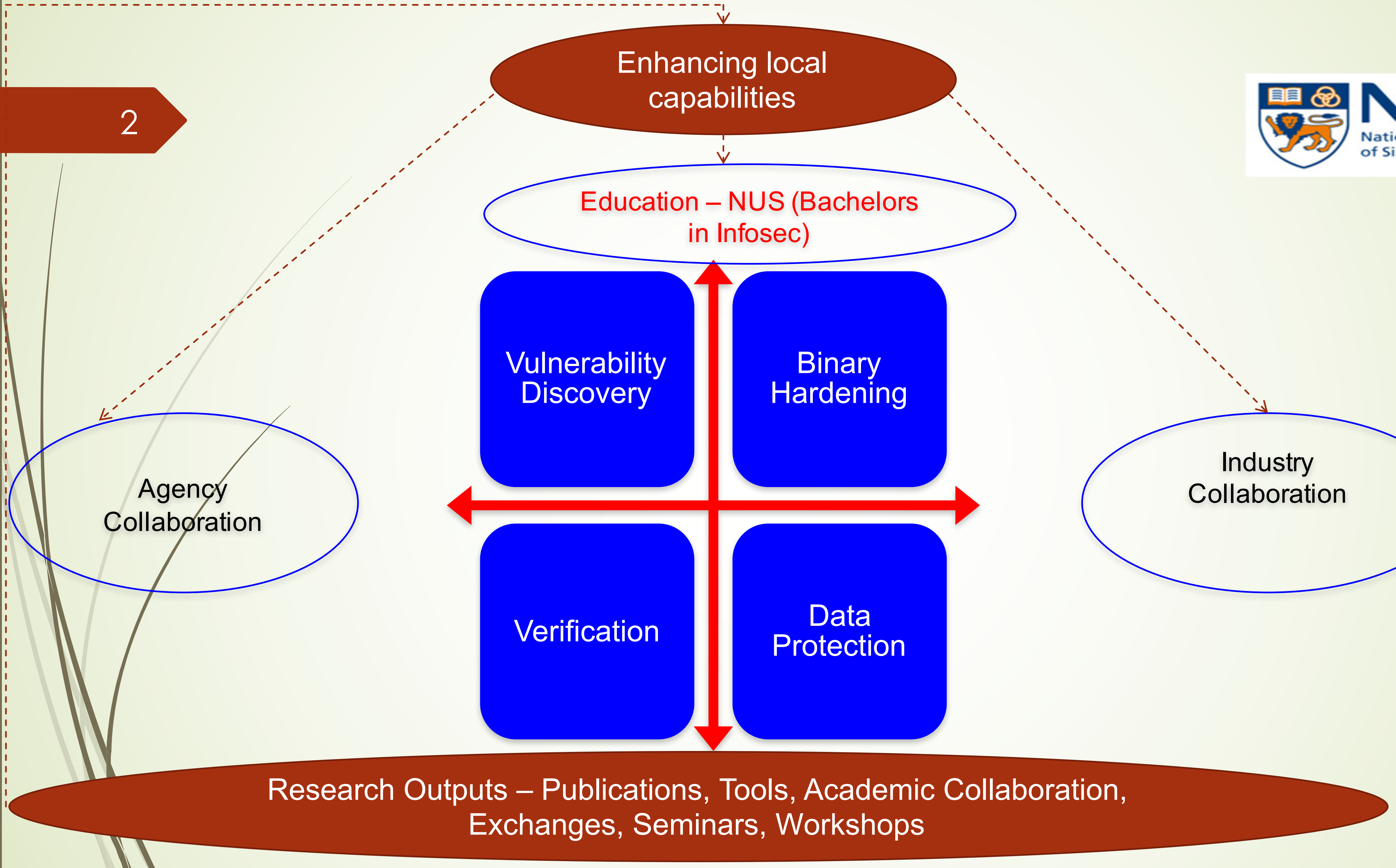
Software Vulnerability Detection and Repair

1

Prof. Abhik Roychoudhury
National University of Singapore

KLEE Workshop London 2018


Enhancing local capabilities
Education – NUS (Bachelors in Infosec)
Agency Collaboration
Vulnerability Discovery
Binary Hardening
Verification
Data Protection
Industry Collaboration
Research Outputs – Publications, Tools, Academic Collaboration, Exchanges, Seminars, Workshops

Space of Problems

- **Fuzz Testing**
  - Feed semi-random inputs to find hangs and crashes
- **Continuous fuzzing**
  - Incrementally find new “problems” in software
- **Crash reproduction**
  - Re-construct a reported crash, crashing input not included due to privacy
- **Reaching nooks and corners**
- **Localizing reported observable errors**
- **Patching reported errors from input-output examples**

Space of Techniques

Search

- Random
- Biased-random
- Genetic (AFL Fuzzer)
- …

- *Low set-up overhead*
- *Fast, less accurate*
- **Use objective function to steer**

Symbolic Execution

- Dynamic Symbolic execution
- Concolic Execution
- Cluster paths based on symbolic expressions of variables
- ….

- *High set-up overhead*
- *Slow, more accurate*
- **Use logical formula to steer**

In this talk …

Search

- Enhance the effectiveness of search techniques, with symbolic execution as inspiration

- [CCS16, CCS17, ICSE15]

Symbolic Execution

- Explore capabilities of symbolic execution beyond search, in program repair

- [ICSE13, 15, 16, 18]

History of fuzzing

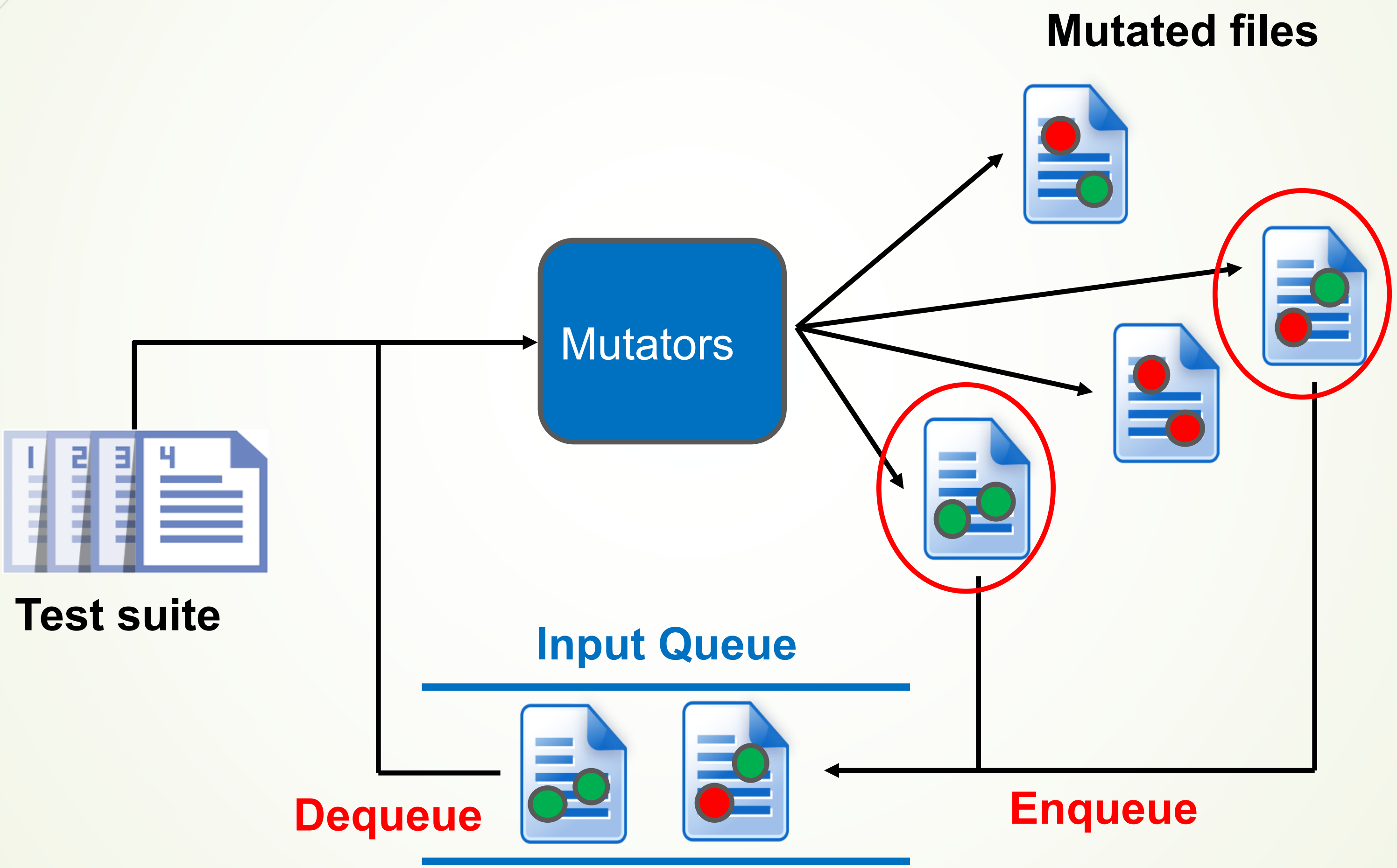
Developed by Barton Miller, see

http://pages.cs.wisc.edu/~bart/fuzz/

Fuzz testing is a simple technique for feeding random input to applications. The approach has three characteristics.

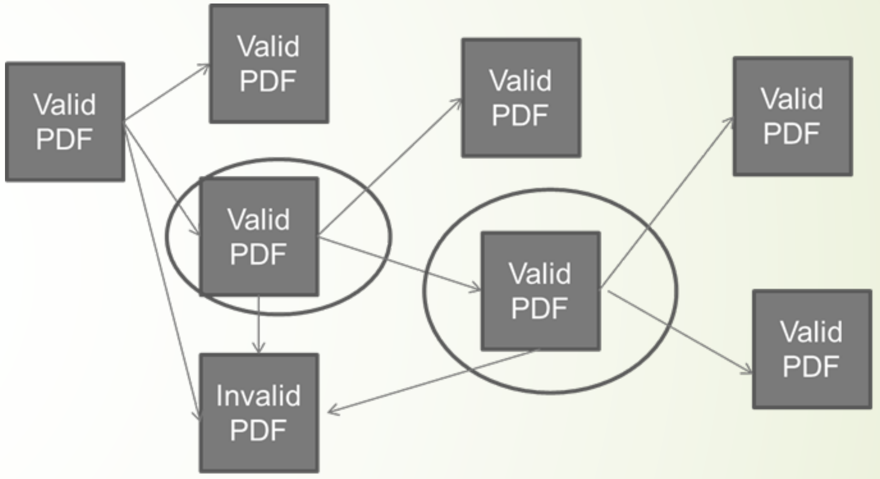
- *The input is **random**. We do not use any model of program behavior, application type, or system description. This is sometimes called **black box testing**.*
- *The reliability criteria is **simple**: if the application **crashes or hangs**, it is considered to fail the test, otherwise it passes. Note that the application does not have to respond in a sensible manner to the input, and it can even quietly exit.*
- *As a result of the first two characteristics, fuzz testing can be **automated** to a high degree and results can be compared across applications, operating systems, and vendors.*

Grey-box Fuzzing, as in AFL



Grey-box Fuzzing Algorithm

- Input: Seed Inputs S
- 1: $T_x = \varnothing$
- 2: T = S
- 3: if T = $\varnothing$ then
- 4: add empty file to T
- 5: end if
- 6: repeat
- 7: t = **chooseNext**(T)
- 8: p = **assignEnergy**(t)
- 9: for i from 1 to p do
- 10: t0 = **mutate_input**(t)
- 11: if t0 crashes then
- 12: add t0 to T_x
- 13: else if **isInteresting**(t0) then
- 14: add t0 to T
- 15: end if
- 16: end for
- 17: until timeout reached or abort-signal
- Output: Crashing Inputs T_x



Exercises common path that rejects invalid PDF

Short paths rejecting syntactically invalid inputs are exercised by fuzzer too frequently

Programming by experienced people

Schematic

- if (condition1)
-     return   *// short path, frequented by many many inputs*
- else if (condition2)
-     exit   *// short paths, frequented by many inputs*
- else ….

Prioritize low probability paths

- ✓ Use grey-box fuzzer which keeps track of path id for a test.
- ✓ Find probabilities that fuzzing a test t which exercises π leads to an input which exercises π'

$$\pi \xrightarrow{p} \pi'$$

- ✓ Higher weightage to low probability paths discovered, to gravitate to those -> discover new states in Markov Chain with minimal effort.

```
void crashme (char* s) {
    if (s[0] == 'b')
        if (s[1] == 'a')
            if (s[2] == 'd')
                if (s[3] == '!')
                    abort ();
}
```

Power-Schedules

- Constant: $p(i) = \alpha(i)$
  - AFL uses this schedule (fuzzing ~1 minute)
  - $\alpha(i)$ .. how AFL judges fuzzing time for the test exercising path i
- Cut-off Exponential:

$$\boxed{p(i) = \begin{cases} 0, & \text{if } f(i) > \mu \\ \min\left(\alpha(i)/\beta * 2^{s(i)}, M\right) & \text{otherwise} \end{cases}}$$

β is a constant

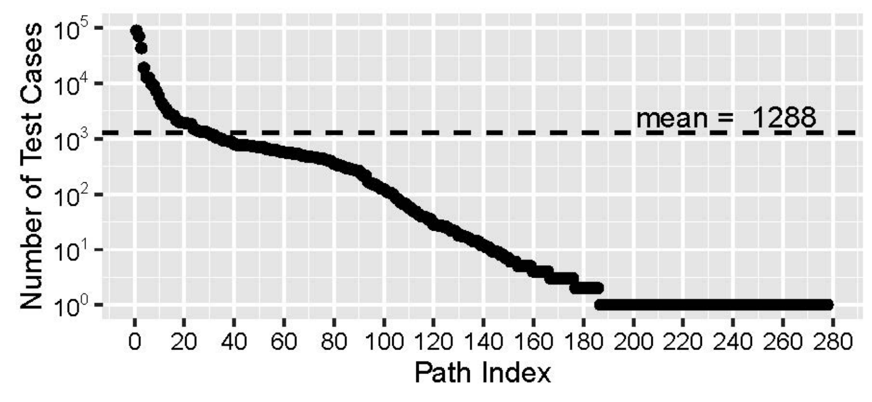
$s(i)$ #times the input exercising path i has been chosen for fuzzing

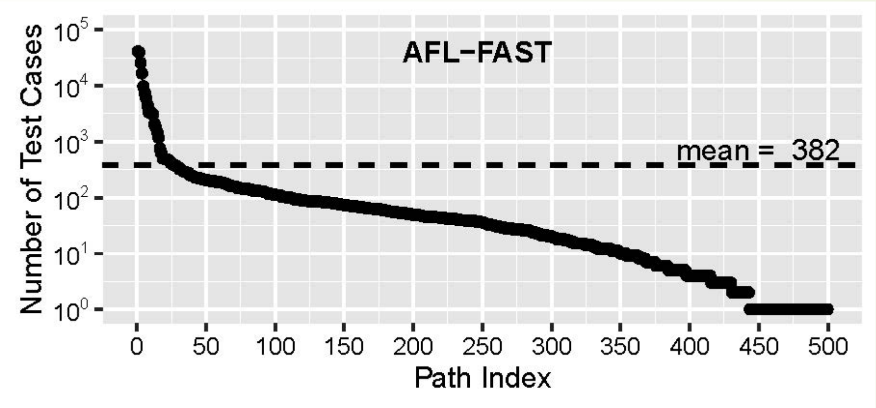
$f(i)$ #fuzz exercising path i (path-frequency)

μ mean #fuzz exercising a discovered path (avg. path-frequency)

M maximum energy expendable on a state

Results





Independent evaluation found crashes 19x faster on DARPA Cyber Grand Challenge (CGC) binaries

Integrated into main-line of AFL fuzzer within a year of publication (CCS16), which is used on a daily basis by corporations for finding vulnerabilities

Comments on the technologies

Y **Hacker News** new | comments | show | ask | jobs | submit login

▲ Fuzzing Perl: A Tale of Two American Fuzzy Lops (geeknik.net)

82 points by geeknik 66 days ago | hide | past | web | 18 comments | favorite

The paper [1] on AFLFast is, IMO, a great example of where academia shines: carefully looking at how and why something works, developing some theory and a working model, and then using that to get a substantial improvement on the state of the art (and doing a nice evaluation to show that it really works).

▲ nickpsecurity 65 days ago [-]

The abstract sounds like it. They said with no program analysis, though. I thought program analysis was good enough that it could probably auto-generate tests for every branch in a program, possibly in less time or with more assurance. W as I wrong or is this a parallel subfield?

▲ moyix 65 days ago [-]

The question of whether randomized testing or program analysis gives you more coverage of a program is a really interesting one. Böhme actually has an earlier paper that addresses this question: https://www.comp.nus.edu.sg/~mboehme/paper/FSE14.pdf

Use of Grey-box Fuzzing

- **Greybox Fuzzing** is frequently used, daily in corporations
  - **State-of-the-art** in automated vulnerability detection
  - **Extremely efficient** coverage-based input generation
    - All program analysis before/at **instrumentation time**.
    - Start with a seed corpus, choose a seed file, **fuzz it**.
    - Add to corpus **only if new input increases coverage**.
  - **Cannot be directed, unlike symbolic execution!**

In this talk …

Search

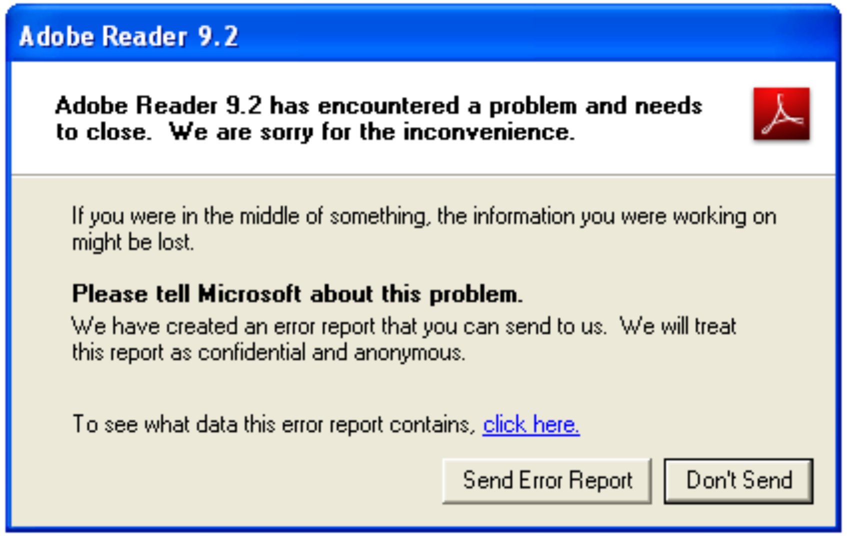
- Enhance the effectiveness of search techniques, with symbolic execution as inspiration

  - **Enhance coverage, how to make it directed?**

Symbolic Execution

- Explore capabilities of symbolic execution beyond directed search

Directed Fuzzing instead of Coverage

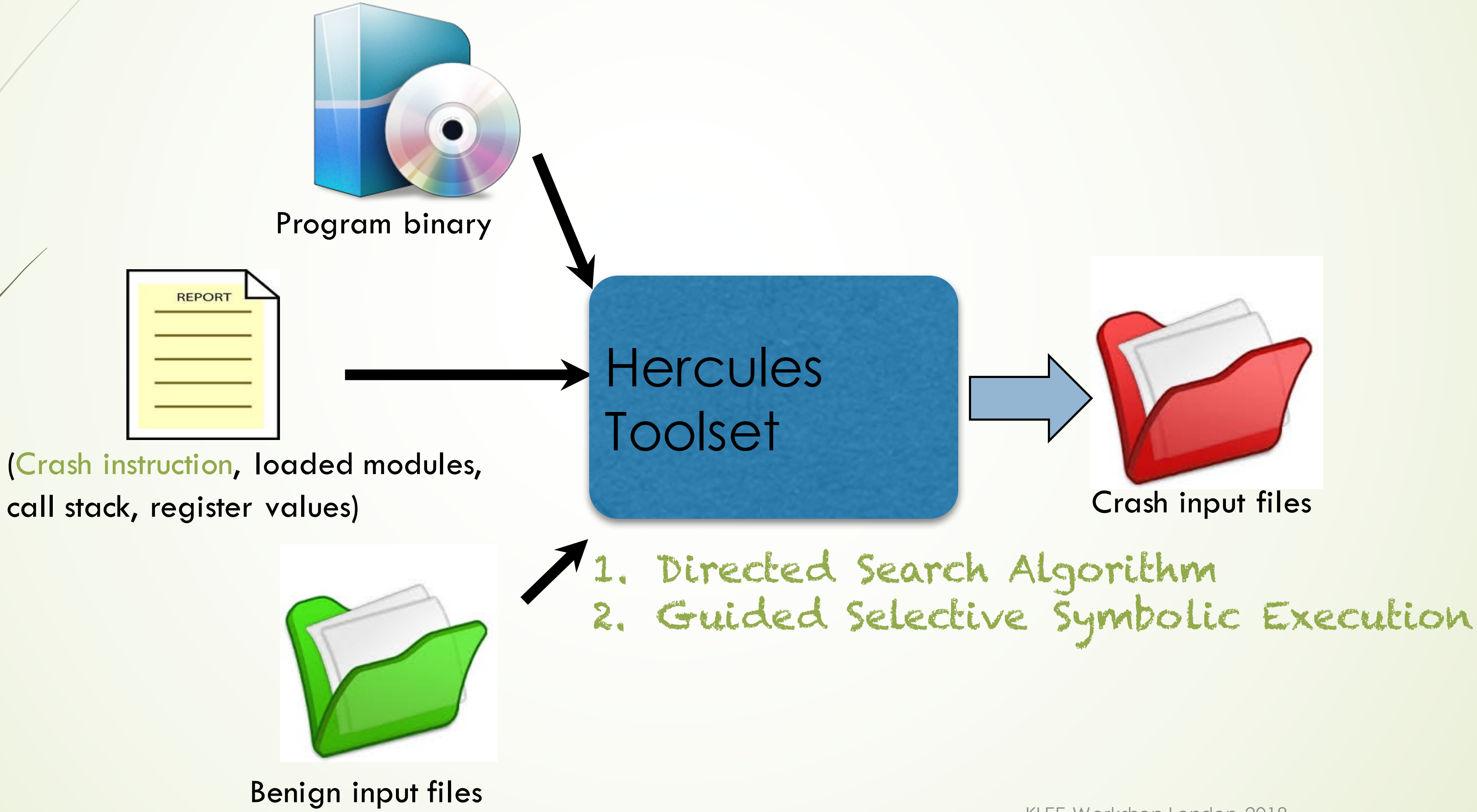


Crash reproducing supports

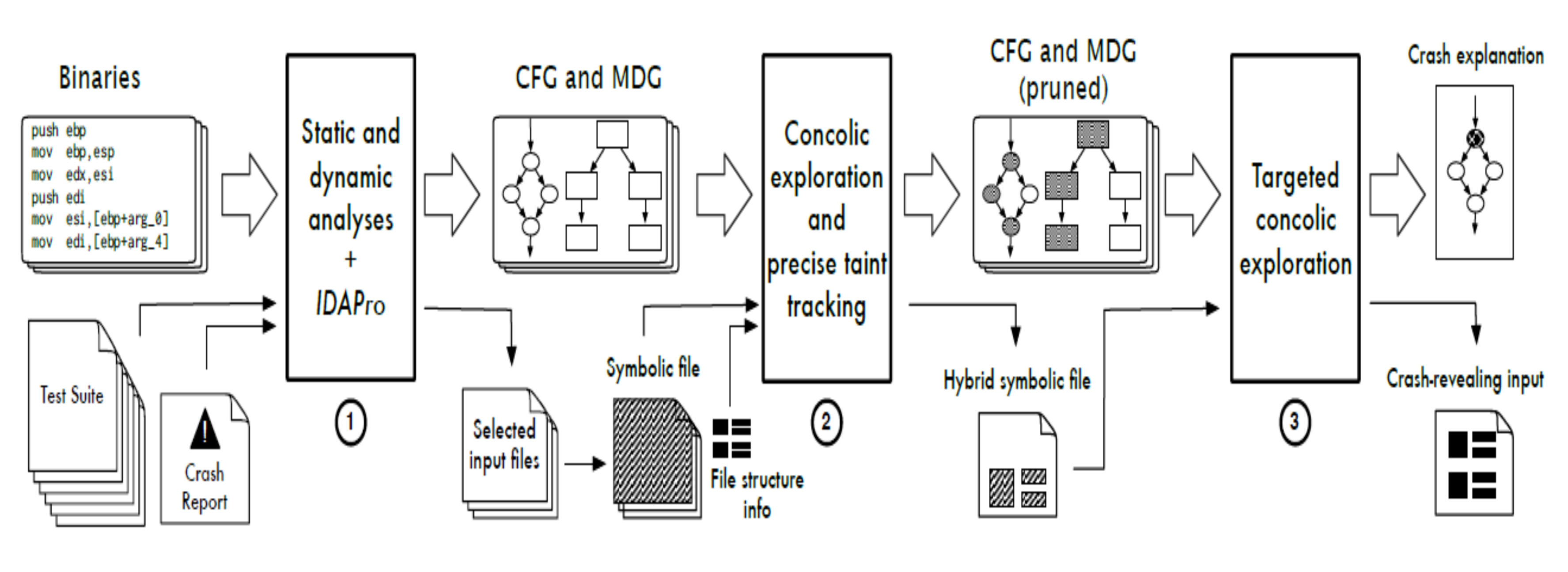
- In-house debugging and fixing
- Vulnerability checking

Using symbolic execution

Reproduced vulnerabilities in Acrobat Reader, Media Player with 24 hour time bound



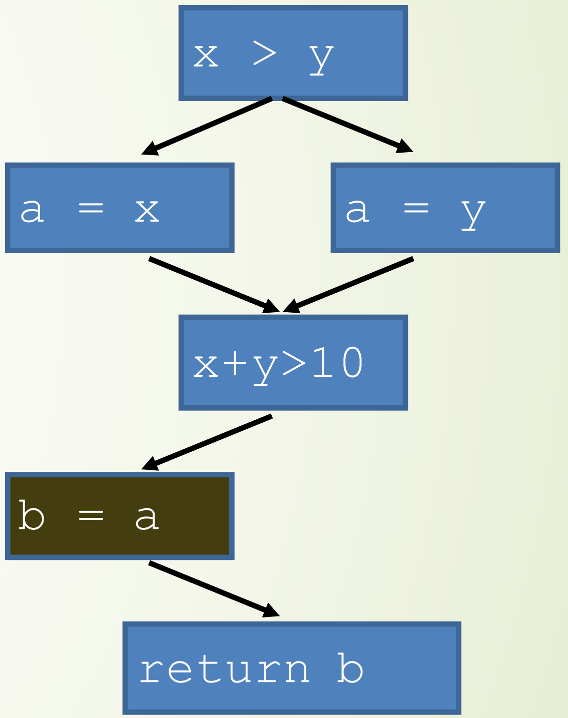
Symbolic Analyzer



Reproduced vulnerabilities in Acrobat Reader, Media Player with 24 hour time bound

(Earlier) View-point

- **Directed Fuzzing: classical constraint satisfaction prob.**
  - **Program analysis to identify program paths that reach given program locations.**
  - **Symbolic Execution to derive path conditions for any of the identified paths.**
  - **Constraint Solving to find an input that**
    - **satisfies the path condition and thus**
    - **reaches a program location that was given.**



$$\varphi_1 = (x>y) \wedge (x+y>10)$$
$$\varphi_2 = \neg(x>y) \wedge (x+y>10)$$

(Later) View-point

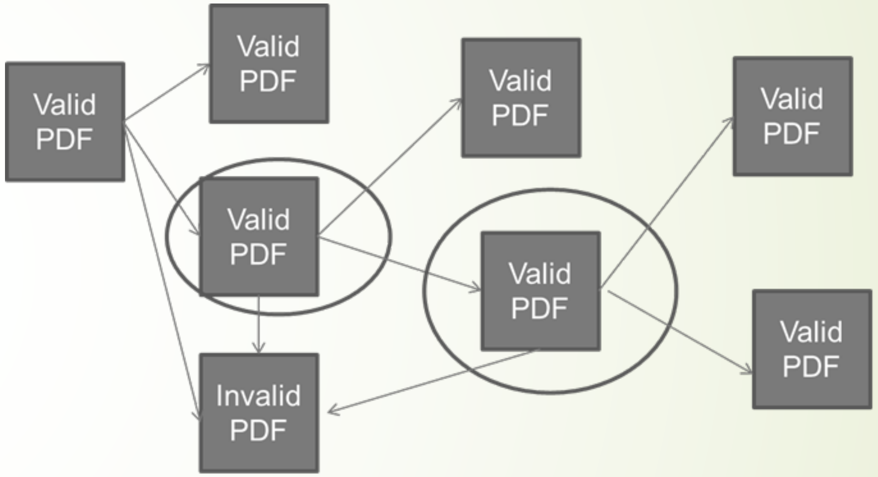
- **Directed Fuzzing** as **optimization problem**!

1. **Instrumentation Time:**
   - Instrument program to **aggregate distance values**.

2. Runtime, **for each input**
   - decide **how long to be fuzzed** based on distance.
     - If input is **closer** to the targets, it is fuzzed for **longer**.
     - If input is **further away** from the targets, it is fuzzed for **shorter**.

Power Schedules - Recap

- Input: Seed Inputs S
- 1: $T_{x} = \varnothing$
- 2: T = S
- 3: if T = $\varnothing$ then
- 4: add empty file to T
- 5: end if
- 6: repeat
- 7: t = **chooseNext**(T)
- 8: p = **assignEnergy**(t)
- 9: for i from 1 to p do
- 10: t0 = **mutate_input**(t)
- 11: if t0 crashes then
- 12: add t0 to T_{x}
- 13: else if **isInteresting**(t0) then
- 14: add t0 to T
- 15: end if
- 16: end for
- 17: until timeout reached or abort-signal
- Output: Crashing Inputs T_{x}

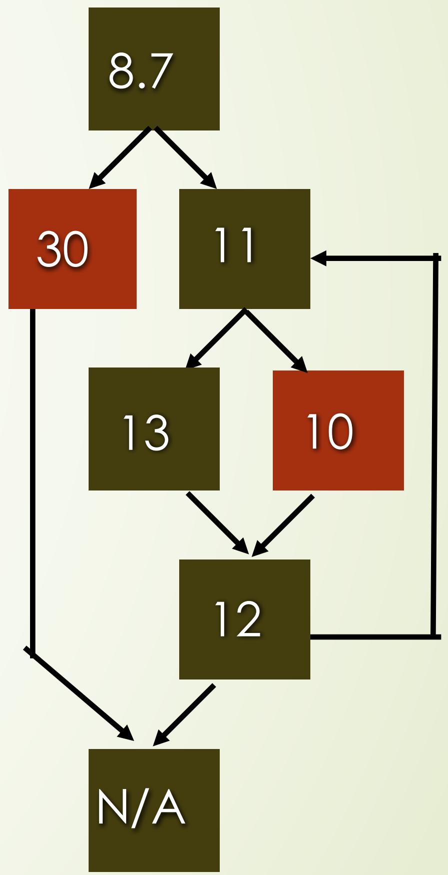


Exercises common path that rejects invalid PDF

Short paths rejecting syntactically invalid inputs are exercised by fuzzer too frequently

Instrumentation

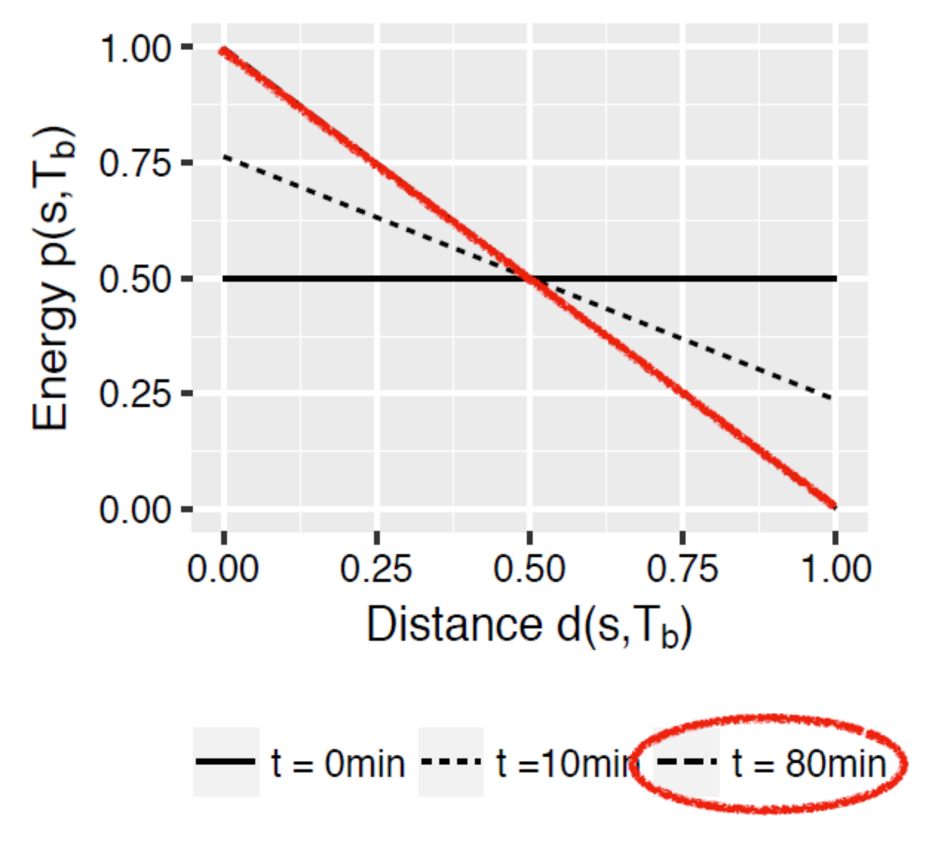
- **Function-level target distance** using call graph (CG)
- **BB-level target distance** using control-flow graph (CFG)
  1. Identify **target BBs** and assign **distance 0**
  2. Identify BBs that call **functions** and assign **10*FLTD**
  3. For **each BB**, compute harmonic mean of (length of shortest path to any function-calling BB + 10*FLTD).



CFG for function b

Directed fuzzing as optimization

- Integrating Simulated Annealing as power schedule
  - In the beginning (t = 0min), assign the **same energy to all seeds**.
  - Later (t=10min), assign a **bit more energy** to seeds that are **closer**.
  - At exploitation (t=80min), assign **maximal energy** to seeds that are **closest**.



In this talk …

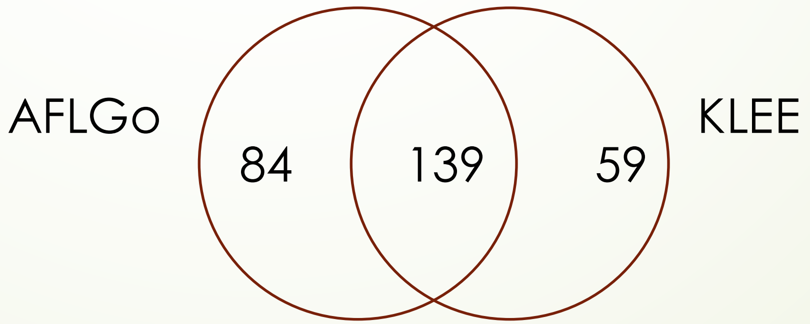
Search

- Enhance the effectiveness of search techniques, with symbolic execution as inspiration

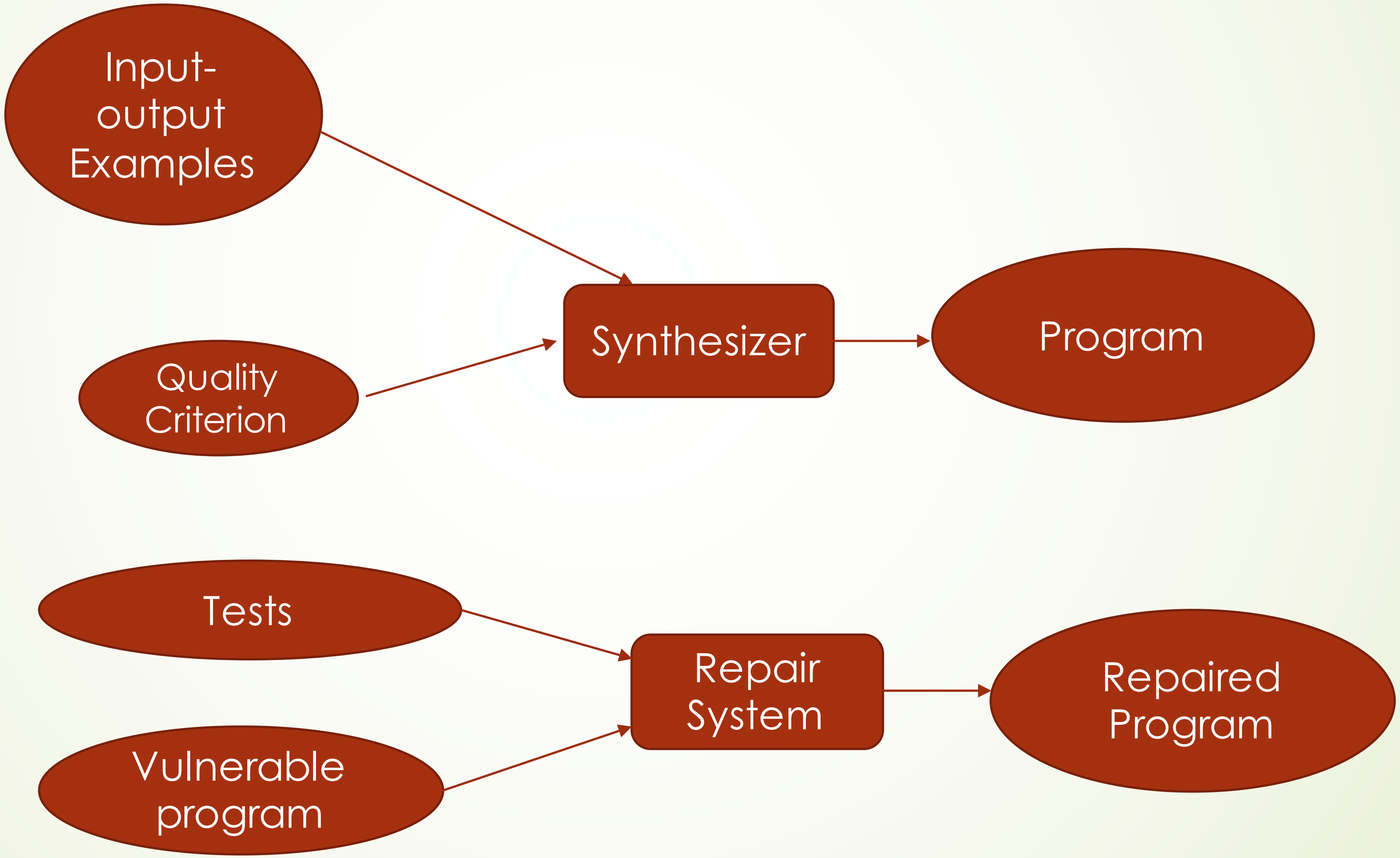
Symbolic Execution

- **Explore capabilities of symbolic execution beyond directed search**

- **Enhance coverage**
- **Achieve directed search**

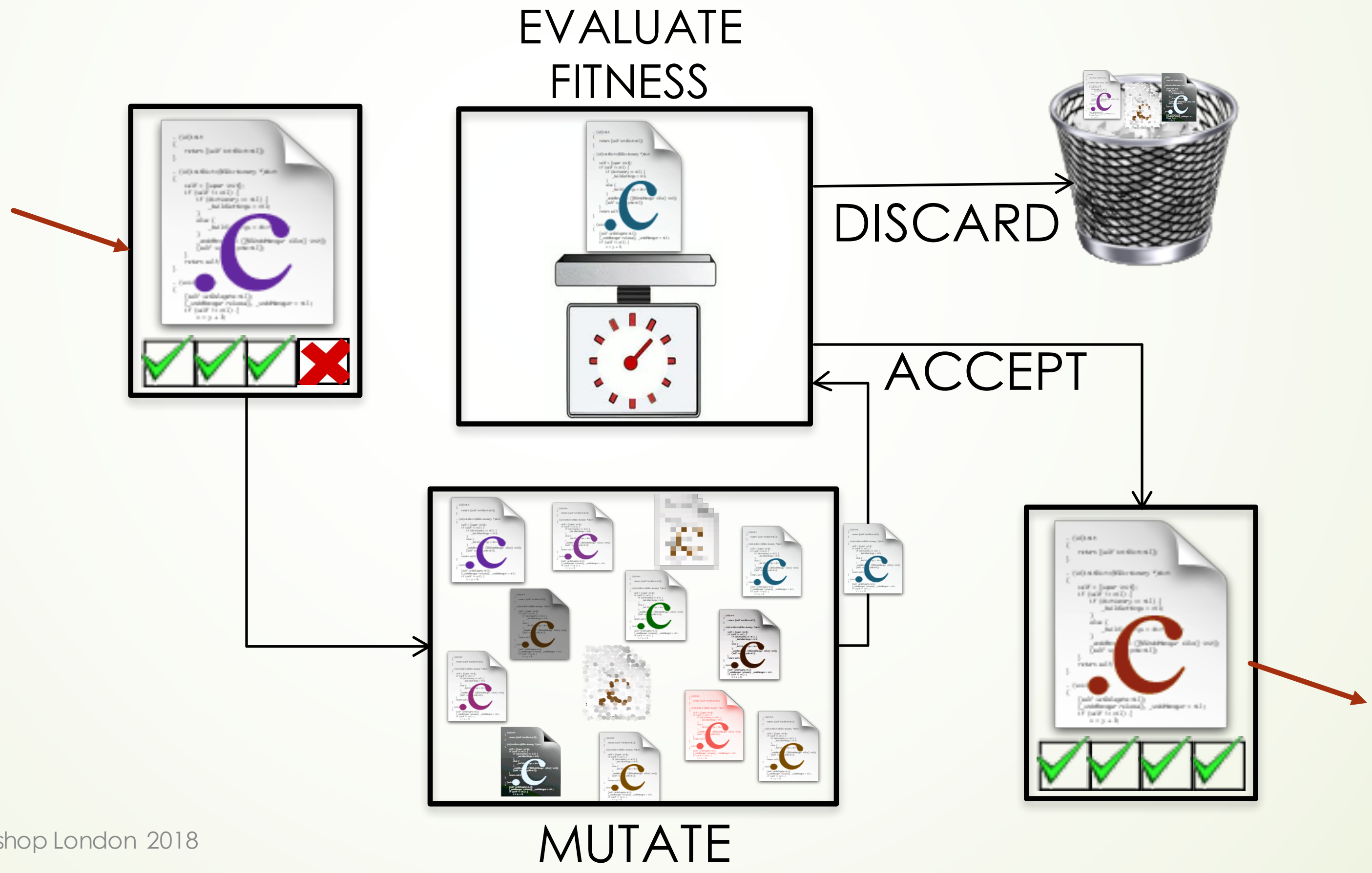


Program Synthesis and Repair



Search-based approach

- 2009: GenProg [Weimer-et-al-ICSE]

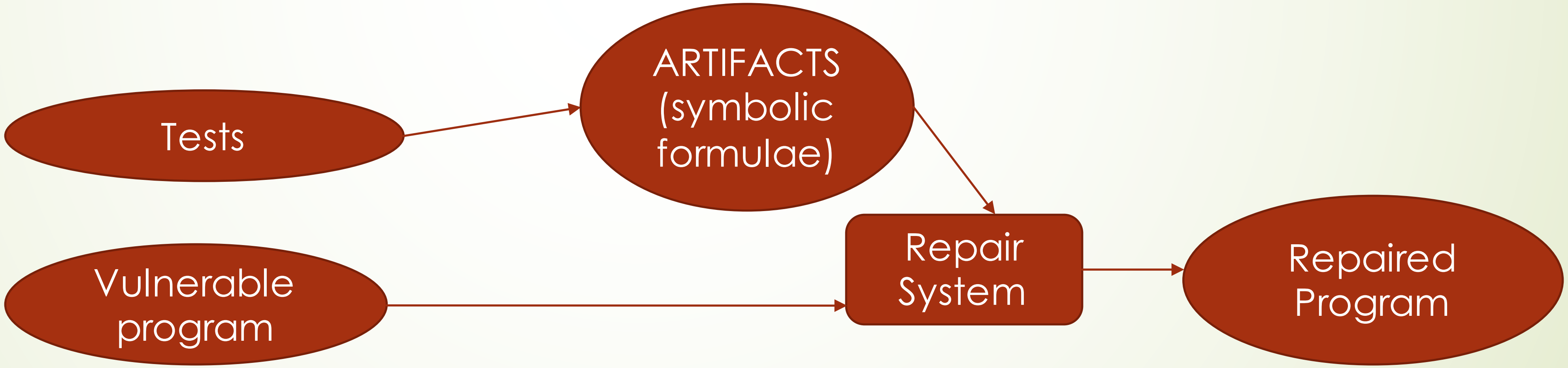


Over-fitting in Tests -> Program

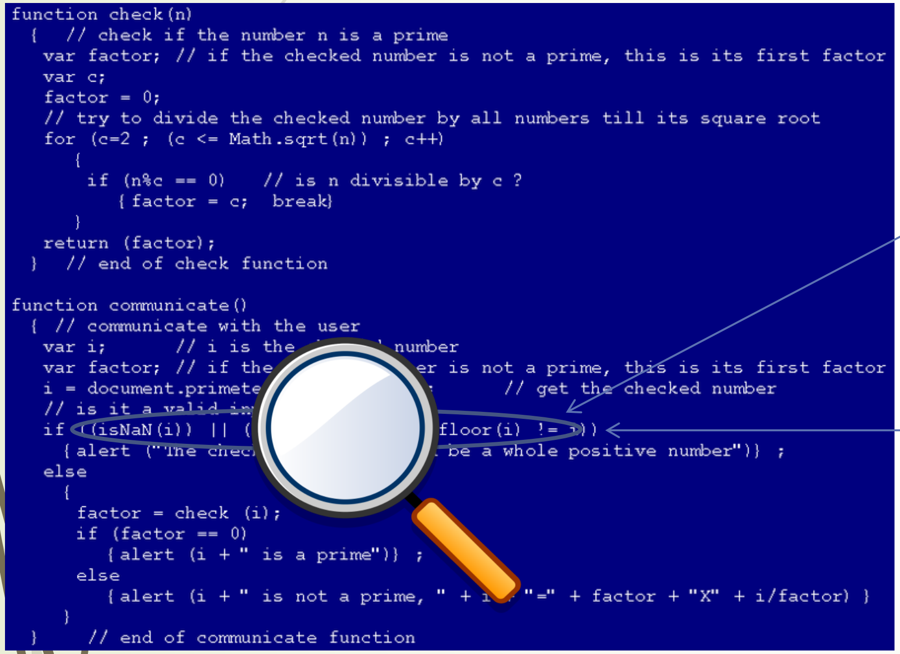
Avoid

if (input1) return output1
else if (input2) return output2
else if (input3) return output3
….

Generalize beyond the provided tests using symbolic reasoning.



View-point on Repair



3. Validate the candidate patches.

2. Generate the candidate patches in this line.

1. Where to fix – in which line?

Syntactic approach

3. What are the expressions which will return these values?

2. What values should be returned by these lines? <*inp=1, ret=0*>

1. Where to fix – in which line?

Semantic approach

Example

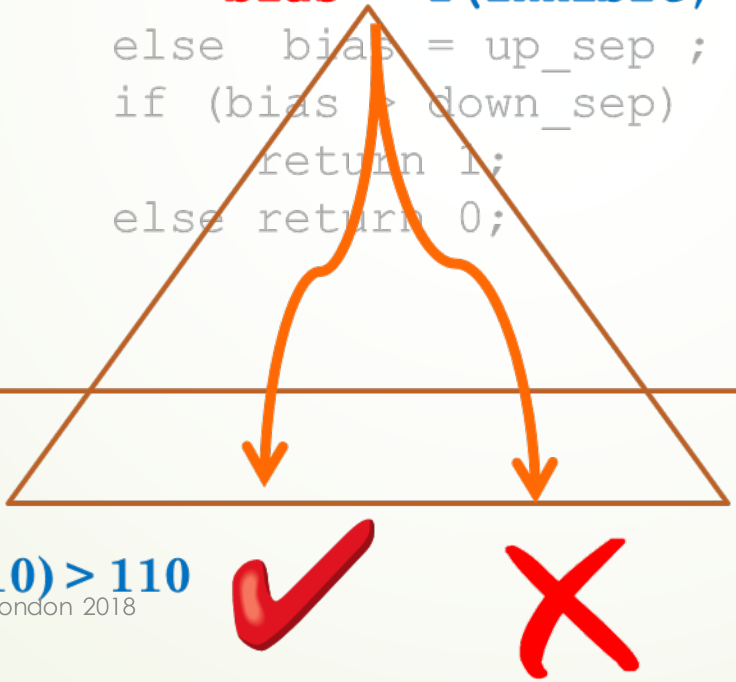
```
int is_upward( int inhibit, int up_sep, int down_sep){
       int bias;
       if (inhibit)
           bias = down_sep; //   bias= up_sep + 100
       else  bias = up_sep ;
       if (bias > down_sep)
            return 1;
       else return 0;
}
```

inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**
1	0	100	0	0	pass
1	***11***	***110***	***0***	***1***	***fail***
0	100	50	1	1	pass
1	***-20***	***60***	***0***	***1***	***fail***
0	0	10	0	0	pass

Patch synthesis

Inhibit == 1	up_sep == 11	down_sep == 110

```
int is_upward( int inhibit, int up_sep, int
down_sep){
      int bias;
      if (inhibit)
            bias = f(inhibit, up_sep, down_sep)
      else  bias = up_sep ;
      if (bias > down_sep)
            return 1;
      else return 0;
}
```



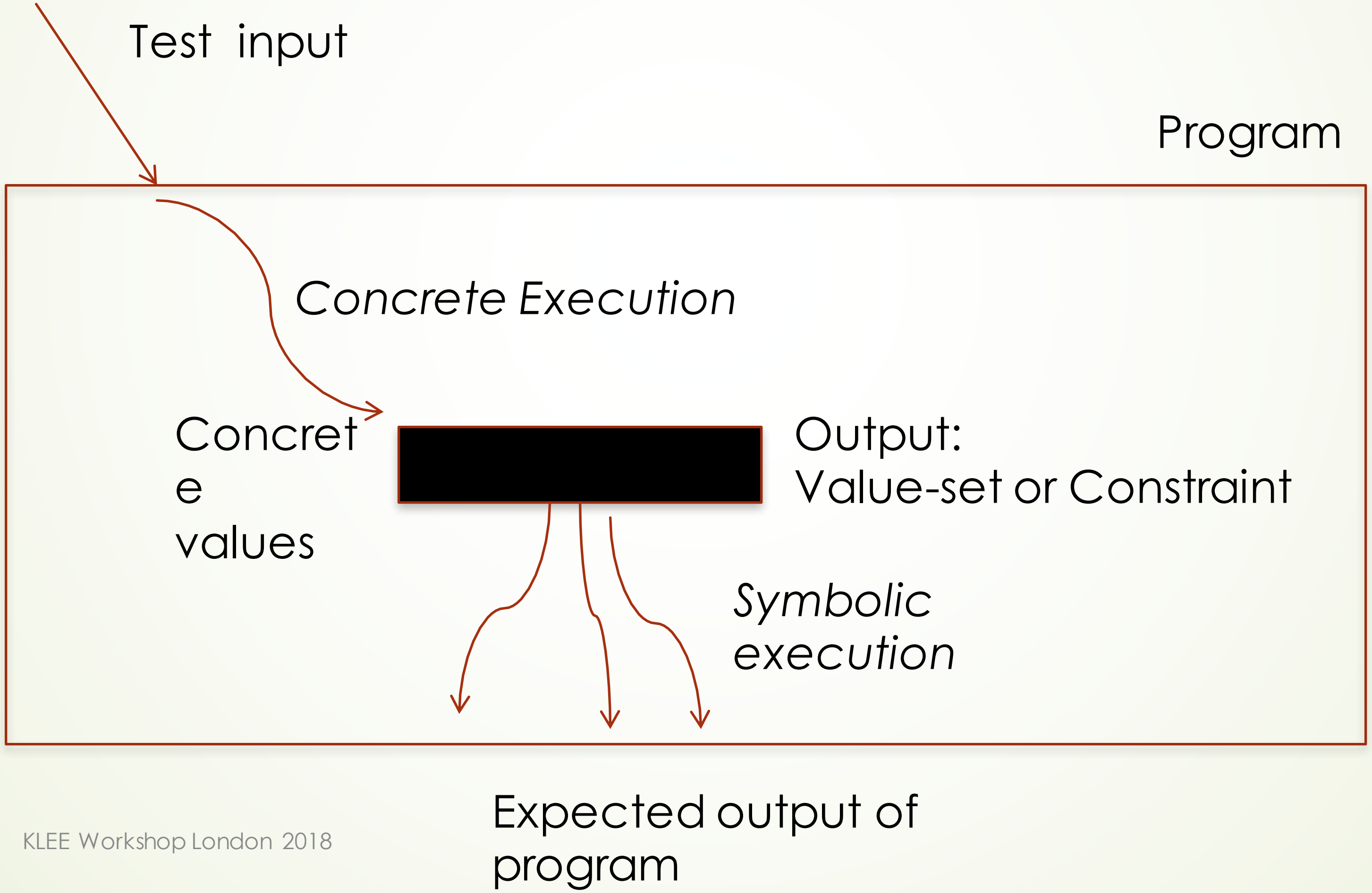
Symbolic Execution

$f(1,11,110) > 110$

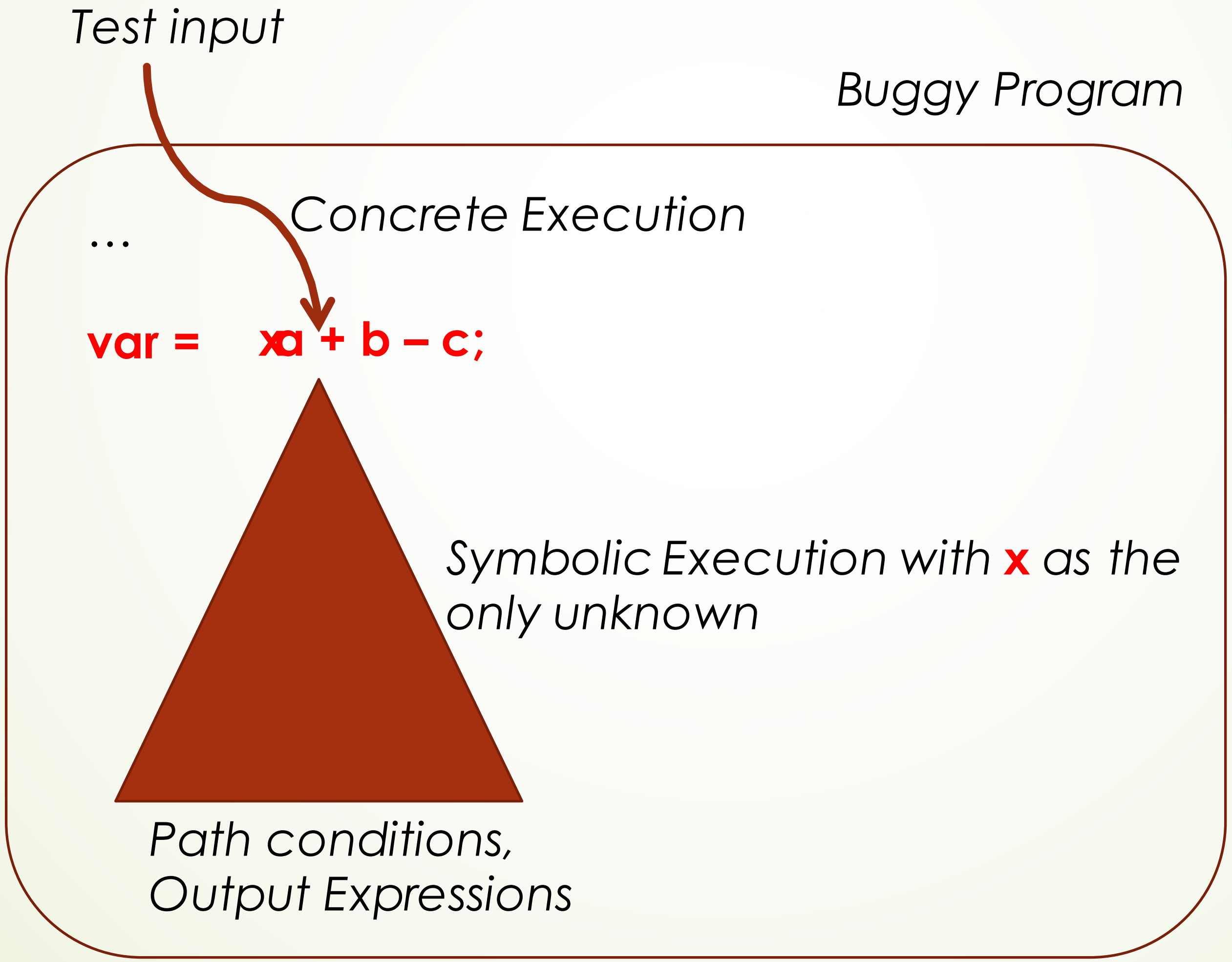
Patch synthesis

- Accumulated constraints
  - *f(1,11, 110) > 110* $\wedge$
  - *f(1,0,100) ≤ 100* $\wedge$
  - …
- Find a f satisfying this constraint
  - By fixing the set of operators appearing in f
- Candidate methods
  - Search over the space of expressions
  - Program synthesis with fixed set of operators ✔
- Generated fix
  - `f(inhibit,up_sep,down_sep) = up_sep + 100`

Semantic Repair



High-level view



x = f(Live Vars)

Get properties of function f via symbolic execution.

Construct a function f which satisfies these properties!

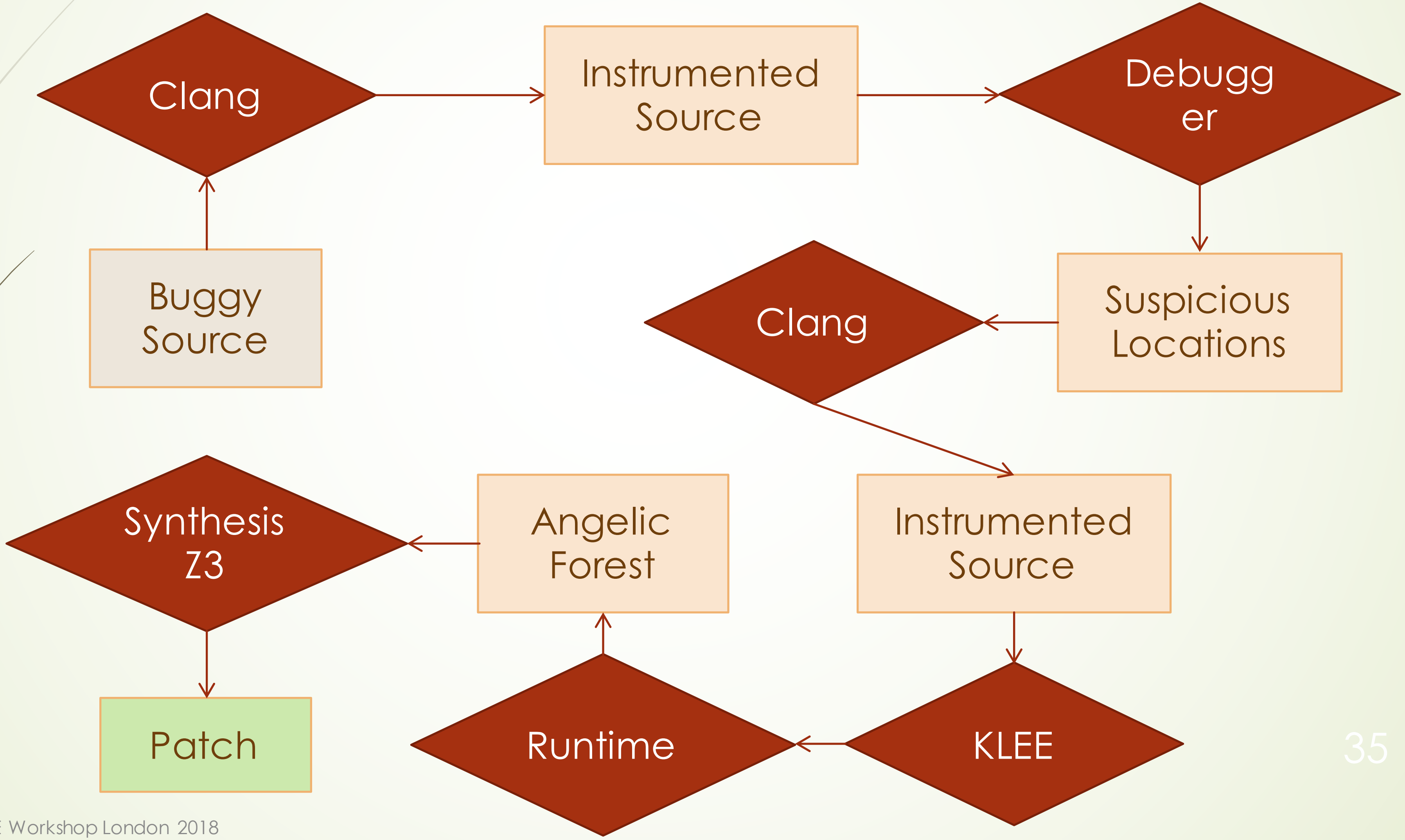
Program Repair given tests

- Generate –and-test patches **(GenProg, *CMU/Michigan*)**

- **Specification inference and patch synthesis**
  - Infer specification or properties about the patch to be synthesized.
  - Meet the specification by enumeration, or by solving constraints.
  - Various works – **SemFix, Angelix (*NUS*), Nopol (*KTH*), SPR (*MIT*), …**

- Ordering of search space of patches
  - Use minimality to *prioritize* the search space. *(NUS)*
  - Use learning approaches to *prioritize* the search space. *(MIT)*
    - Patch templates can be learnt from human fixes. *(HKUST)*

http://angelix.io [ICSE13,16]



State-of-the technology

Subject	LoC	Repair time (min)
wireshark	2814K	23
php	1046K	62
gzip	491K	4
gmp	145K	14
libtiff	77K	14

Defect	Fixed Expressions
Libtiff-4a24508-cc79c2b	2
Libtiff-829d8c4-036d7bb	2
CoreUtils-00743a1f-ec48bead	3
CoreUtils-1dd8a331-d461bfd2	2
CoreUtils-c5ccf29b-a04ddb8d	3

Scalability
Quality: Less functionality-deleting repair than any other tool.

Heartbleed

```
if ( hbtype == TLS1 HB REQUEST) {
    . . .
    memcpy (bp , pl , payload ) ;
    . . .
}
```

(a) The buggy part of the Heartbleed-vulnerable OpenSSL

```
if ( hbtype == TLS1 HB REQUEST
    && payload + 18 < s->s3->rrec.length) {
    . . .
}
```

(b) A fix generated automatically

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
        return 0;
. . .
if ( hbtype == TLS1_HB_REQUEST) {
    . . .
}
else if ( hbtype == TLS1_HB_RESPONSE) {
    . . .
}
return 0;
```

(c) The developer-provided repair

Use a reference implementation

User-define condition: length = 3 & a[0] < a[1] < a[2]

```
int search(int x, int a[], int length) {
  int i;
  for (i=0; i<length; i++) {
    if (x == a[i])
      return i;
  }
  return −1;
}
```

(a) Correct linear search

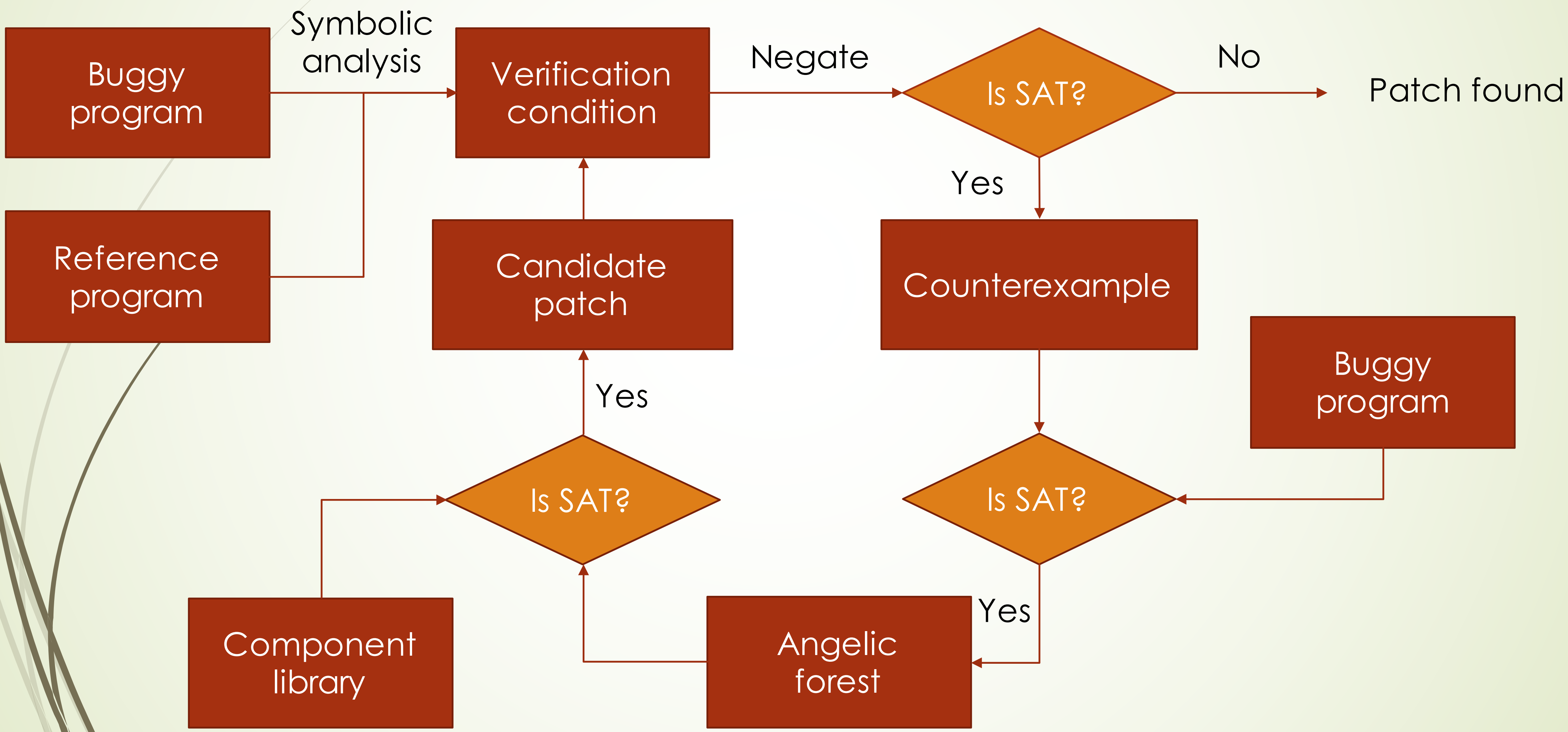
```
int search(int x, int a[], int length) {
  int L = 0;
  int R = length-1;
  do {
    int m = (L+R)/2;
    if (x == a[m]) {
      return m;
    } else if (x < a[m]) { // bug fix: x > a[m]
      L = m+1;
    } else {
      R = m-1;
    }
  } while (L
  return -1;
}
```

(b) Buggy binary search

Verification condition

Experiments on embedded Linux Busybox

SemGraft (ICSE18)



SemGraft results

GNU Coreutils as reference

Program	Commit	Bug	Angelix	SemGraft
sed	c35545a	Handle empty match	Correct	Correct
seq	f7d1c59	Wrong output	Correct	Correct
sed	7666fa1	Wrong output	Incorrect	Correct
sort	d1ed3e6	Wrong output	Incorrect	Correct
seq	d86d20b	Don't accepts 0	Incorrect	Correct
sed	3a9365e	Handle s///	Incorrect	Correct

Linux Busybox as reference

Program	Commit	Bug	Angelix	SemGraft
mkdir	f7d1c59	Segmentation fault	Incorrect	Correct
mkfifo	cdb1682	Segmentation fault	Incorrect	Correct
mknod	cdb1682	Segmentation fault	Incorrect	Correct
copy	f3653f0	Failed to copy a file	Correct	Correct
md5sum	739cf4e	Segmentation fault	Correct	Correct
cut	6f374d7	Wrong output	Incorrect	Correct

Novel applications

Use program repair in intelligent tutoring systems to give the students' individual attention.

Study in IIT-Kanpur (FSE17)

KLEE Workshop London 2018

Acknowledgments

Co-authors:

Umair Ahmed & Amey Karkare (IIT-K)
Marcel Boehme (Monash),
Satish Chandra (Facebook),
Lars Grunske & Yannic Noller(Humboldt)
Sergey Mechtaev (NUS)
HDT Nguyen and Dawei Qi
Manh-Dung Nguyen & **Van-Thuan Pham** (NUS)
Mukul Prasad & Hiroaki Yoshida (Fujitsu),
Shin Hwei Tan (SUSTech)
Jooyong Yi (Innopolis)

Grants:

NRF NCR program TSUNAMi project (2015-2020)

DSO grant (2013-15).

Airbus grant (2017-18).

Relevant papers:
http://www.comp.nus.edu.sg/~abhik/projects/Repair/index.html
http://www.comp.nus.edu.sg/~abhik/projects/Fuzz/

Vulnerability detection and patching

Search

- Enhance the effectiveness of search techniques, with symbolic execution as inspiration

  - **Enhance coverage**
  - **Achieve directed search**

Symbolic Execution

- Explore capabilities of symbolic execution beyond directed search

- **Specification inference**
- **Program synthesis/ repair**

For more details

Own website

- http://www.comp.nus.edu.sg/~abhik

Project website

- http://www.comp.nus.edu.sg/~tsunami/

Links on Repair
http://www.comp.nus.edu.sg/~abhik/projects/Repair/index.html

Links on Fuzzing
http://www.comp.nus.edu.sg/~abhik/projects/Fuzz/

Let us talk in the reception now, or tomorrow – if you are interested.

Reflections on Symbolic Execution

Reachability of a location. e.g. KATCH, AFLGo

Bug finding. e.g. KLEE, AFLFast

Specification Inference from buggy program e.g. SemFix, Angelix