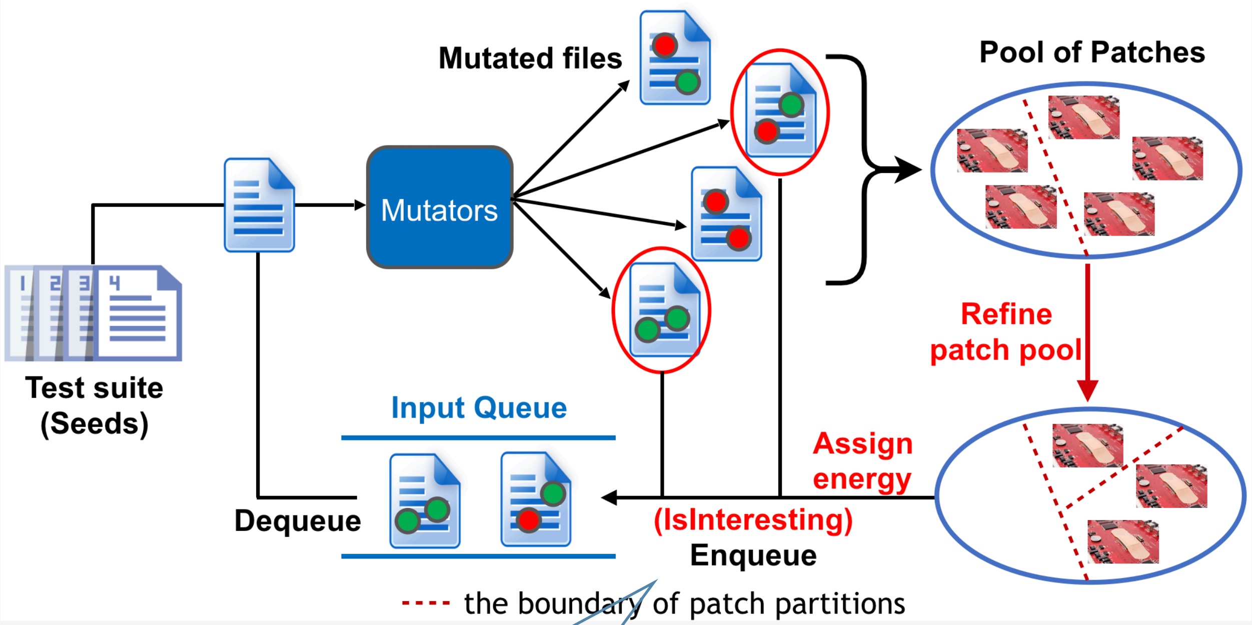


Distinguish crashing and crash-free patches (practical)

Crashing patches may (1) partially fix the crash or (2) unexpectedly introduce new crash



# Crash-avoiding Program Repair



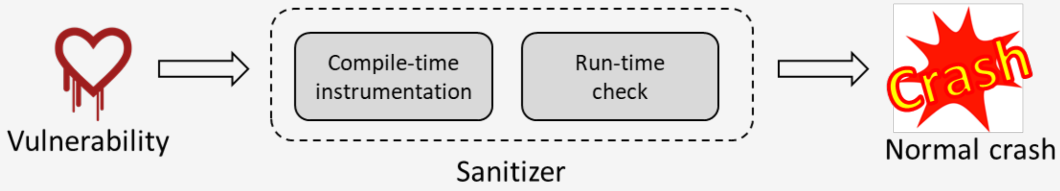
Retain inputs with non-zero separability

*Separability* formulates the ability of a test to find semantic discrepancies between plausible patches (break equivalent partition).

Patches are generated with the objective of passing existing tests.

New tests are generated with the objective of breaking equivalent partitions.

# Default Oracles and Additional Oracles



Vulnerability

Compile-time instrumentation

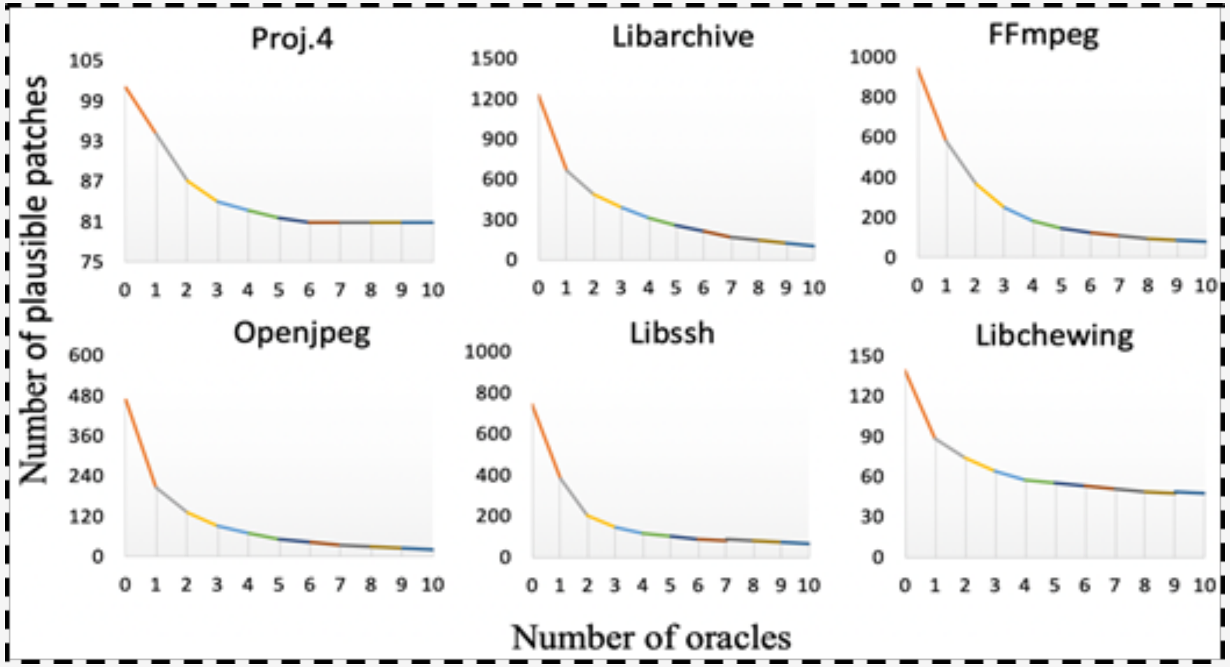
Run-time check

Sanitizer

Normal crash

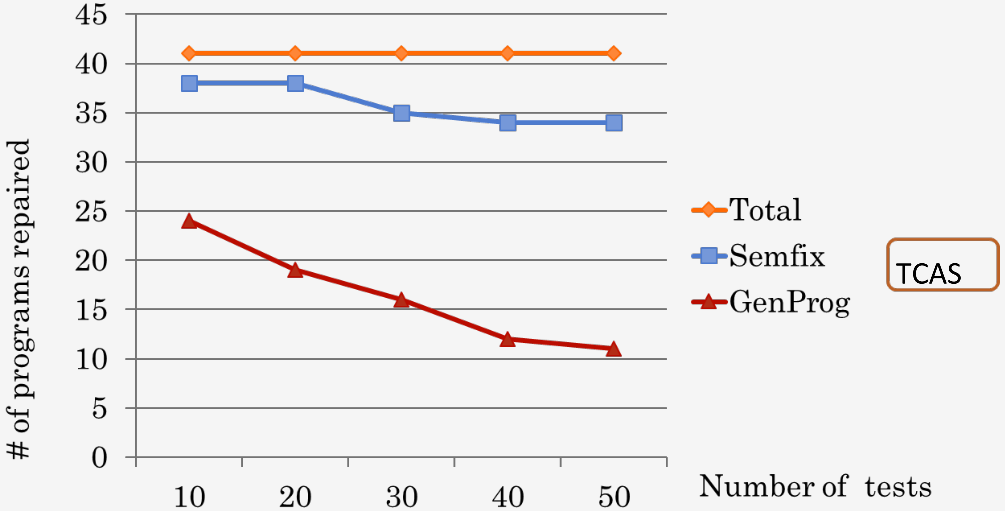
UndefinedBehaviorSanitizer: null-pointer, integer overflow and so on
AddressSanitizer: buffer/stack overflow, memory leak, use-after-free…

Patches are not only checked for crashes, but are also checked against the sanitizers.

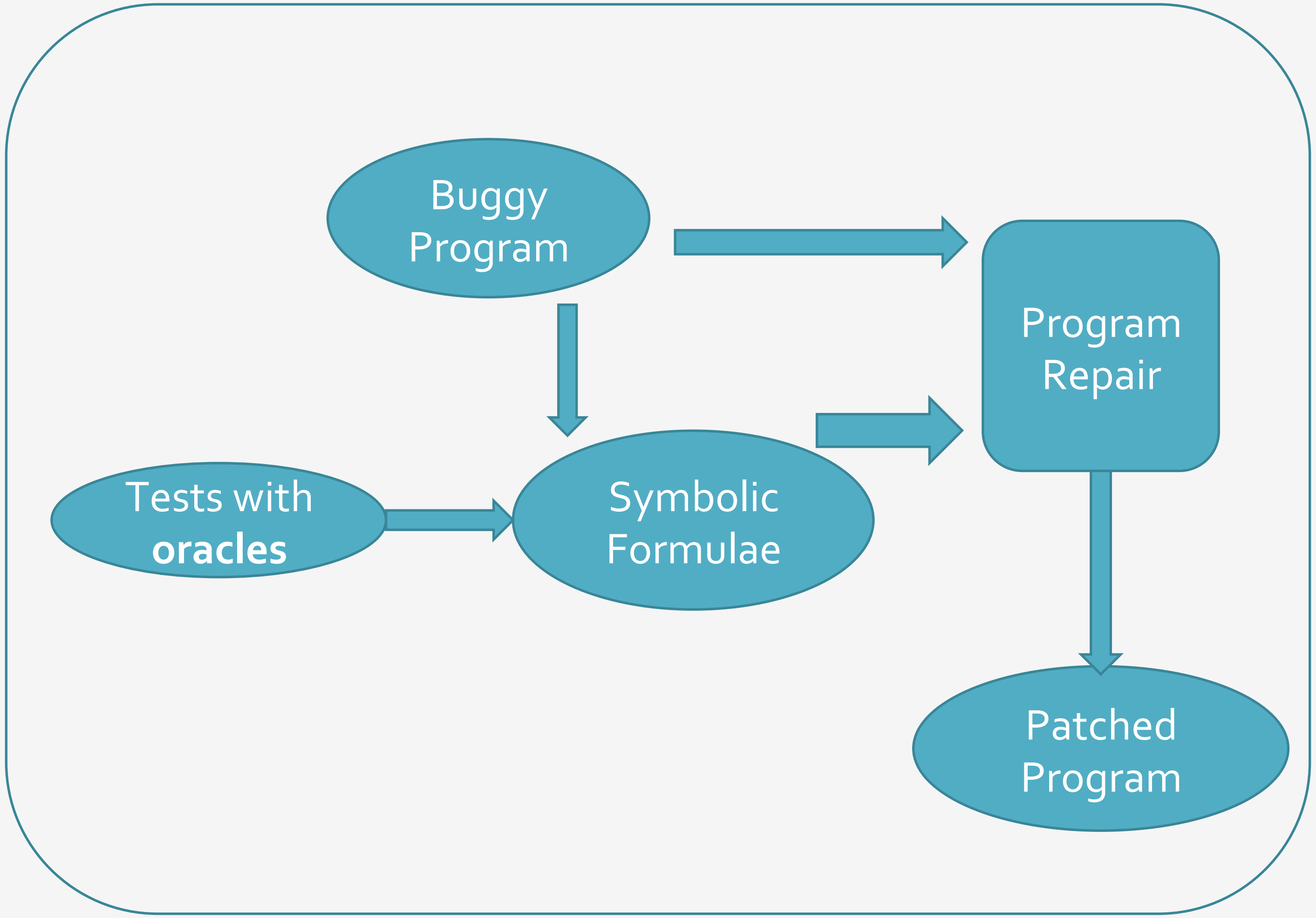


# 3. Combat Over-fitting: Spec. Inference

***Very old result on small programs!***



Overall 90 programs from SIR
SemFix repaired 48/90, GenProg repaired 16/90 for 50 tests.
*GenProg running time is >3 times of SemFix*



# Example

```
int triangle(int a, int b, int c){
    if (a <= 0 || b <= 0 || c <= 0)
        return INVALID;
    if (a == b && b == c)
        return EQUILATERAL;
    if (a == b || b != c) // bug!
        return ISOSCELES;
return SCALENE;
}
```

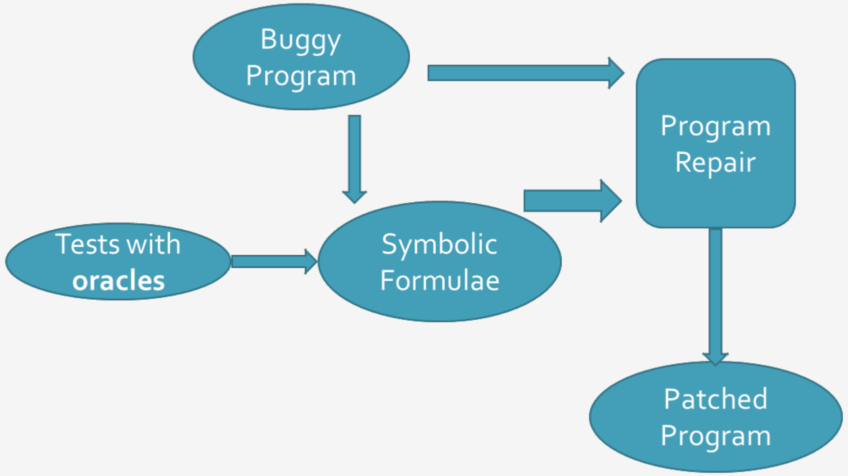
| Test id | a | b | c | oracle | Pass |
|---|---|---|---|---|---|
| 1 | -1 | -1 | -1 | INVALID | ✓ |
| 2 | 1 | 1 | 1 | EQUILATERAL | ✓ |
| 3 | 2 | 2 | 3 | ISOSCELES | ✓ |
| 4 | 2 | 3 | 2 | ISOSCELES | ✗ |
| 5 | 3 | 2 | 2 | ISOSCELES | ✗ |
| 6 | 2 | 3 | 4 | SCALANE | ✗ |

Correct fix
(a == b || b== c || a == c)

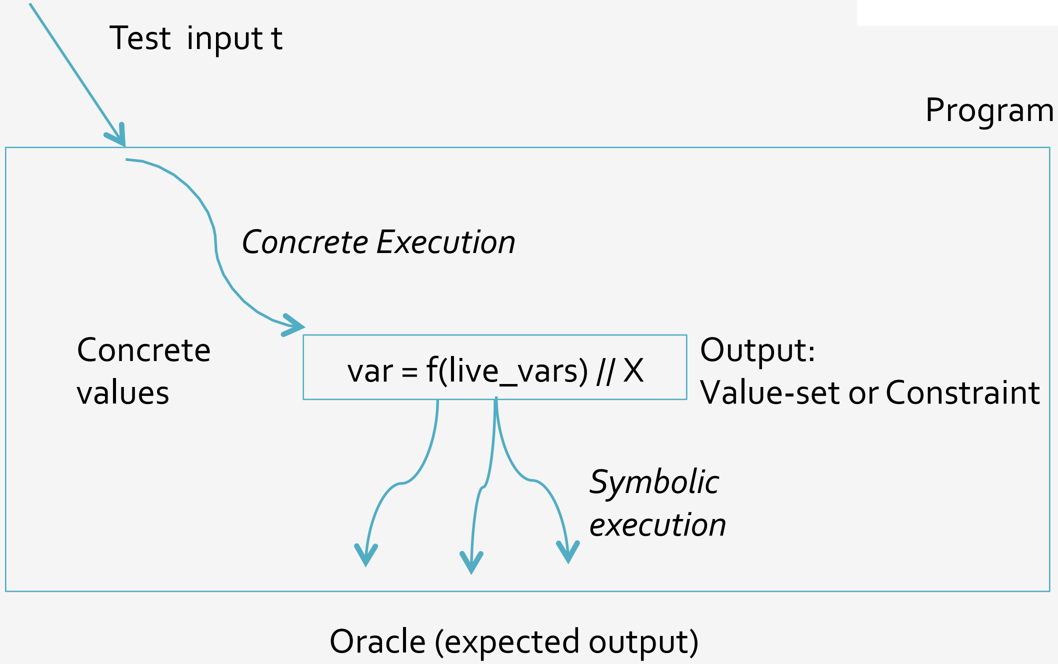
Automatically generate the constraint
$f(2,2,3) \wedge f(2,3,2) \wedge f(3,2,2) \wedge \neg f(2,3,4)$

Solution
$f(a,b,c) = (a == b \,||\, b == c \,||\, a == c)$

# Specification Inference



[ICSE13, SemFix]



$$\bigvee_{j \in Paths} (\, pc_j \wedge out_j \;==\; expected\_out(t)\,)$$

$$\wedge$$

$$f(t) \;==\; X$$

*Repair constraint*

# So, far

**Syntax-based Schematic**

```
for  e in Search-space{
     Validate e against Tests
}
```

1. Where to fix, which line?
2. Generate patches in the candidate line
3. Validate the candidate patches against correctness criterion.

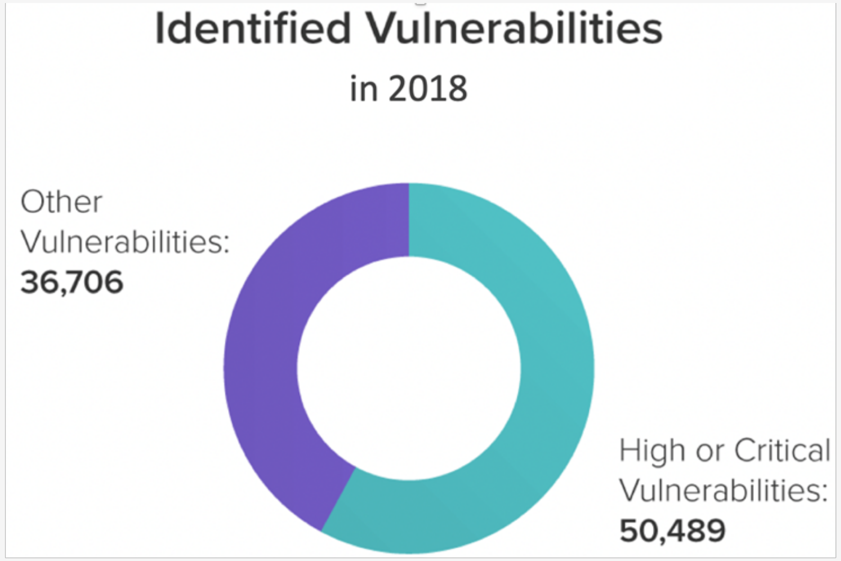
**Semantics-based Schematic**

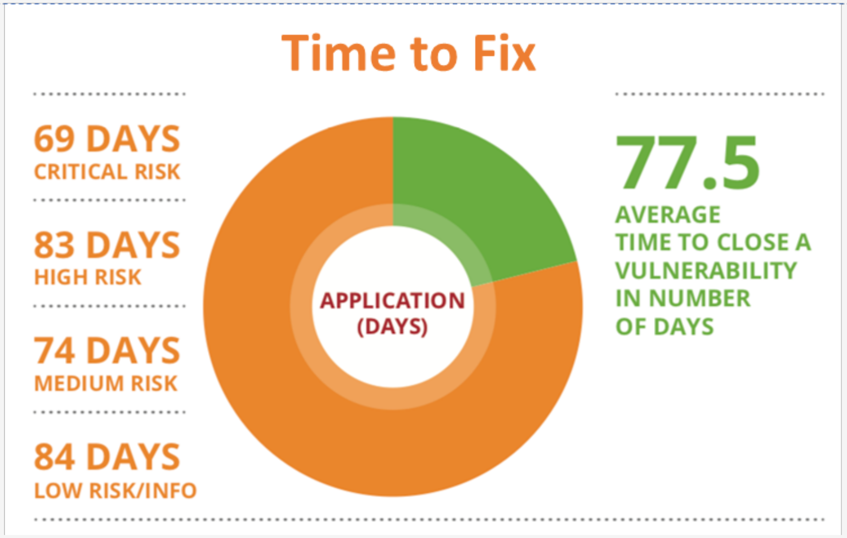
```
for  t in Tests {
     generate repair constraint $\Psi_t$
}
Synthesize e from $\wedge_t \Psi_t$
```

1. Where to fix, which line(s)?
2. What values should be returned by those lines, e.g. *<inp ==1, ret== 0>*
3. What are the expressions which will return such values?

# Shift of outlook: Vulnerability repair

Number of identified vulnerabilities in 2018: 81915





On average, it took developer 69 days to fix the critical vulnerabilities.

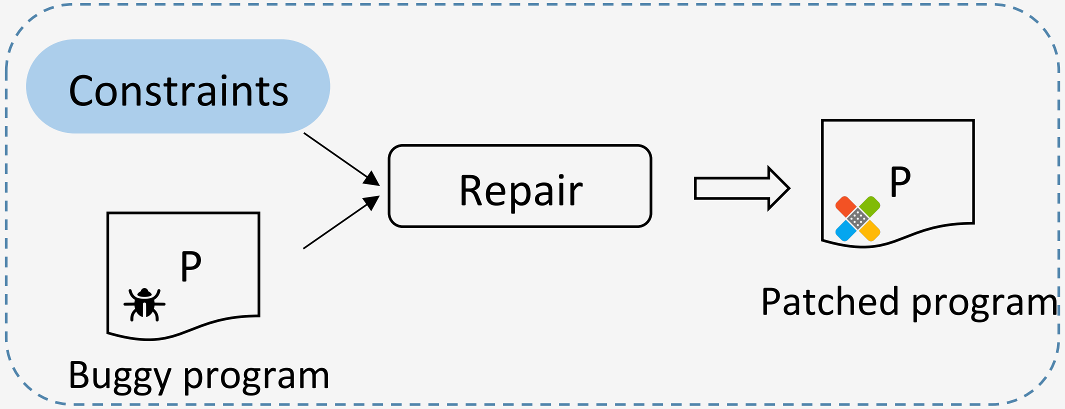
# 4. Combat Overfitting: Constraint Extraction

- Program vulnerability can be formalized as violations of constraints, e.g. buffer overflow

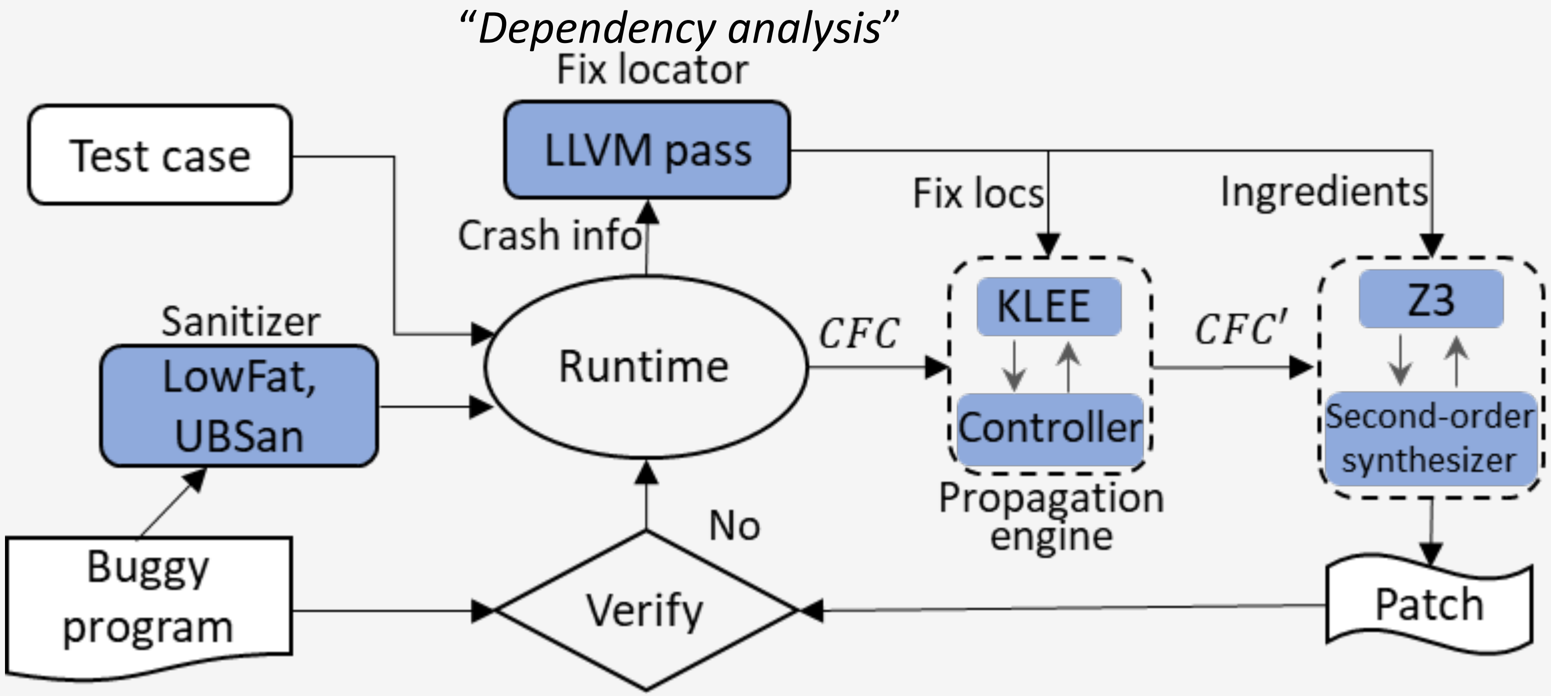
access(buffer) < base(buffer) + size(buffer)

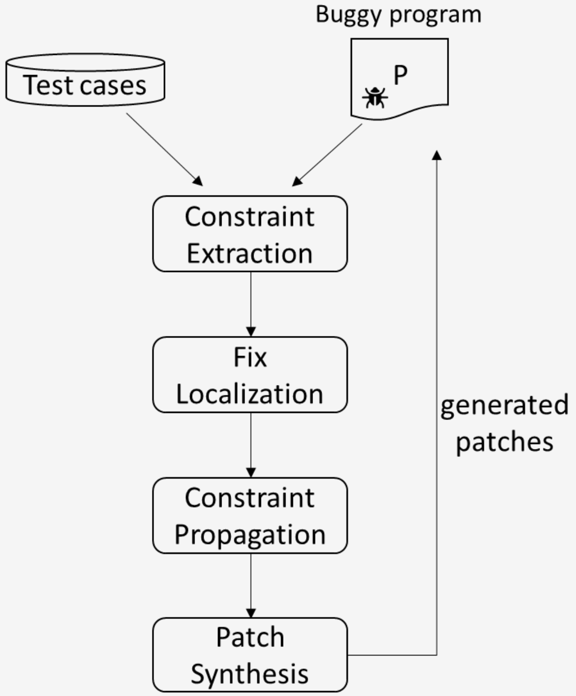
- These constraints can be **automatically** extracted when a vulnerability/crash is witnessed on a given test

- The constraints serve as additional specifications for Automated program repair (APR) to fix the bug for all tests.



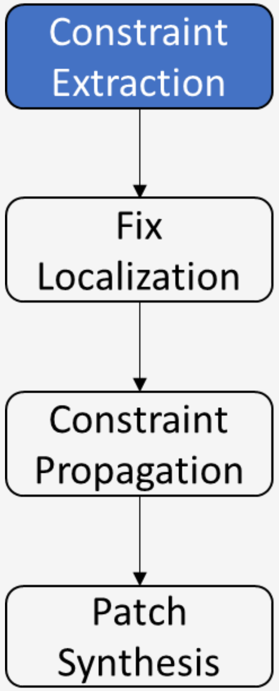
# Solution





*"The C and C++ programming languages are notoriously insecure yet remain indispensable. Developers therefore resort to a multi–pronged approach to find security issues before adversaries. These include manual, static, and dynamic program analysis. Dynamic bug finding tools or "**sanitizers**" --- can find bugs that elude other types of analysis because they observe the actual execution of a program, and can therefore directly observe incorrect program behavior as it happens." Song et al 2018.*

# Constraint Extraction

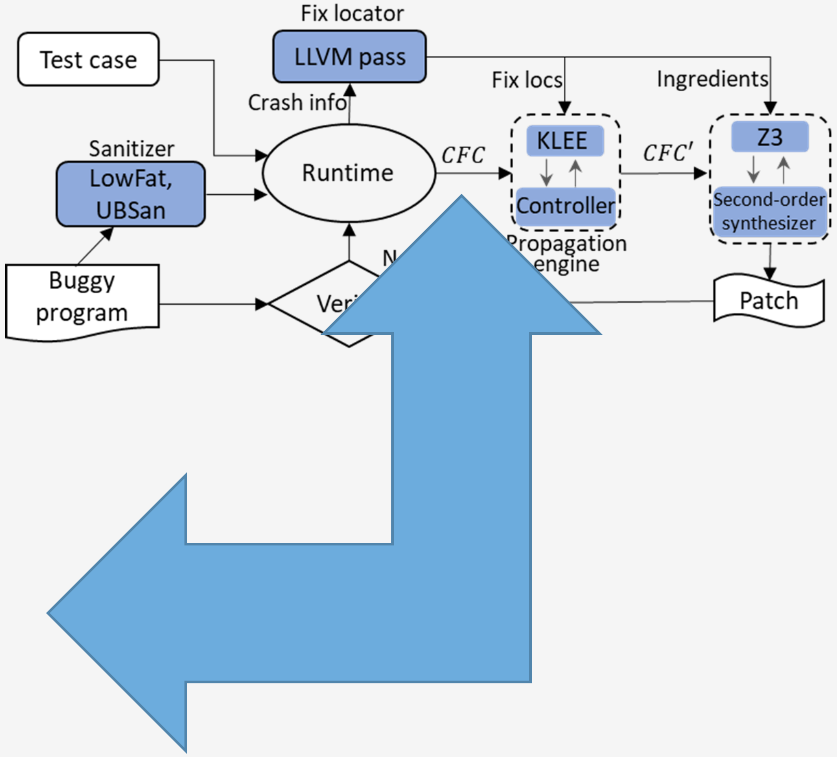

Constraint Extraction

Fix Localization

Constraint Propagation

Patch Synthesis

```
char getValue(char[] arr, int index){
    int len = size(arr);
    if (index <= len)
        return arr[index];
    return 0;
}
```



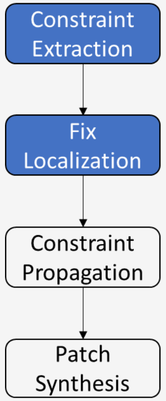
## Detect buggy program state on concrete input

- input -> arr: {1, 2, 3}; index: 3
- Buggy state -> arr[3]
- Concrete constraint violation: 3 (*index*) >= 3 (*len*)

## Generalize crash-free constraints $\varphi$ to cover the whole input space

- Using template-based approach
- Map concrete states to symbolic states
- Symbolic constraint violation: *index* >= *len*

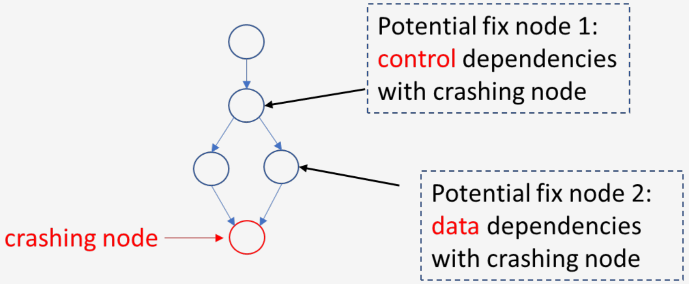
# Constraint Propagation



- Spectrum-base Fault localization depends a number of high-quality test cases.
- In a common scenario when security vulnerabilities are found, only one failing test (exploit) is available.

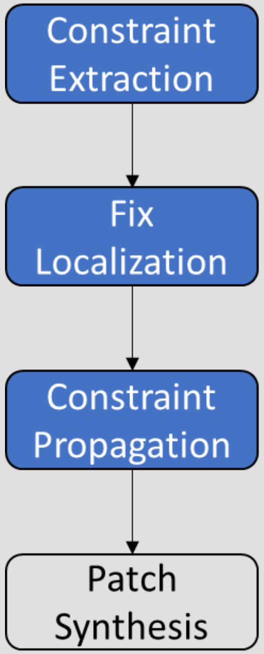
⇩

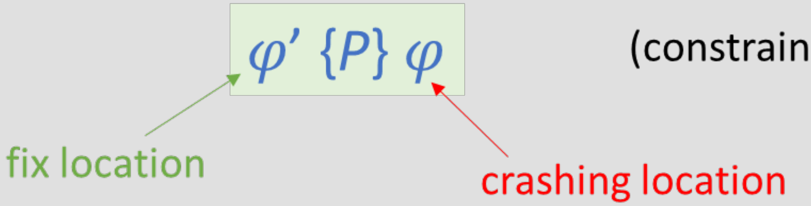
Dependency-based fix localization



Potential fix node 1: control dependencies with crashing node

Potential fix node 2: data dependencies with crashing node





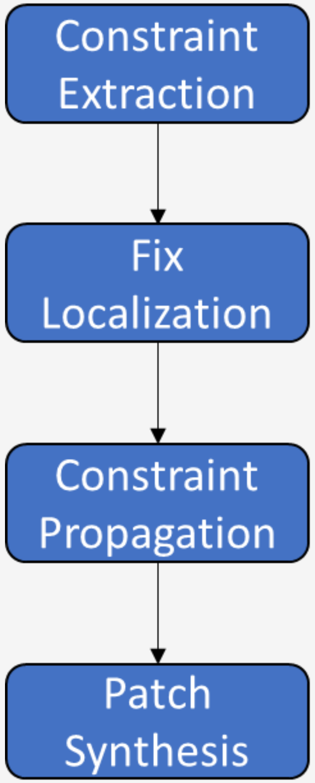
$$\varphi' \{P\} \varphi$$ (constraint-propagation)

$\varphi$: the crash-free constraint at crashing location

*P:* the program statements between fix and crashing location.

$\varphi$ ': the least restrictive (weakest) precondition that will guarantee the postcondition $\varphi$

# Synthesis



$[e \longmapsto e']\varphi' \; \{P\} \; \varphi$ (repair)

To ensure $\varphi'$ is satisfied after applying the patch, solve second-order formula.

$$\bigwedge_{j=1}^{|\Pi|} e' = f(V) \wedge pc_i \Rightarrow \varphi'_i$$

We are synthesizing a second-order expression $f$, which takes as inputs the live variables $V$.

After applying $f$, all the $\varphi'_i$ is guaranteed to be satisfied.

$\varphi'_i$ is generated by backward propagating $\varphi$ along path $i$.

# Data-set and Results on CVEs

The correct/total patches generated by Prophet, Angelix, Fix2Fit and ExtractFix

| Program | #CVEs | Prophet | Angelix | Fix2Fit | ExtractFix |
|---|---:|---:|---:|---:|---:|
| Libtiff | 11 | 1 / 7 | 0 / 7 | 1 / 7 | 6 / 9 |
| Binutils | 2 | - | - | 0 / 1 | 1 / 2 |
| Libxml2 | 5 | 0 / 3 | 0 / 0 | 1 / 4 | 2 / 4 |
| Libjpeg | 4 | 1 / 3 | - | - | 2 / 3 |
| FFmpeg | 2 | - | - | 1 / 2 | 1 / 2 |
| Jasper | 2 | 0 / 2 | 0 / 2 | 0 / 2 | 1 / 2 |
| Coreutil | 4 | 0 / 2 | - | 1 / 3 | 2 / 2 |
| Total | 30 | 2 / 17 | 0 / 9 | 4 / 19 | 16 / 24 |

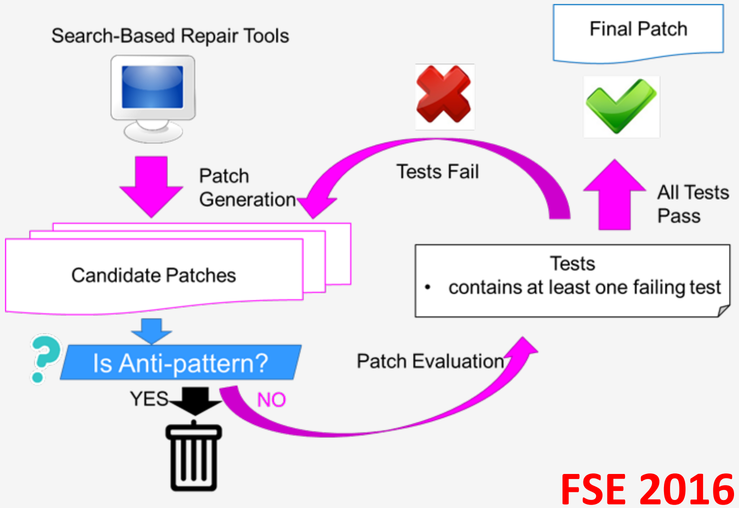
*Prophet: Uses enumerative search and machine learning for ranking patches.*

*Angelix: scalable version of symbolic execution plus synthesis based approach.*

*Fix2Fit: combination of repair and test generation using fuzzing*

*ExtractFix: Extract constraints using sanitizers*

# Over-fitting in Program Repair



Search-Based Repair Tools

Patch Generation

Candidate Patches

Is Anti-pattern?

YES

NO

Tests Fail

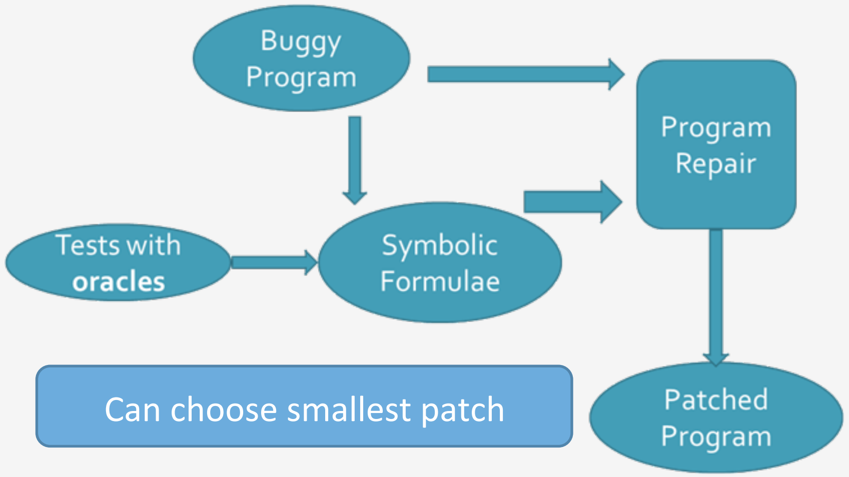
Final Patch

All Tests Pass

Tests
- contains at least one failing test

Patch Evaluation

**FSE 2016**



Buggy Program

Tests with **oracles**

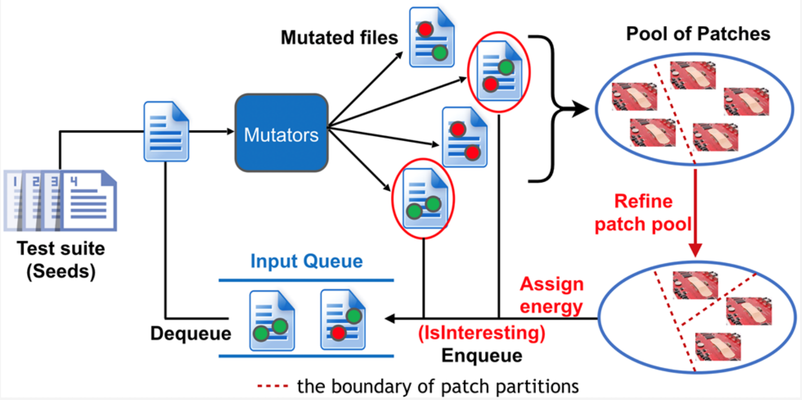
Symbolic Formulae

Program Repair

Patched Program

Can choose smallest patch

**ICSE 2013, ICSE 2015**



Mutated files

Pool of Patches

Mutators

Test suite (Seeds)

Refine patch pool

Input Queue

Assign energy

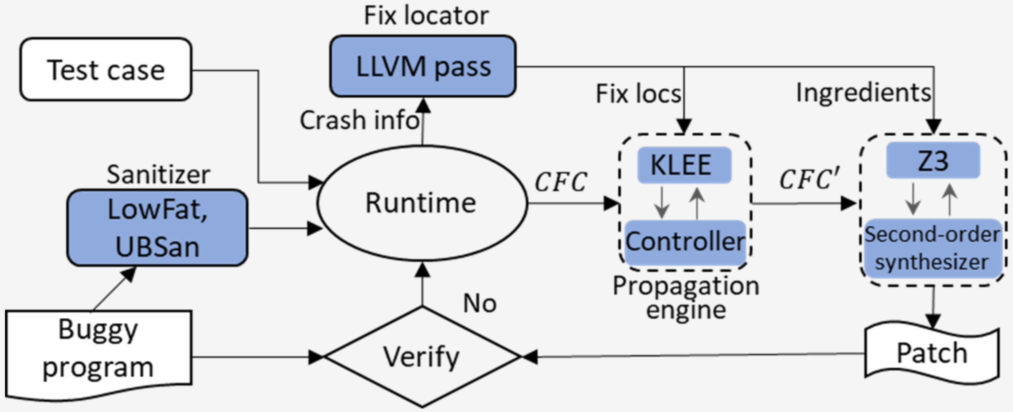
Dequeue

(IsInteresting)

Enqueue

---- the boundary of patch partitions

**TOSEM 2018, ISSTA 2019**



Test case

Fix locator

LLVM pass

Crash info

Fix locs

Ingredients

Sanitizer

LowFat, UBSan

Runtime

$CFC$

KLEE

Controller

Propagation engine

$CFC'$

Z3

Second-order synthesizer

No

Buggy program

Verify

Patch

**Recent Unpublished**