Symbolic Execution and Fuzz Testing

Prof. Abhik Roychoudhury

National University of Singapore

Thanks to organizers and ISSISP

- Steve Blackburn
- Adrian Herrera
- Tony Hosking
- Shane McGrath and all organizers of the event.

Ack. to former students and grant

Marcel. Boehme, PhD. NUS 2014, Post-doc NUS -> Lecturer Monash

Van Thuan Pham, PhD. 2017

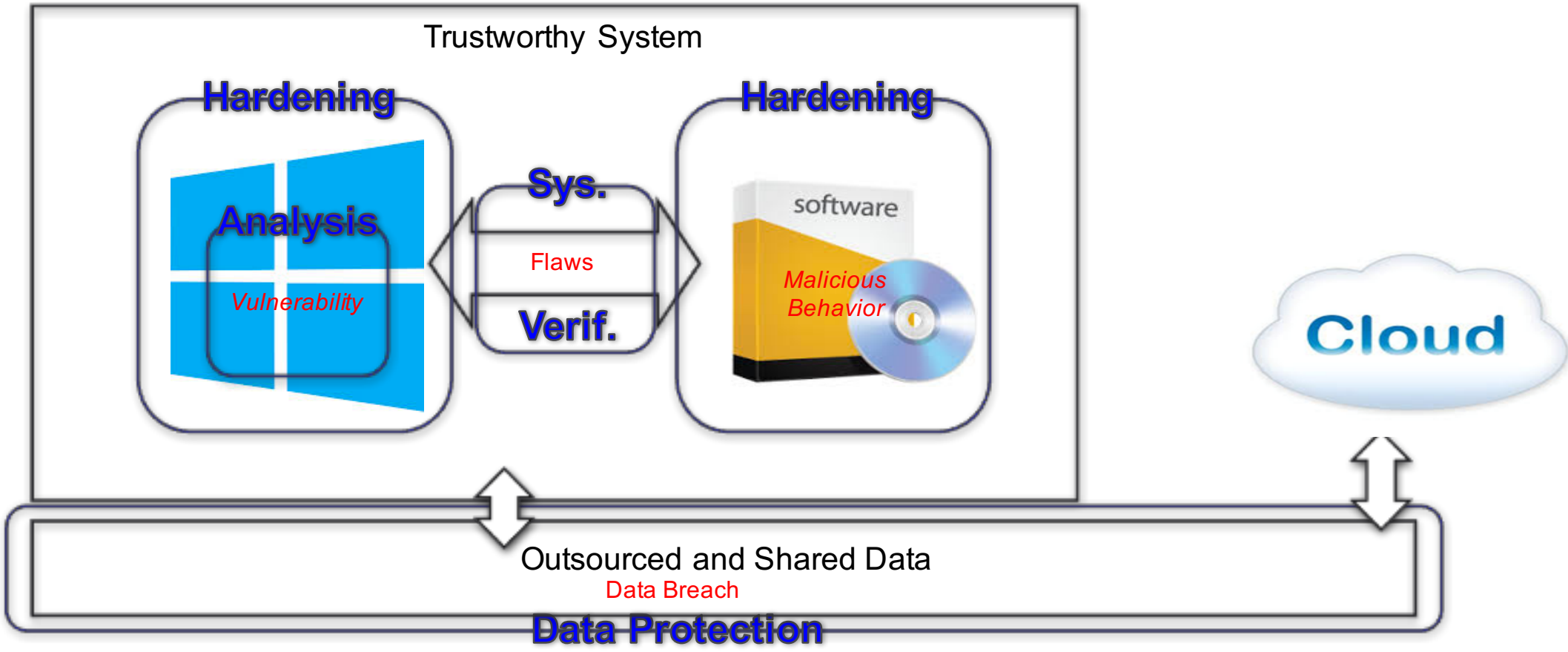
Sergey Mechtaev, PhD. 2018 -> Lecturer University College London

Shin Hwei Tan, PhD. 2018 -> Asst Prof, SUSTech, Shenzen. China

Jooyong Yi, past post-doc -> Asst Prof. Innopolis

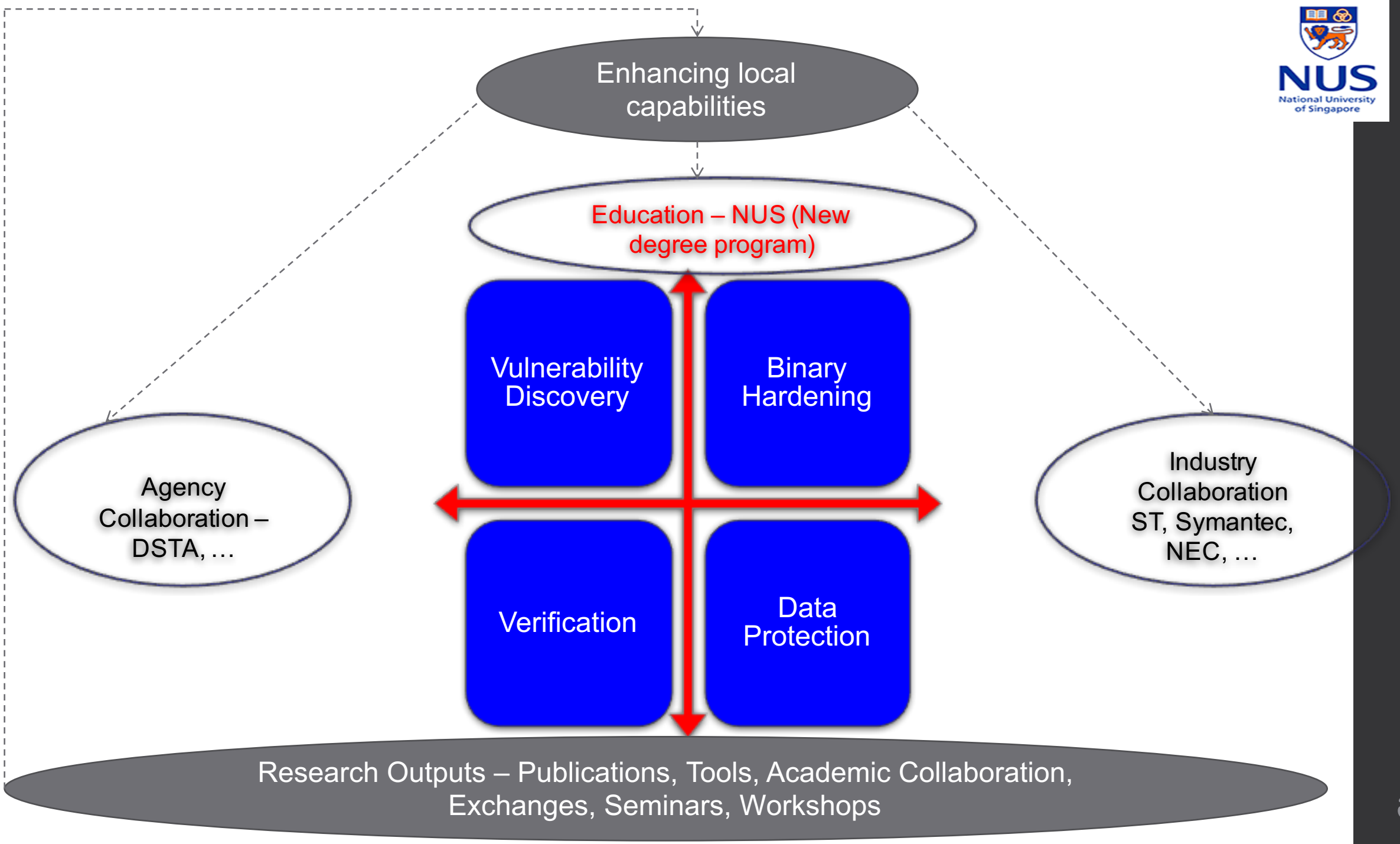
ACKNOWLEDGEMENT: National Cyber Security Research program from NRF Singapore http://www.comp.nus.edu.sg/~tsunami/ and DSO National Labs

COTS-integrated Platforms



Binary analysis of paramount need for software acquisition or assembly.

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


Enhancing local capabilities
Education – NUS (New degree program)
Vulnerability Discovery
Binary Hardening
Verification
Data Protection
Agency Collaboration – DSTA, …
Industry Collaboration ST, Symantec, NEC, …
Research Outputs – Publications, Tools, Academic Collaboration, Exchanges, Seminars, Workshops

Plan

- History of Symbolic execution
  - Symbolic Execution and Program Testing
- Use in fuzz testing
- Lead up to specification inference
- How the ideas of symbolic execution can be transported to automated program repair

Short Videos

- https://youtu.be/C1hl_ujw6B0
- (1 Minute)

- https://youtu.be/EHBjMSQvIpg
- (1 Minute)

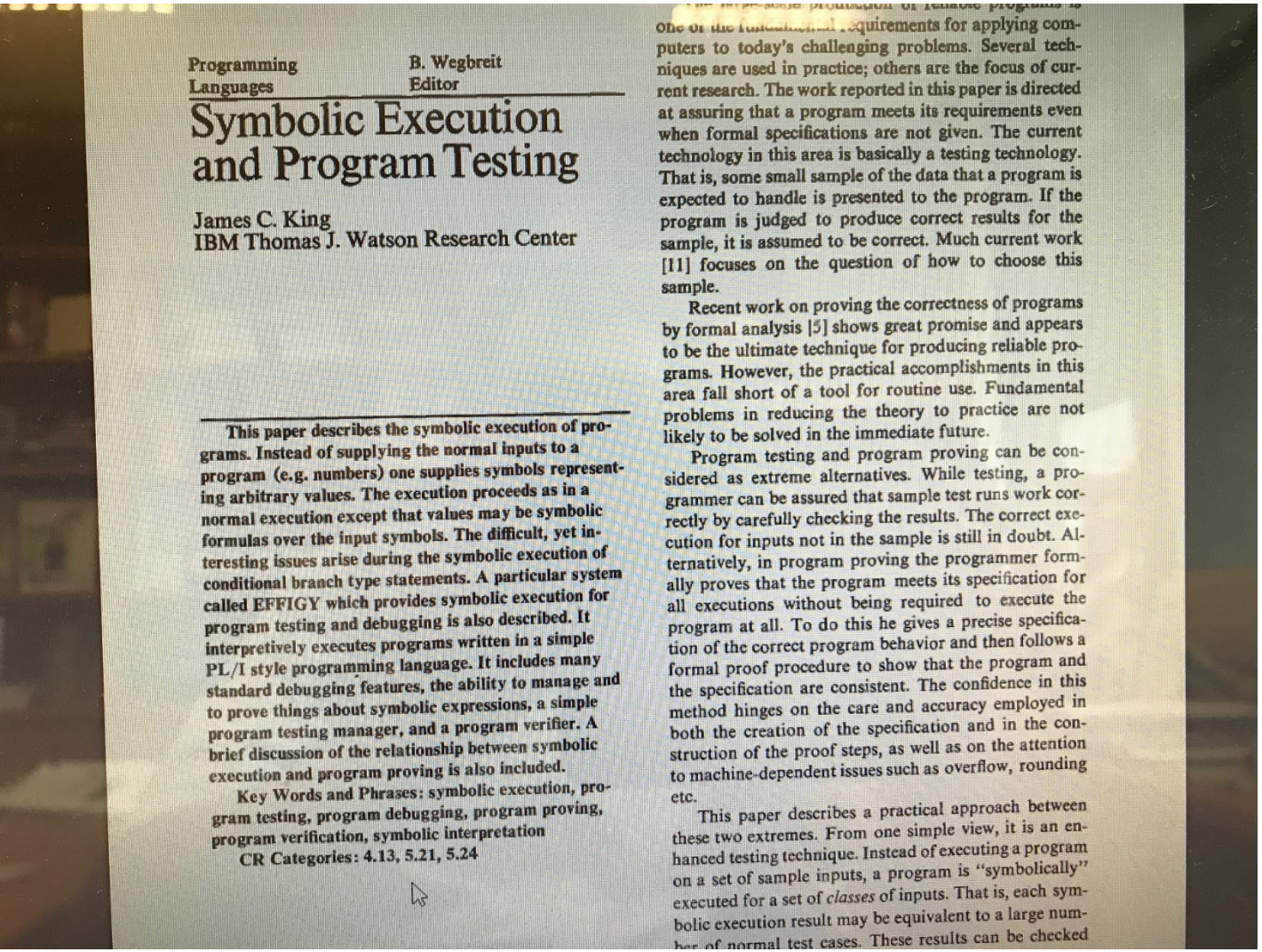
In this(?) talk …

Search

- Enhance the effectiveness of search techniques, with symbolic execution as inspiration
- **Systematic Fuzz Testing**

Symbolic Execution

- Explore capabilities of symbolic execution beyond search
- **Automated Program Repair**



Programming Languages

B. Wegbreit Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

This paper describes the symbolic execution of programs. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. The difficult, yet interesting issues arise during the symbolic execution of conditional branch type statements. A particular system called EFFIGY which provides symbolic execution for program testing and debugging is also described. It interpretively executes programs written in a simple PL/I style programming language. It includes many standard debugging features, the ability to manage and to prove things about symbolic expressions, a simple program testing manager, and a program verifier. A brief discussion of the relationship between symbolic execution and program proving is also included.

Key Words and Phrases: symbolic execution, program testing, program debugging, program proving, program verification, symbolic interpretation
CR Categories: 4.13, 5.21, 5.24

...quirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

Recent work on proving the correctness of programs by formal analysis [5] shows great promise and appears to be the ultimate technique for producing reliable programs. However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved in the immediate future.

Program testing and program proving can be considered as extreme alternatives. While testing, a programmer can be assured that sample test runs work correctly by carefully checking the results. The correct execution for inputs not in the sample is still in doubt. Alternatively, in program proving the programmer formally proves that the program meets its specification for all executions without being required to execute the program at all. To do this he gives a precise specification of the correct program behavior and then follows a formal proof procedure to show that the program and the specification are consistent. The confidence in this method hinges on the care and accuracy employed in both the creation of the specification and in the construction of the proof steps, as well as on the attention to machine-dependent issues such as overflow, rounding etc.

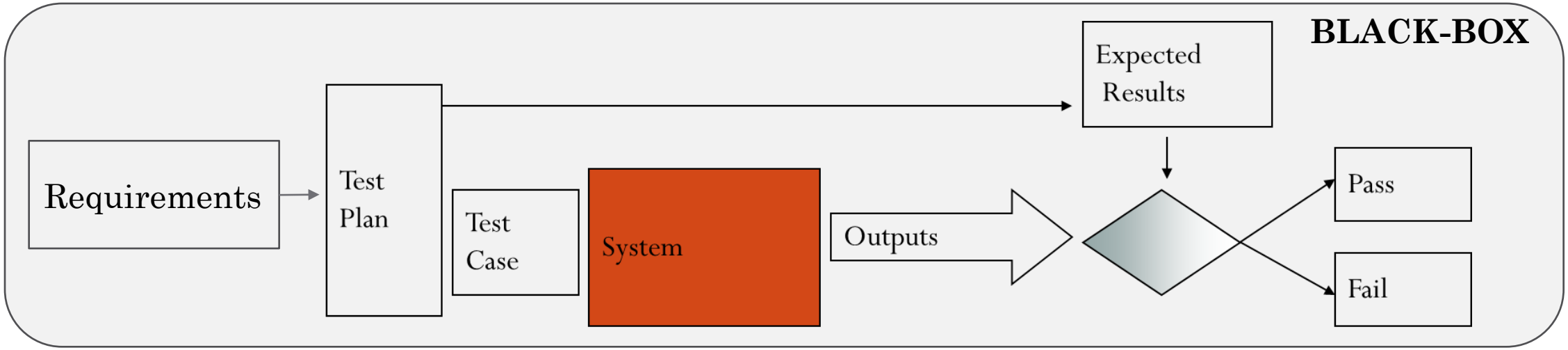
This paper describes a practical approach between these two extremes. From one simple view, it is an enhanced testing technique. Instead of executing a program on a set of sample inputs, a program is "symbolically" executed for a set of *classes* of inputs. That is, each symbolic execution result may be equivalent to a large number of normal test cases. These results can be checked

"Program testing and program proving can be considered as extreme alternatives. ....

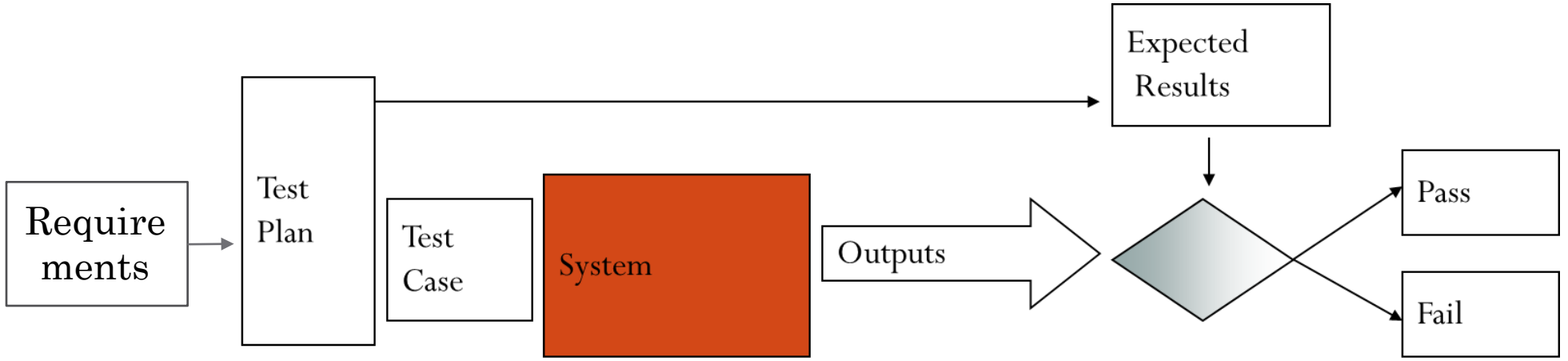
This paper describes a practical approach between these two extremes ...

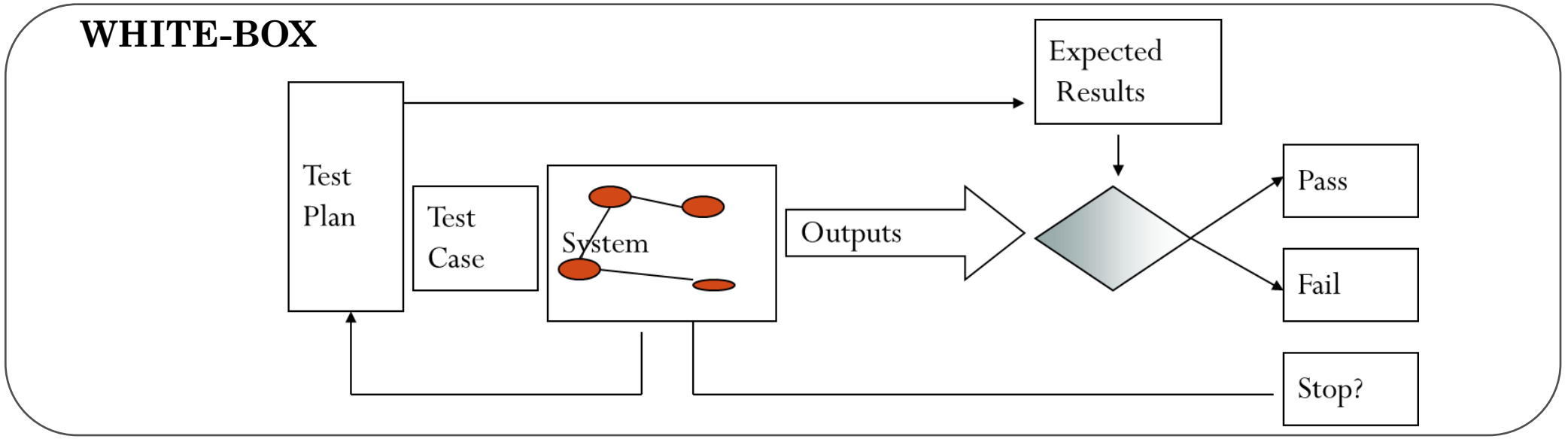
Each symbolic execution result may be equivalent to a large number of normal tests"

Testing

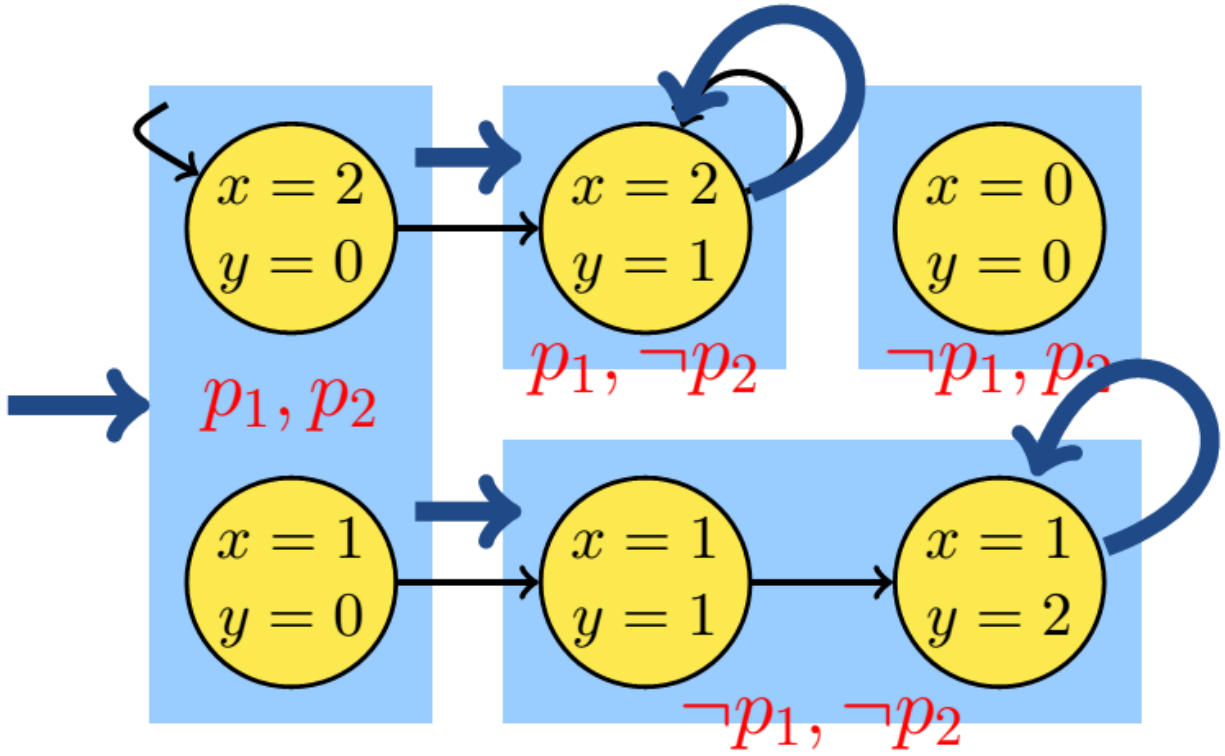

BLACK-BOX
Requirements
Test Plan
Test Case
System
Outputs
Expected Results
Pass
Fail

Testing


Require ments
Test Plan
Test Case
System
Outputs
Expected Results
Pass
Fail


WHITE-BOX
Test Plan
Test Case
System
Outputs
Expected Results
Pass
Fail
Stop?

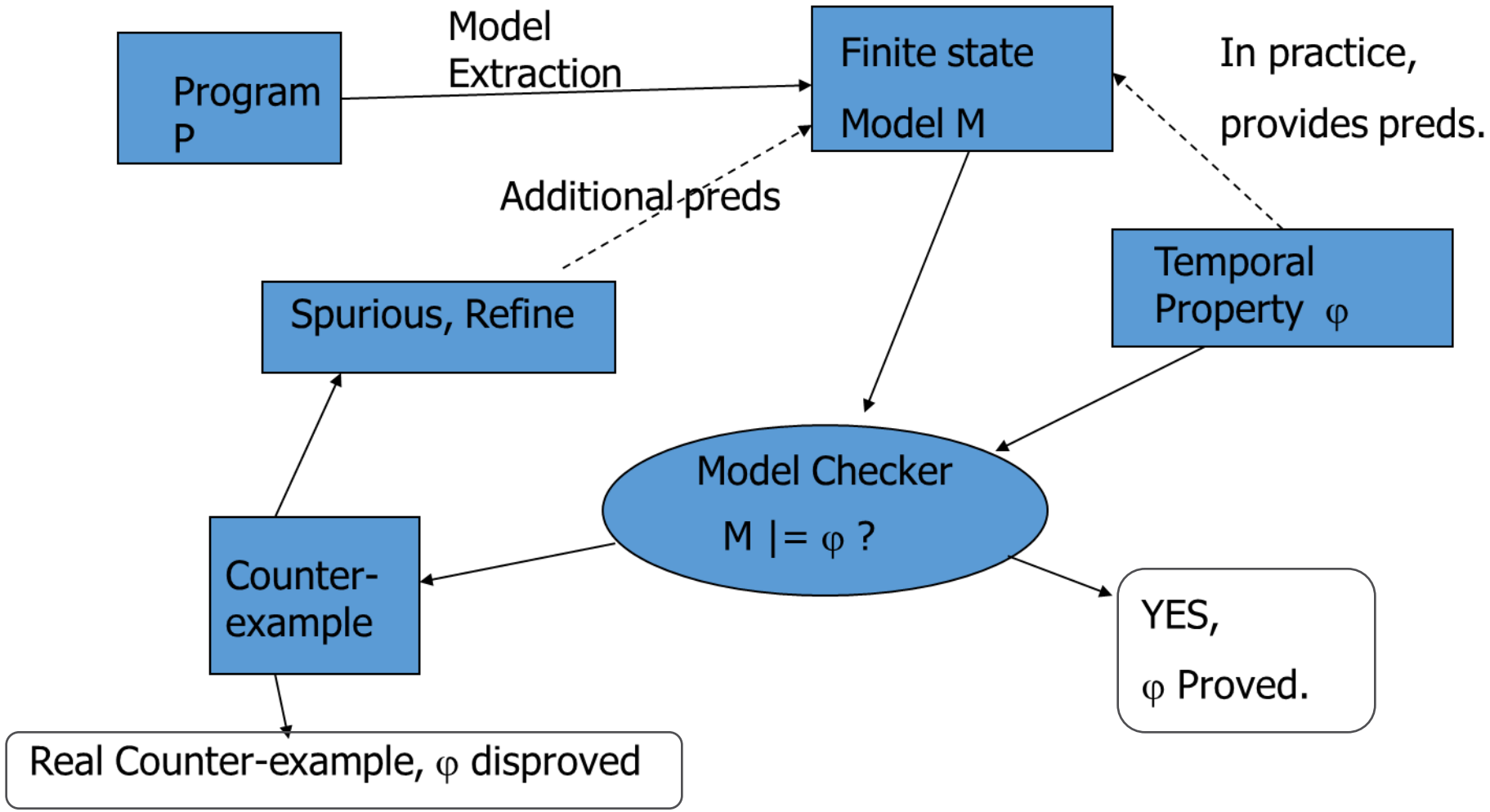
Proving via SW Model Checking

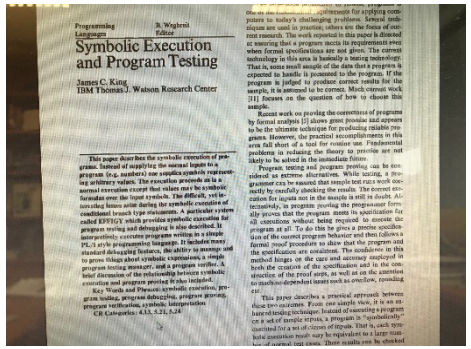


Property:

$$x > y \vee y \neq 0 \quad \Longleftrightarrow \quad p_1 \vee \neg p_2$$

Proving: SW Model Checking




Programming Languages — Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

Blurring the lines: Symbolic Exec.

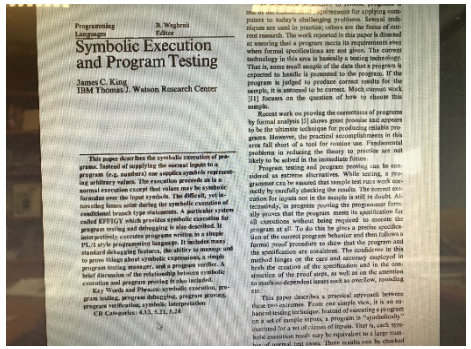
```
SEARCH( A, L, U, X, found, j){

int j, found = 0;
while (L  <= U  && found == 0){
          j = (L+U)/2;
          if (X == A[j]){  found  = 1;}
          else if (X < A[j]){ U = j -1; }
          else{ L = j +1; }
}
if (found == 0){ j =  L - 1;}

}
```

SEARCH(A, 1, 5, X, found, j)

X == A[3]	found == 1 j == 3
X == A[1] && X < A[3]	found == 1 j == 1
X < A[1] && X <A[3]	found == 0 j == 0
X = A[2] && X > A[1] && X <A[3]	found == 1 j == 2

….

Testing ?
Comprehension??
Verification ???


Programming Languages — Editor: B. Wegbreit

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

Blurring the lines: Symbolic Exec.

```
SEARCH( A, L, U, X, found, j){

int j, found = 0;
while (L   <= U   && found == 0){
           j = (L+U)/2;
           if (X == A[j]){  found  = 1;}
           else if (X < A[j]){ U = j -1; }
           else{ L = j +1; }
}
if (found == 0){ j =  L - 1;}

}
```

SEARCH(A, 1, 5, 20, found, j)

SEARCH(A, 1, 5, X, found, j)

SEARCH(A, N, N+4, X, found, j)

SEARCH(A, 1, M, X, found, j)

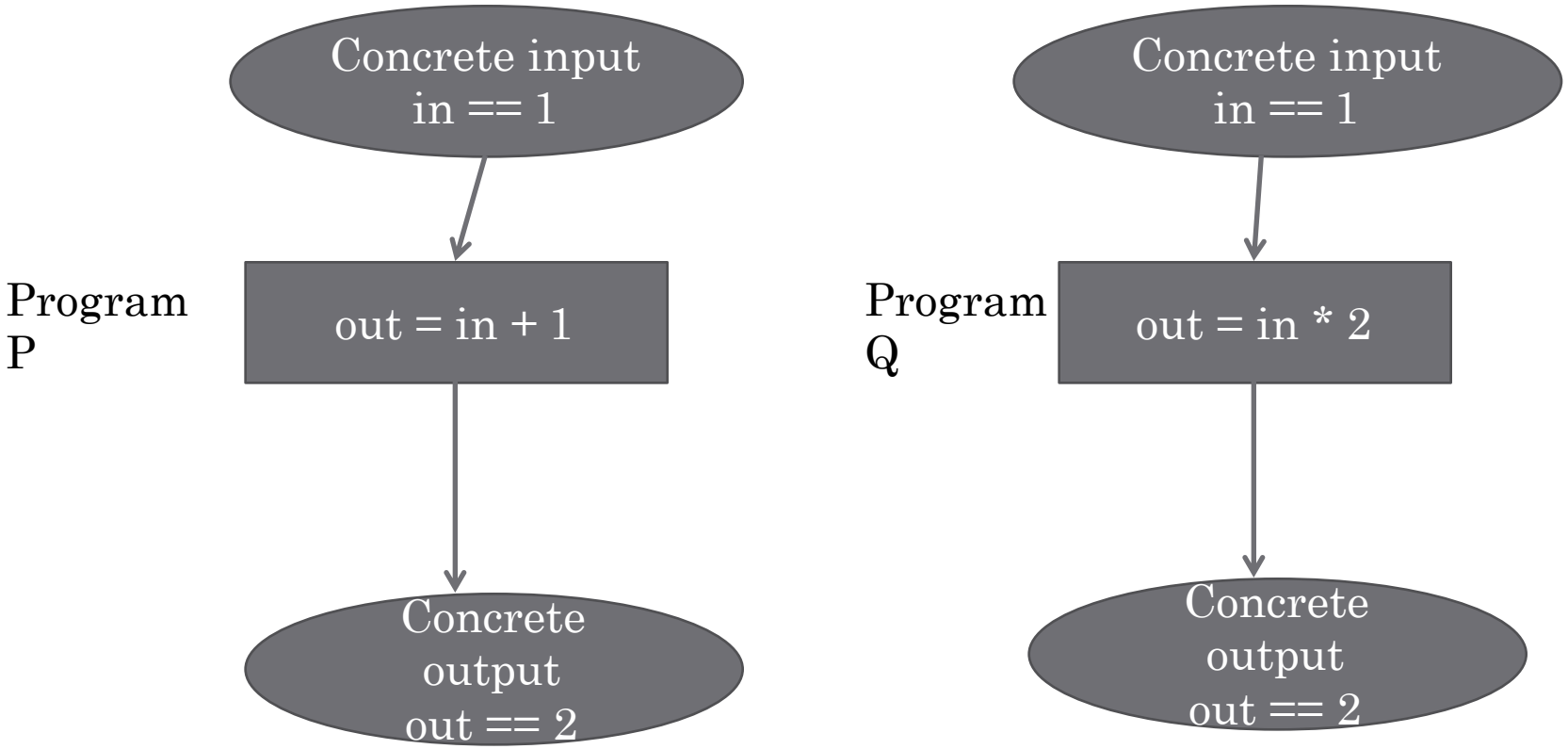
Testing ?
Comprehension??
Verification ???

Primer on SE

Abhik Roychoudhury

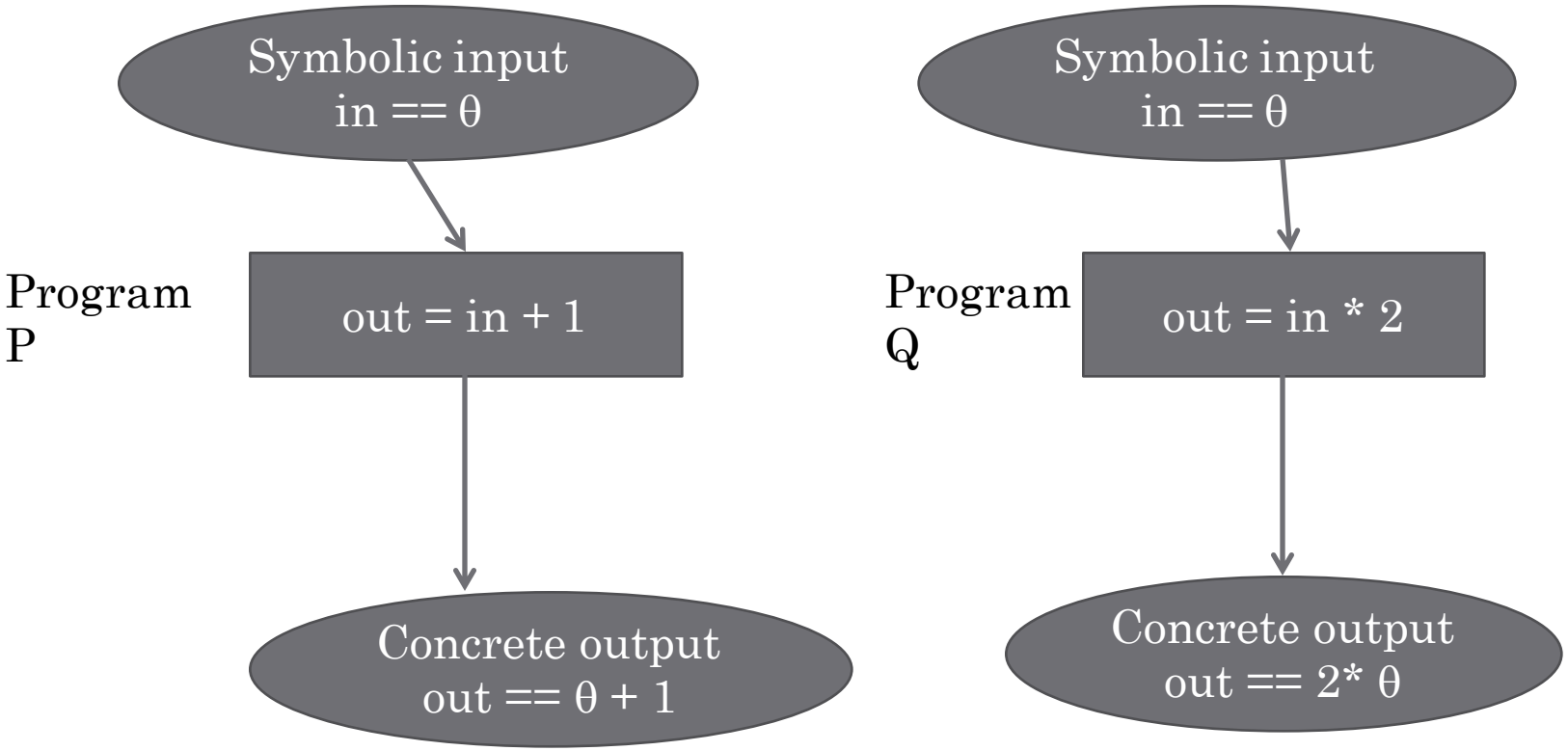
National University of Singapore

Concrete execution



No observable difference!

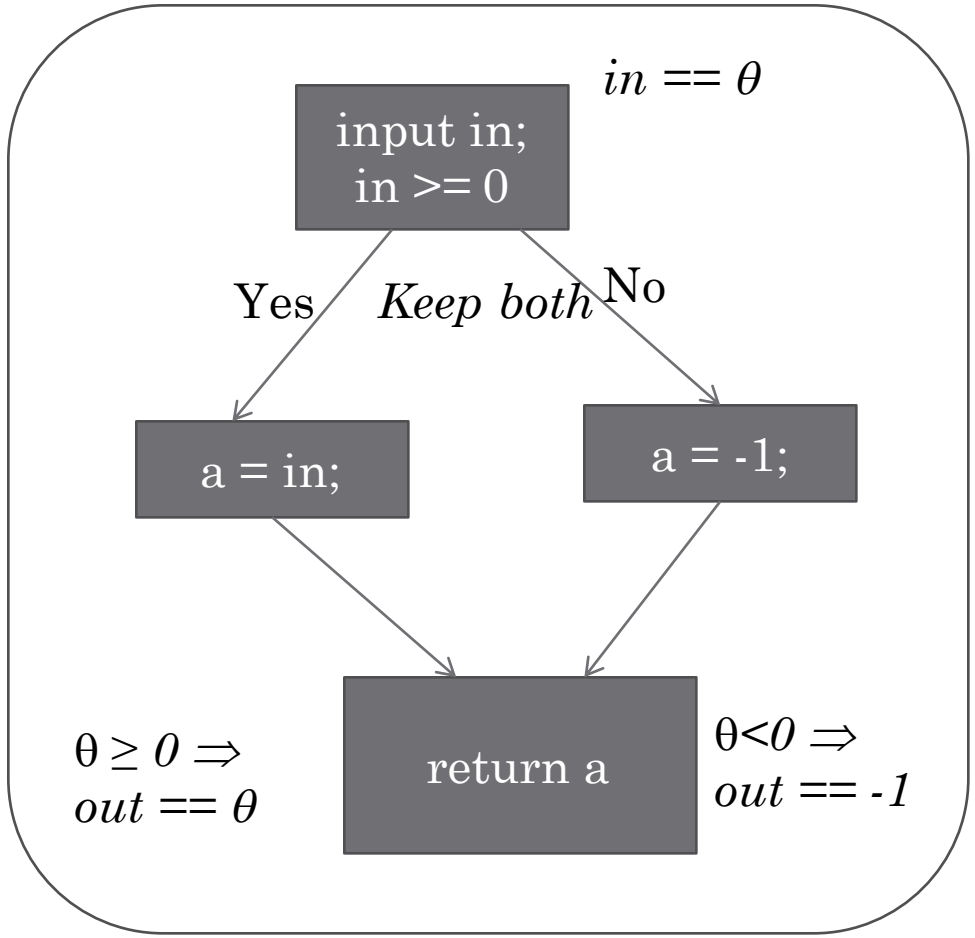
Execution with symbolic inputs



To expose difference, try to find θ such that $\theta + 1 \neq 2 * \theta$

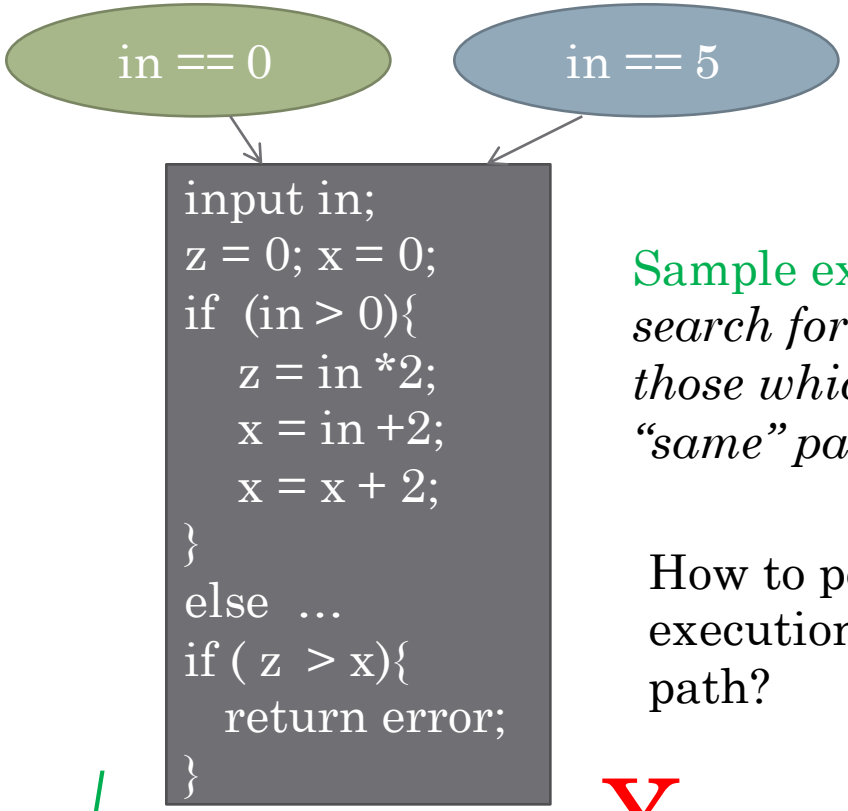
Path exploration based symbolic execution

```
input in;

if  (in >= 0)
     a = in;
else
     a = -1;

return a;
```



On-the-fly path exploration

Instead of analyzing the whole program, shift from one program path to another.



Sample exploration: *Continue the search for failing inputs. Try those which do not go through the "same" path.*

How to perform symbolic execution along a single path?

√ X

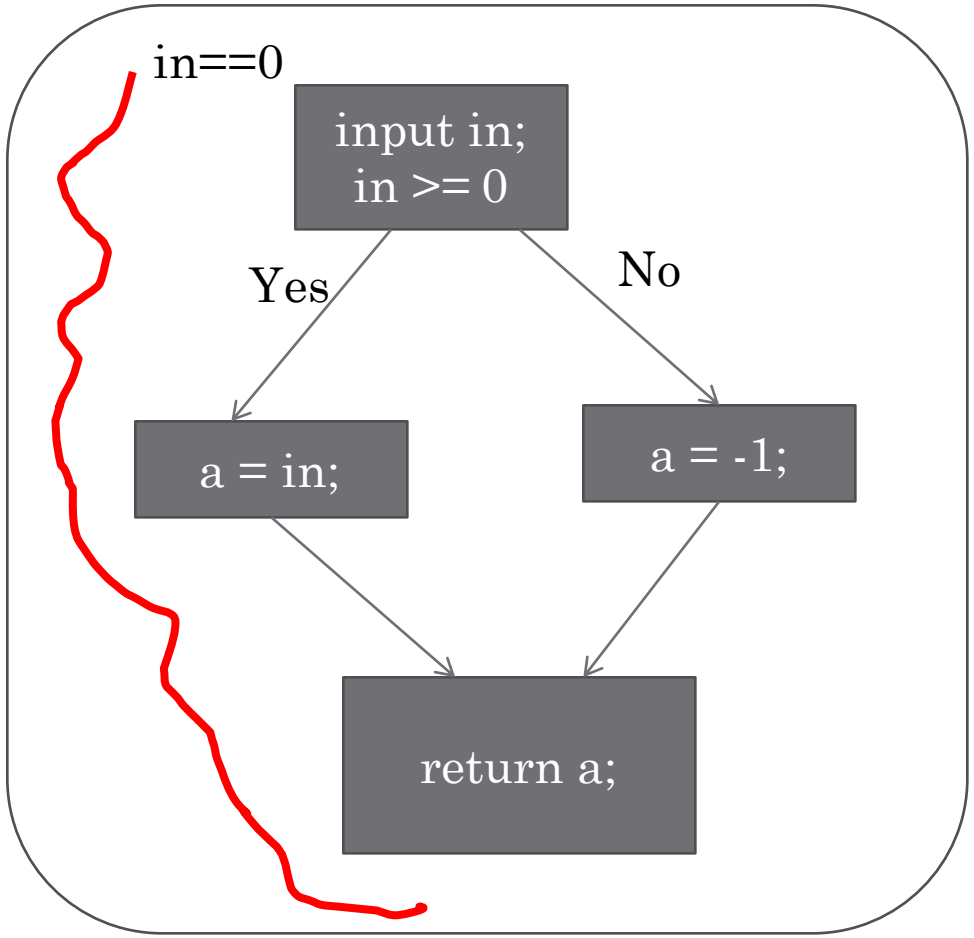
Exploring one path

Useful to find:

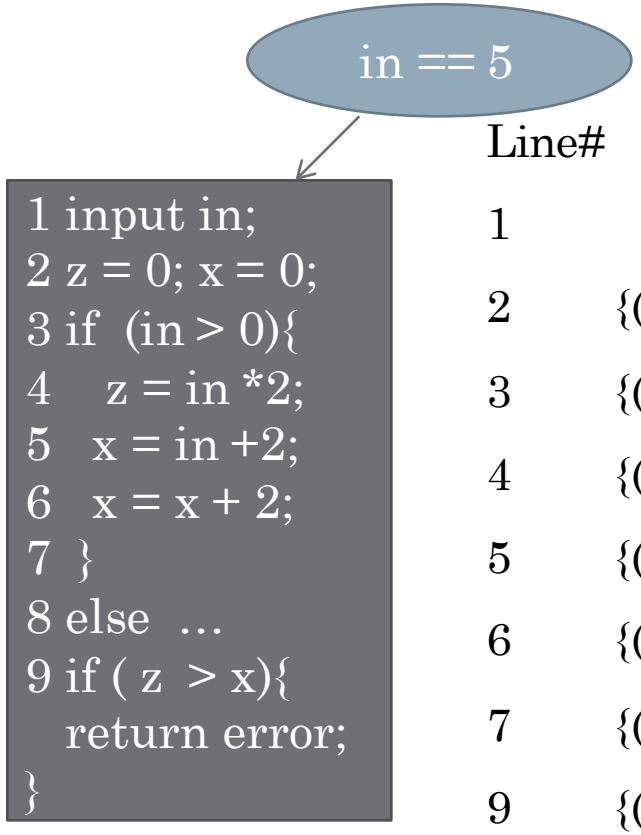
"the set of all inputs which trace a given path"

Path condition

in ≥ 0



Path condition computation

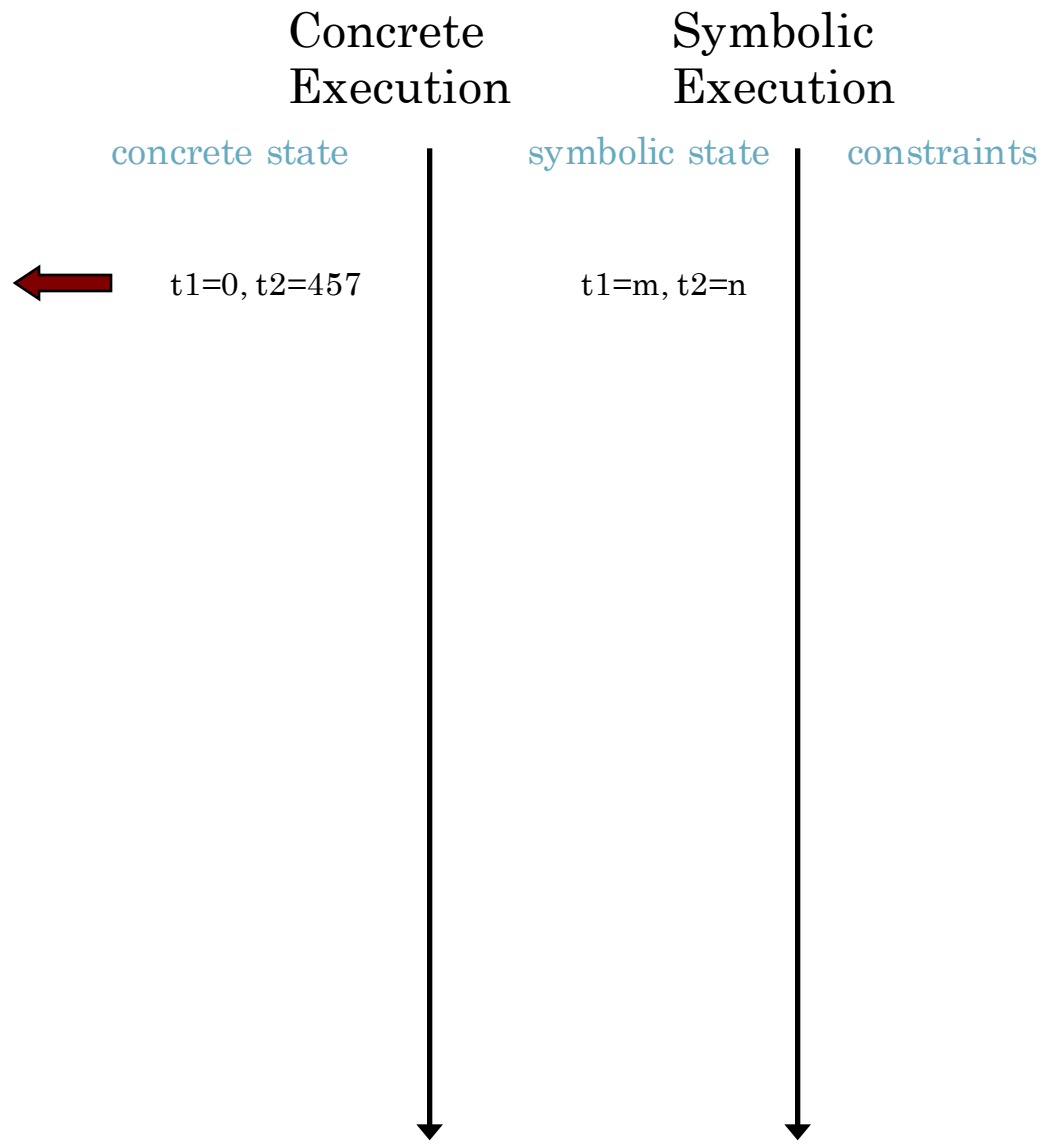


Line#	Assignment store	Path condition
1	{}	*true*
2	{(z,0),(x,0)}	*true*
3	{(z,0),(x,0)}	*in > 0*
4	{(z,2*in), (x,0)}	*in > 0*
5	{(z,2*in), (x,in+2)}	*in > 0*
6	{(z,2*in), (x, in+4)}	*in > 0*
7	{(z, 2*in), (x, in+4)}	*in > 0*
9	{(z, 2*in), (x, in+4)}	*in>0 ∧ (2*in > in +4)*

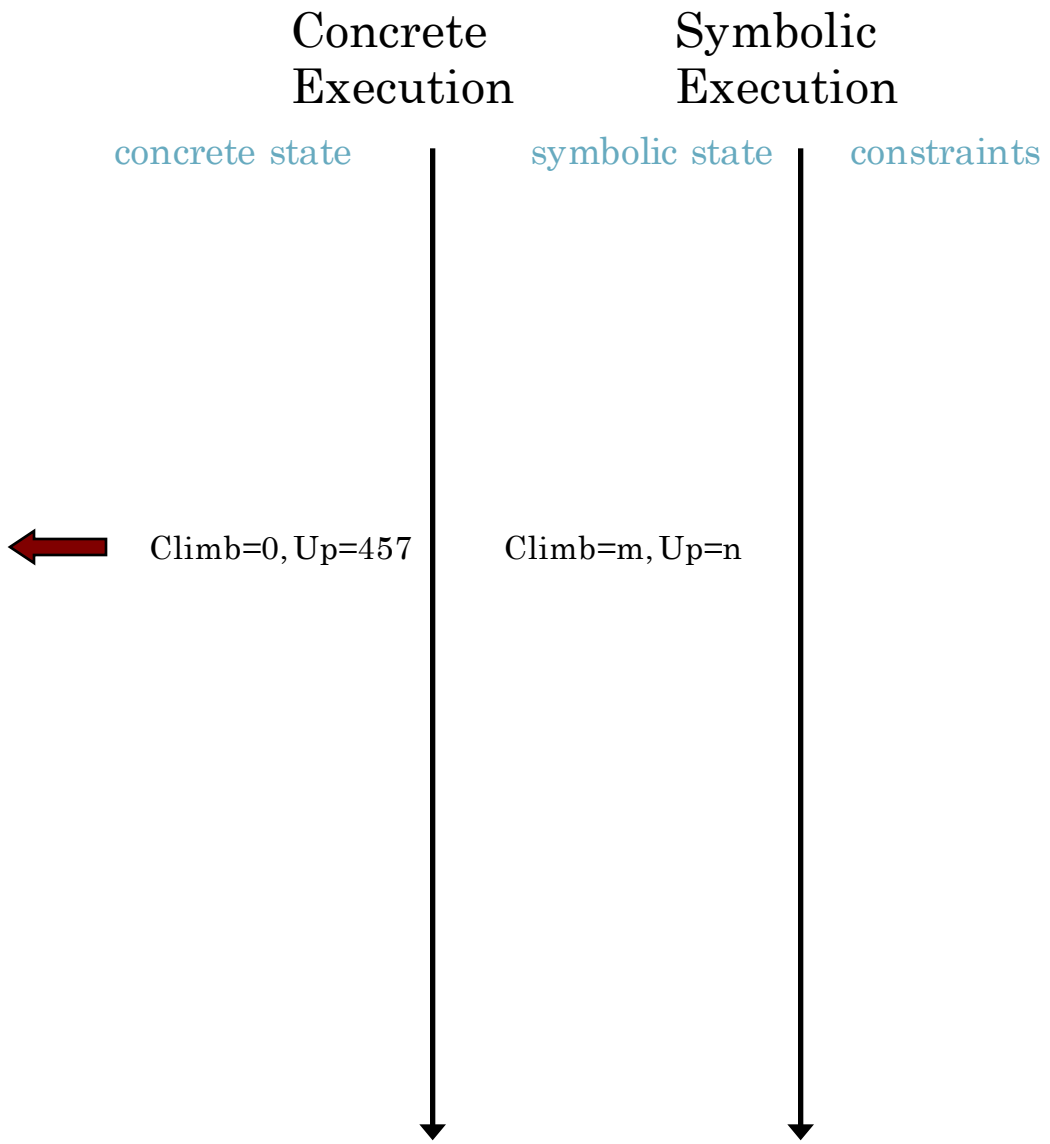
Directed testing

- Start with a random input I.
- Execute program P with I
  - Suppose I executes path p in program P.
  - While executing p, collect a symbolic formula f which captures the set of all inputs which execute path p in program P.
  - **f is the path condition of path p traced by input i.**
- Minimally change f, to produce a formula f1
  - Solve f1 to get a new input I1 which executes a path p1 different from path p.

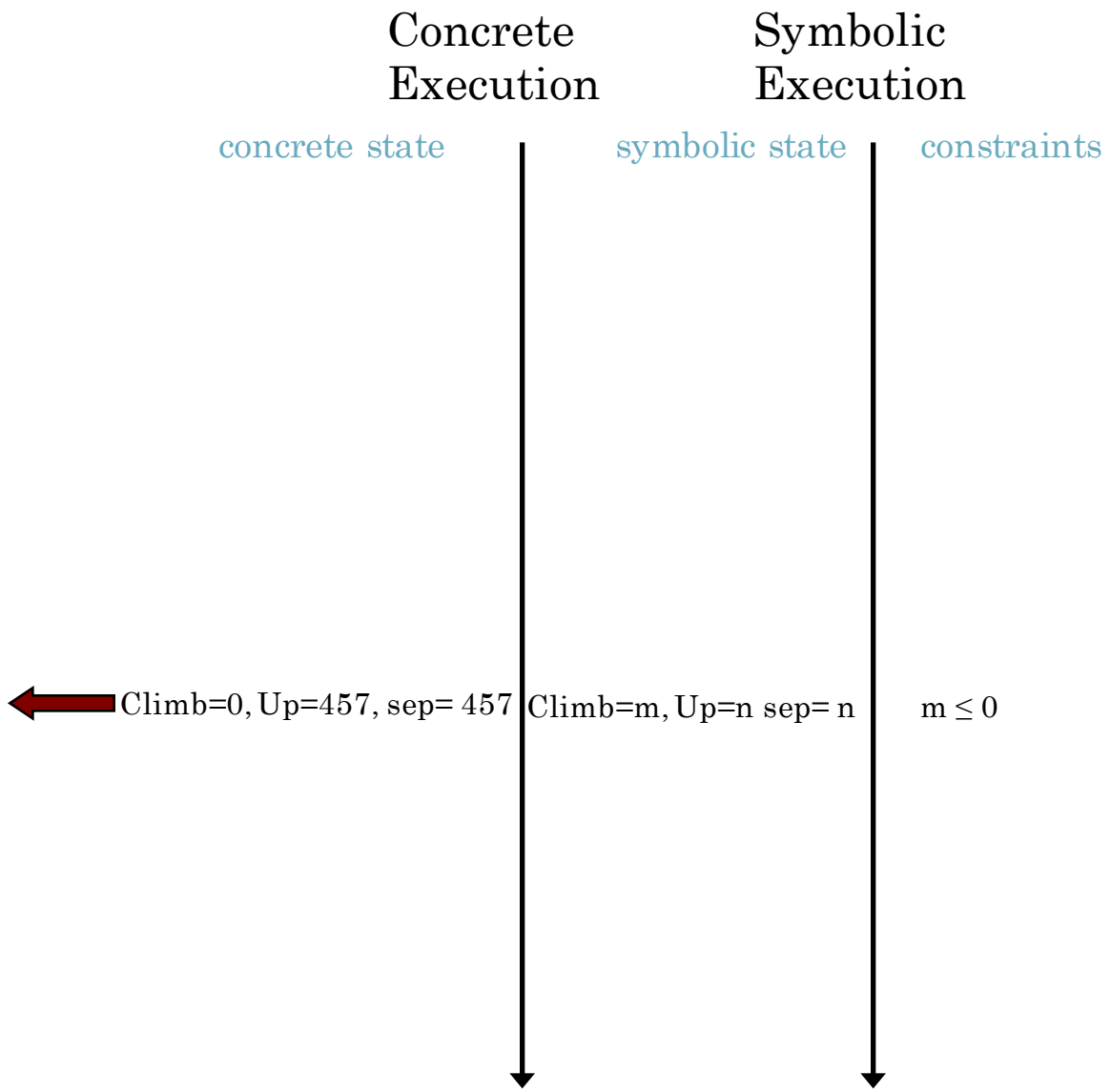
```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
     sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
      upward = 1;
    } else {upward = 0;}
    if (upward < 0){
          abort;
     } else return upward;
}
```



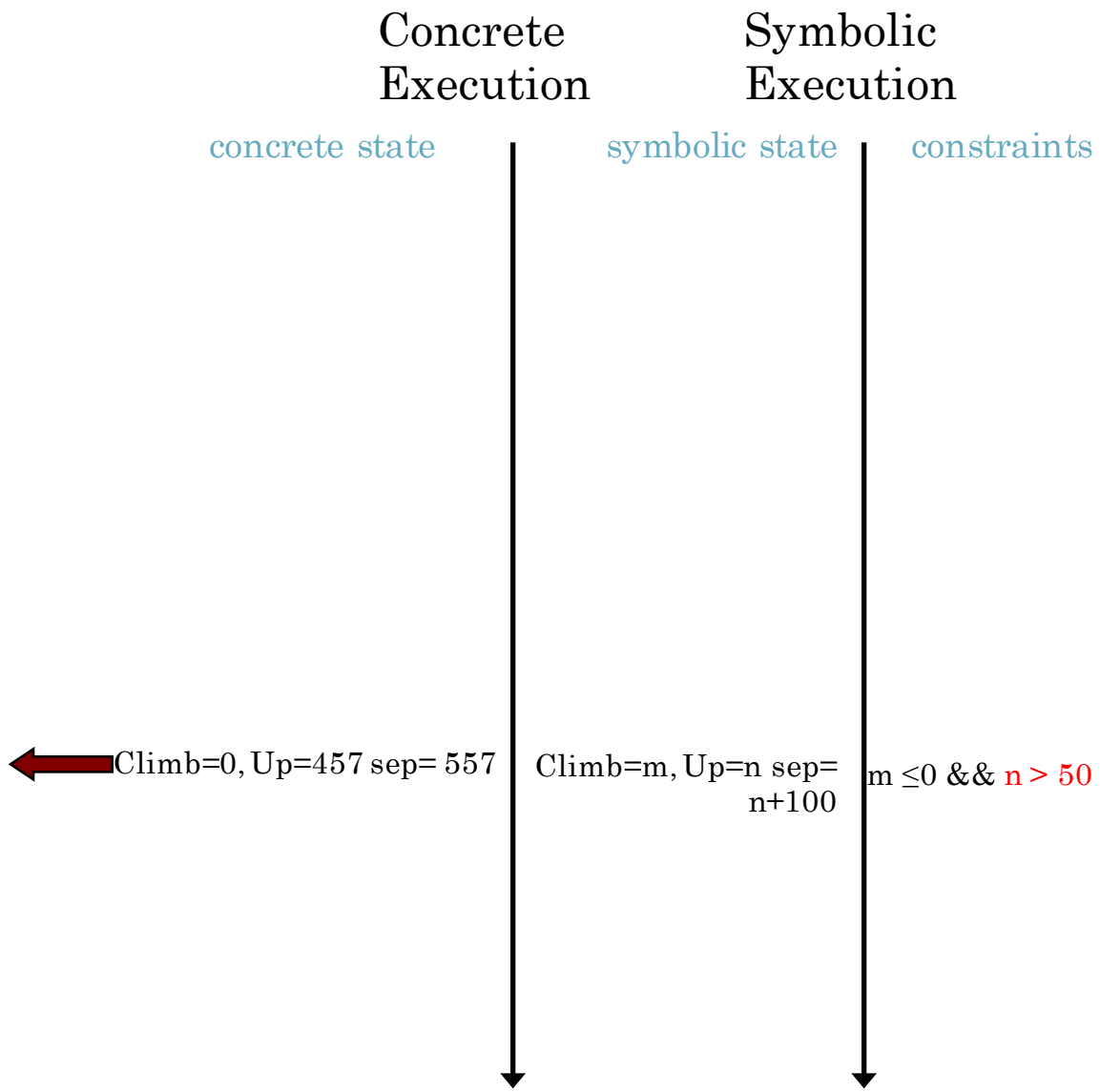
```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
     sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
      upward = 1;
    } else {upward = 0;}
    if (upward < 0){
          abort;
     } else return upward;
}
```



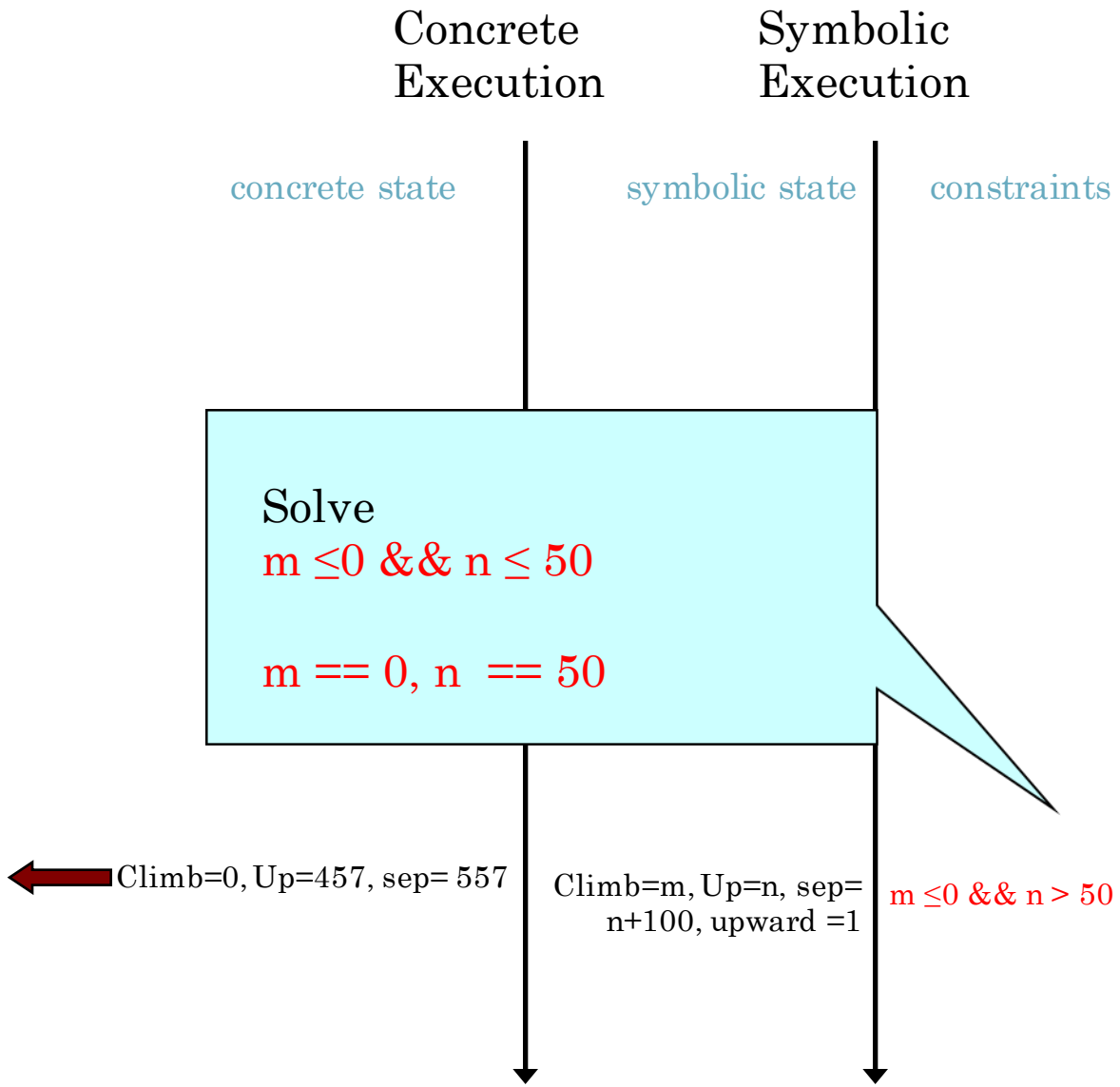
```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
     sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
      upward = 1;
    } else {upward = 0;}
    if (upward < 0){
         abort;
     } else return upward;
}
```



```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb){
     sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
          abort;
     } else return upward;
}
```

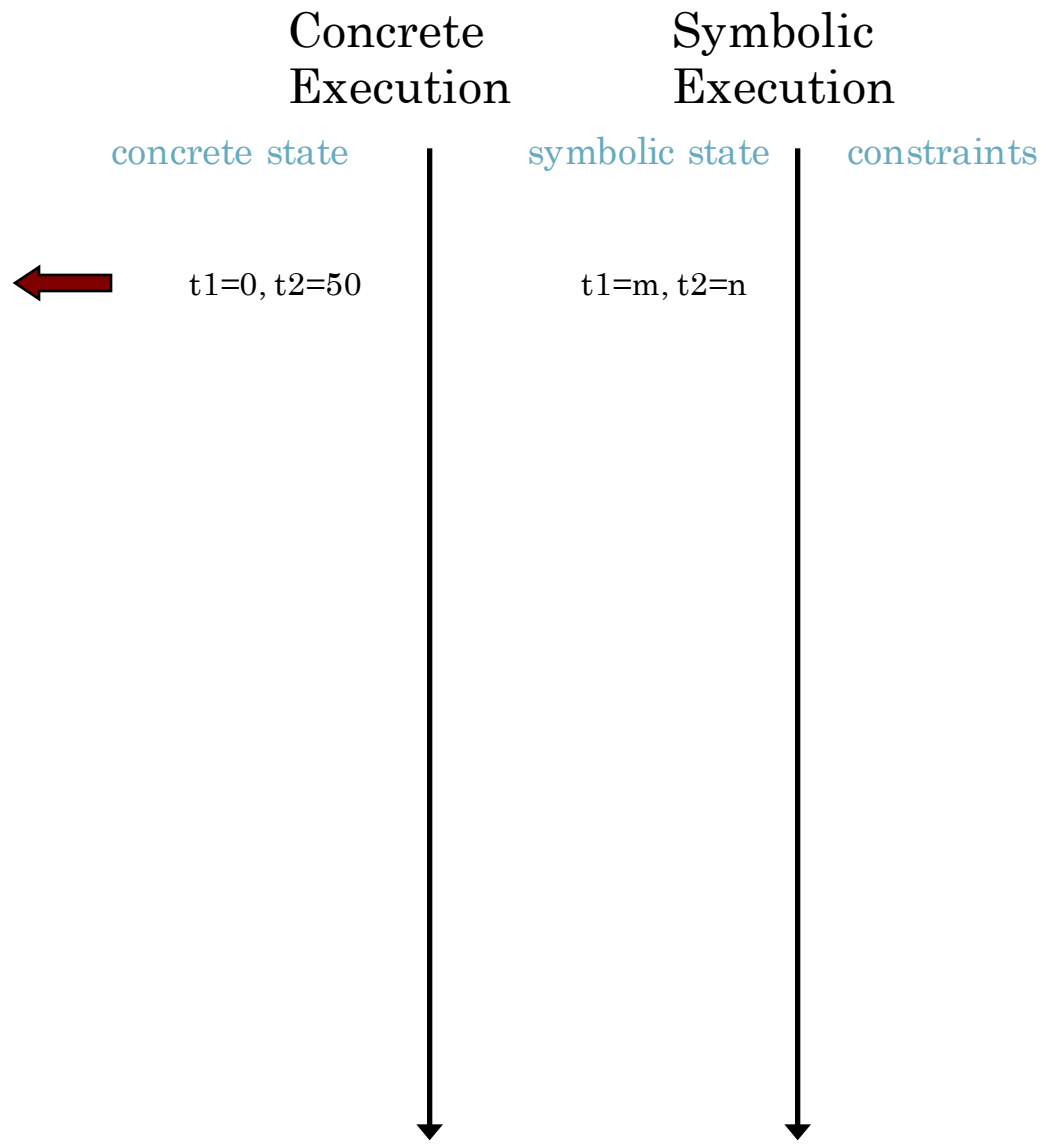


```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb){
      sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
          abort;
     } else return upward;
}
```

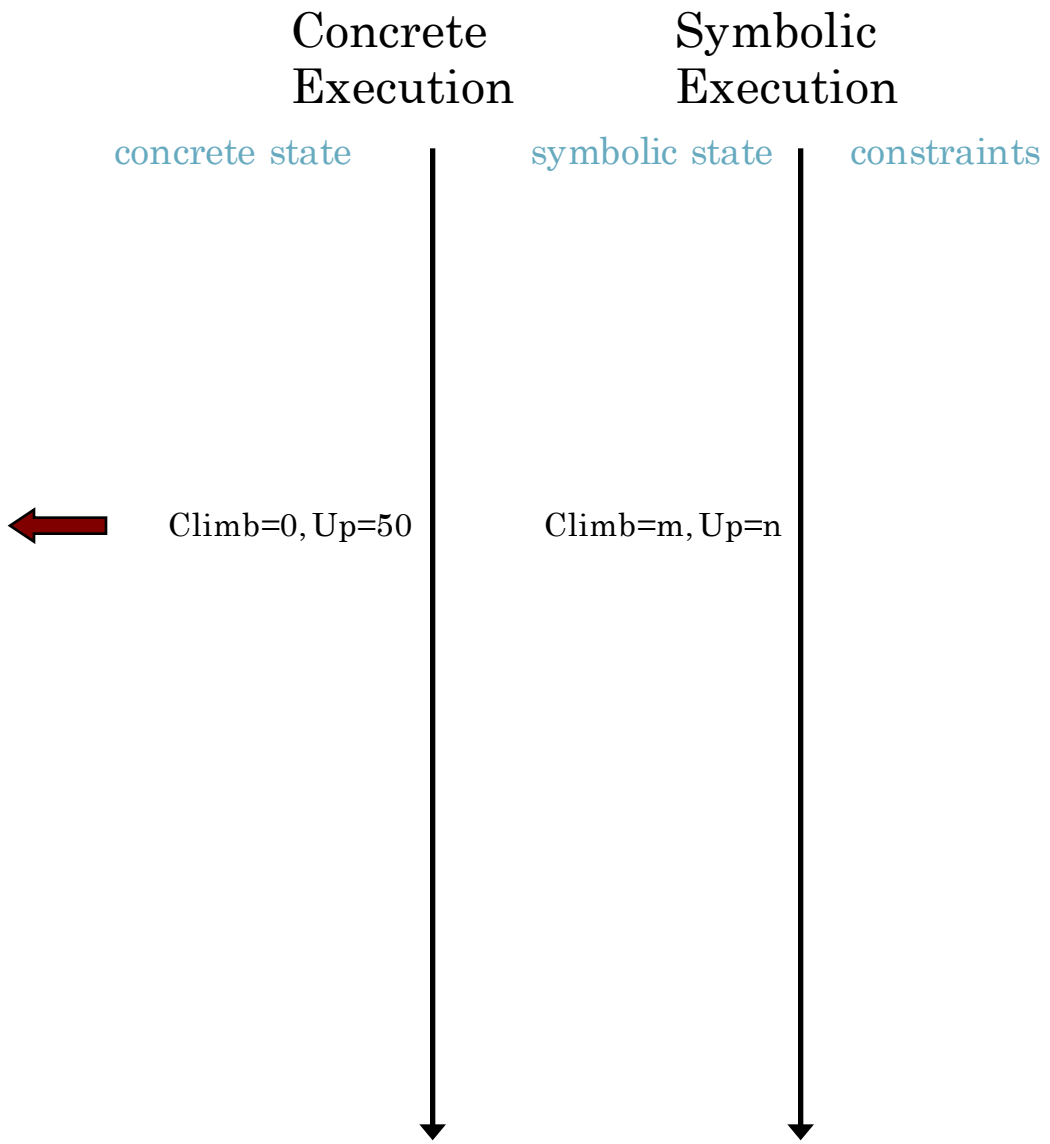


Ack: Koushik Sen (Berkeley)

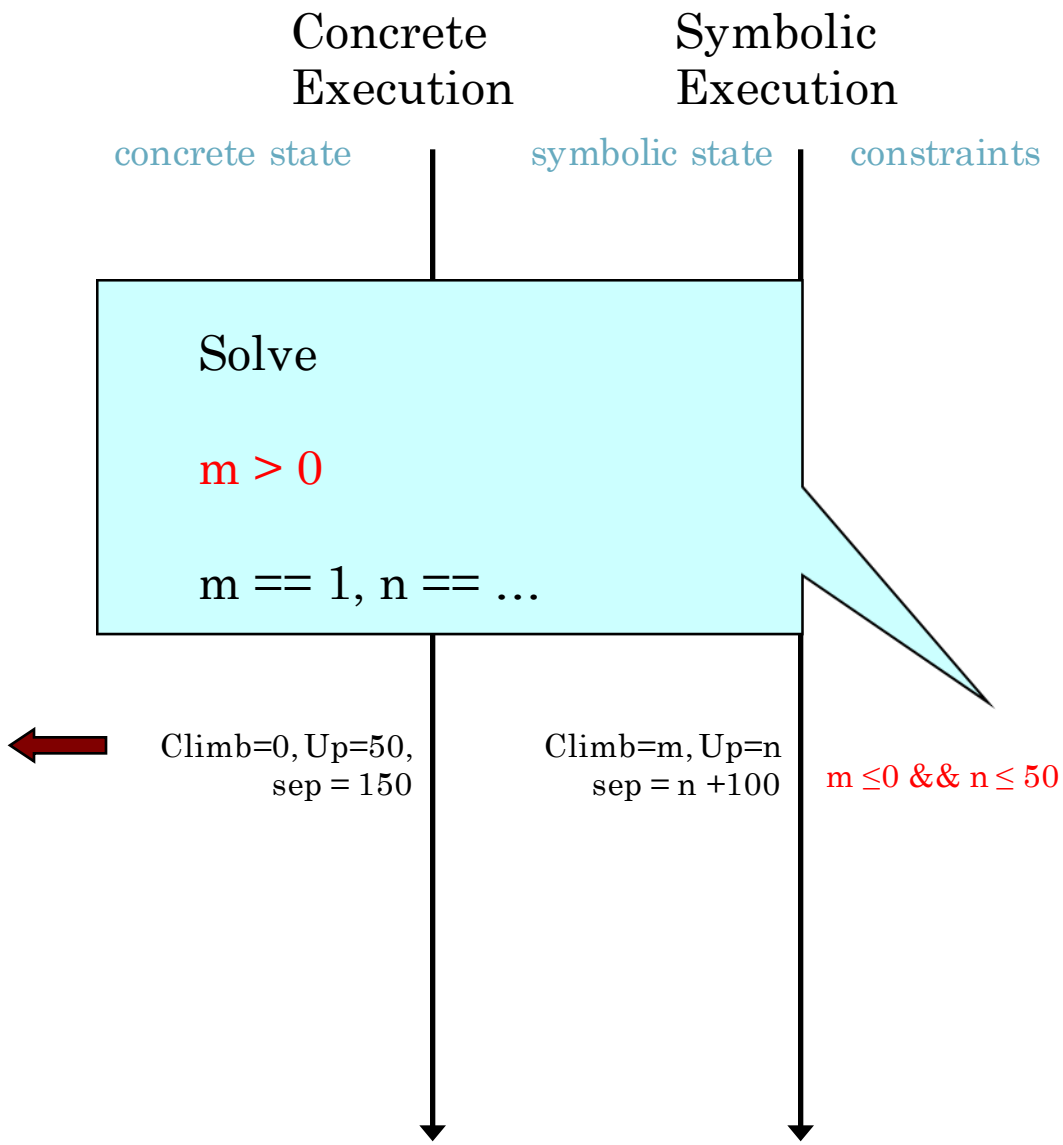
```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
     sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
      upward = 1;
    } else {upward = 0;}
    if (upward < 0){
          abort;
     } else return upward;
}
```



```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
     sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
      upward = 1;
    } else {upward = 0;}
    if (upward < 0){
          abort;
     } else return upward;
}
```

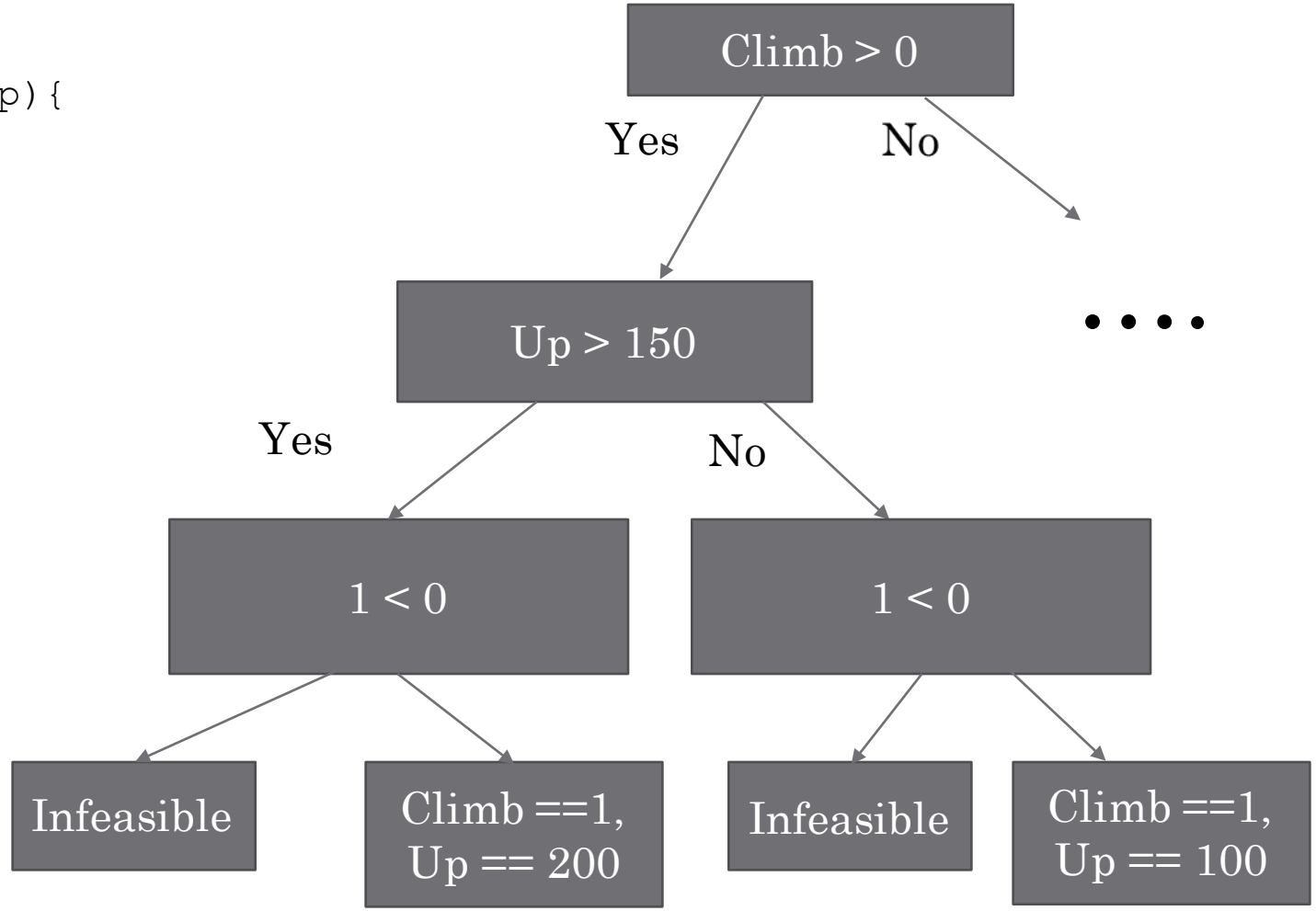


```
main(){
      int t1 = randomInt();
      int t2 = randomInt();
      test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
     sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
      upward = 1;
    } else {upward = 0;}
    if (upward < 0){
         abort;
     } else return upward;
}
```

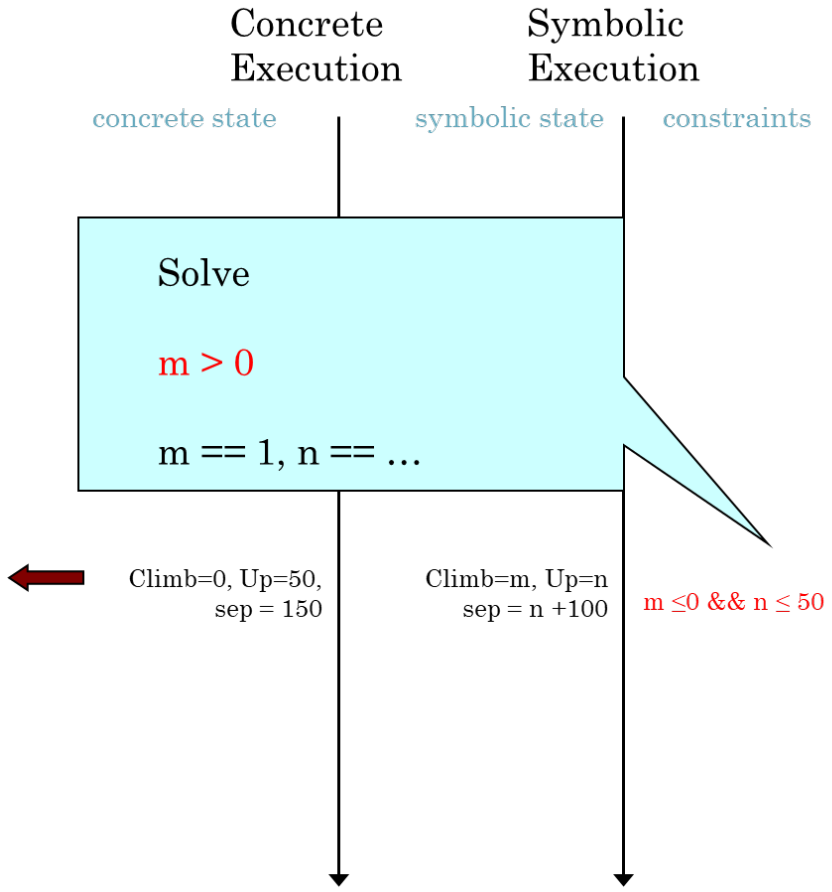


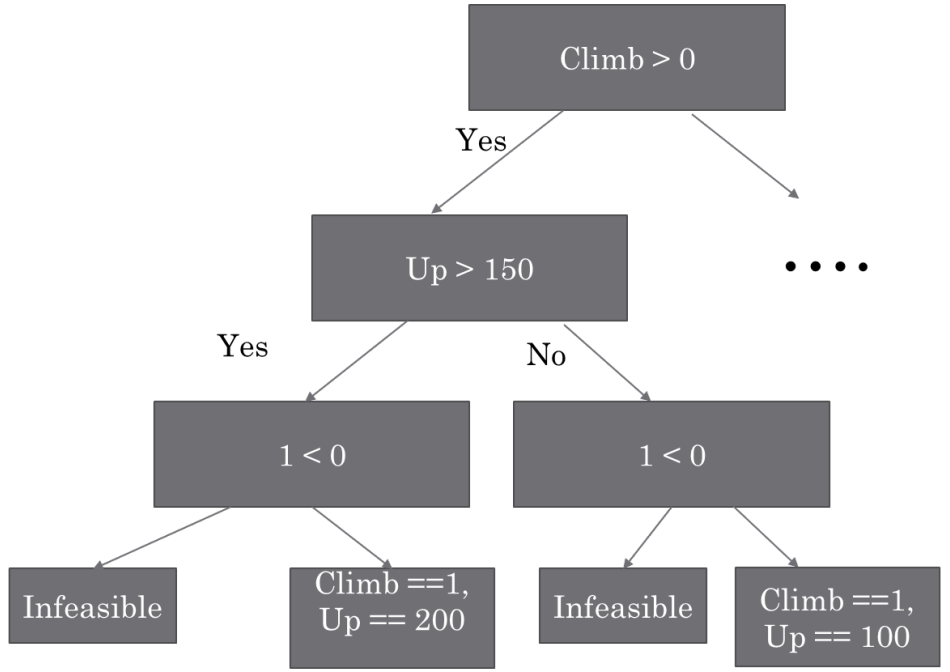
Symbolic Execution Tree

```
int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
      sep = Up;}
  else  {sep = add100(Up);}
  if (sep > 150){
       upward = 1;
    } else {upward = 0;}
  if (upward < 0){
            abort;
  } else return upward;
}
```



Concolic and Symbolic





One path at a time, simplify constraints!

Entire execution tree, Search Strategies!!

Symbolic and Concolic

- Symbolic
  - Execute IF(r)/then/else :fork [provided r is unresolved]
    - Then: PC := PC $\wedge$ r AND
    - Else: PC := PC $\wedge \neg$r

- Concolic:
  - Execute IF(r)
    - Resolved branch condition r using concrete values
    - Suppose true, PC := PC $\wedge$ r , OR
    - Suppose false, PC := PC $\wedge \neg$r

Concolic and Symbolic

```
1  foobar(int x, int y){
2    if (x*x*x > 0){
3      if (x>0 && y==10){
4        abort();
5      }
6    } else {
7      if (x>0 && y==20){
8        abort();
9      }
10   }
11 }
```

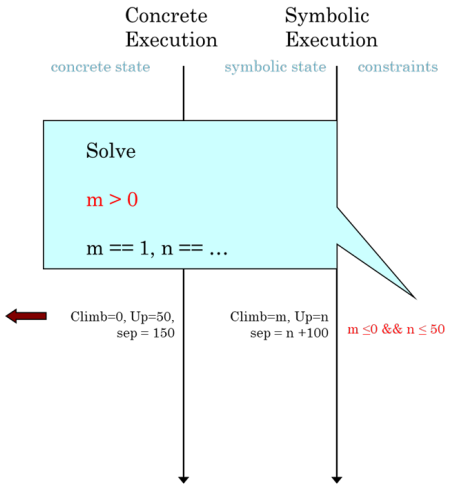
- static analysis based model-checkers would consider both branches
  - both abort() statements are reachable
  - false alarm
- Symbolic execution gets stuck at line number 2
- Concolic finds the error

x*x*x > 0 could be replaced by a library call and the discussion remains the same

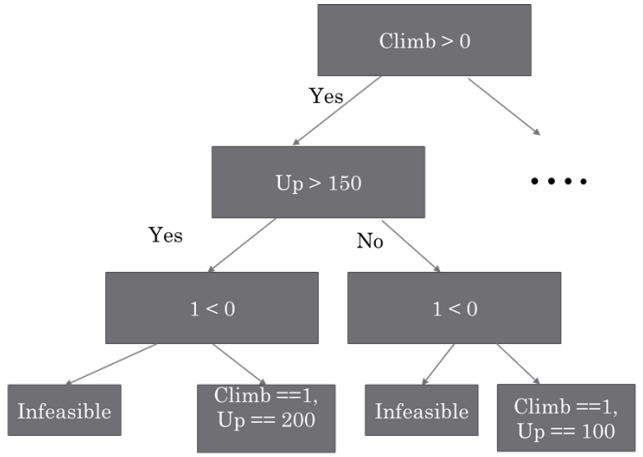
Bug Hunting vs. Reachability

```
…
while (input[ptr] != URI_DELIMITER){
      if (uri_len<80) …;
      uri_len++; ptr++;
}
while (input[ptr] != VERSION_DELIMITER){
      if (ver_len<80) …;
      ver_len++; ptr++;
}
if (ver_len<8|| version[5] != '1') …;
for(i=0,ptr=0; i< uri_len;i++, ptr++)
    msgbuf[ptr] = URI[i];
msgbuf[ptr++] = ',';
for (j=0ptr=0; j<ver_len; j++,ptr++)
    msgbuf[ptr] = version[j];
…
```

Webserver example with loops *(Ack: LESE paper by Saxena et al ISSTA 2008)*



Systematic Path exploration – bug hunting !



Adapted for reachability analysis of locations e.g. tools based on KLEE, **more to come in next hour.**

Just checking

- *.. Whether we are all awake (a bit late in the day !)*

- Consider two programs P1, P2 both of which take integer inputs x, y and produce integer output z.

- P1: if (x > y){ z = x + y; if (z > x){ z = z+1;}} else{z = x – y;}
- P2: if (x < y){z = x – y;} else{ z = x + y;}

- Construct a logical formula which captures all test inputs which generate different outputs in P1 and P2.

Answer:

The path summaries in P1 are

$x \leq y \Rightarrow z == x - y$
$x > y \wedge y > 0 \Rightarrow z == x + y + 1$
$x > y \wedge y \leq 0 \Rightarrow z == x + y$

The path summaries in P2 are
$x < y \Rightarrow z == x - y$
$x \geq y \Rightarrow z == x + y$

By comparing the two path summaries we see that the output expressions are different when x == y and when x > y > 0

Scenario 1: when x == y, P1 returns x – y and P2 returns x + y These two expressions are unequal when y != 0. So, this is captured by the constraint

$y \neq 0 \wedge x == y$

Scenario 2: when x > y > 0, P1 returns x + y + 1 and P2 returns x + y These two expressions are never equal. So, we get the constraint

$x > y > 0$

Fuzz Testing w, w/o SE

Abhik Roychoudhury

National University of Singapore

History of fuzzing

Term coined 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.*

Salient features of fuzzing

- Automated test generation
  - Favor slightly anomalous or malformed or illegal inputs
  - Apart from this issue, try to keep test generation random
- Automated test execution
  - Of course
- Automated and weak notion of test oracle
  - No notion of expected output to see if a test is passing
  - Simply see if the application is hanging.
- Detailed record-keeping
  - For crashing tests, one may find **lot of crashing tests** by fuzzing
- Independent of any programming language, OS etc.
  - No analysis, only execution!

Output of fuzzing

- Lot of crashing tests
  - Voluminous, not directly useful
  - Lot of crashing tests may be a manifestation of the same vulnerability.
  - *Need to cluster crashing tests based on why they crash!*

- *What do we do with output from fuzzing*
  - *Check whether attackers can exploit the vulnerability*
  - *Or, it may be easier to just fix the error rather than checking its exploitability.*

Fuzz Testing

Pioneered by Barton Miller at Unv. of Wisconsin in 1988

And now, in 2016 …

Springfield Project - Fuzzing as a service

OSS-Fuzz - Continuous fuzzing for open-source projects

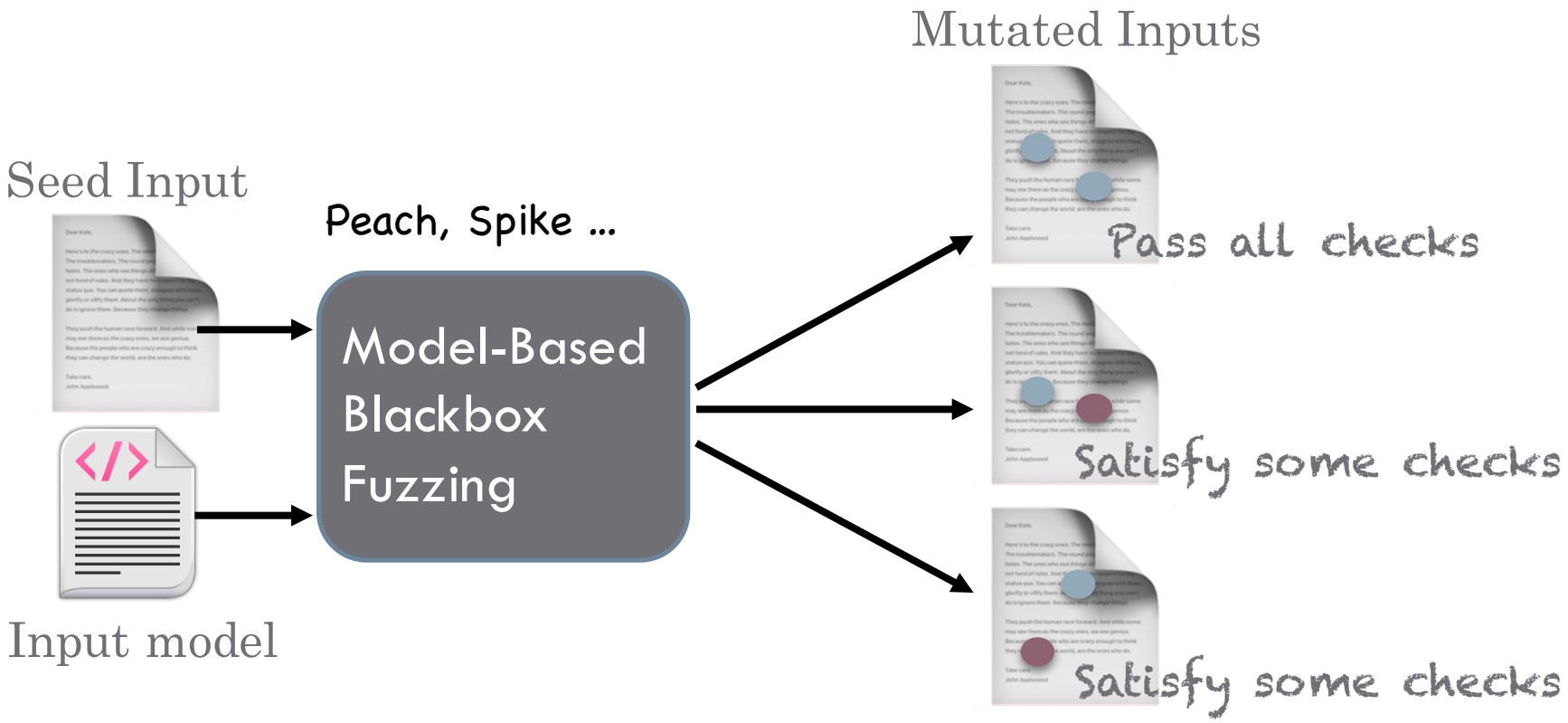
Who cares?

A team of hackers won $2 million by building a machine that could hack better than they could
Read more at *http://www.businessinsider.sg/forallsecure-mayhem-darpa-cyber-grand-challenge-2016-8/#ZuIF7Dmq3aaCAdaq.99*

DARPA Cyber Grand Challenge

Automation of Security [detecting and fixing vulnerabilities in binaries automatically]

(Model-Based) Black-box Fuzzing



Mutational fuzzing

- Inputs
  - Program P
  - Seed input x0
  - Mutation ratio $0 < m \leq 1$
- Next step
  - Obtain an input x1 by randomly flipping $m*|x0|$ bits
  - Run x1 and check if P crashes or terminates properly.
  - In either case document the outcome, and generate next input.
- End of fuzz campaign
  - When time bound is reached, or N inputs are explored for some N.
  - Always make sure that bit flipping does not run same input twice.

Why depend on mutations?

- Many programs take in structured inputs
  - PDF Reader, library for manipulating TIFF, PNG images
  - Compilers which take in programs as input
  - Web-browsers, ...

- Generating a completely random input will likely crash the application with little insight gained about the underlying vulnerability.

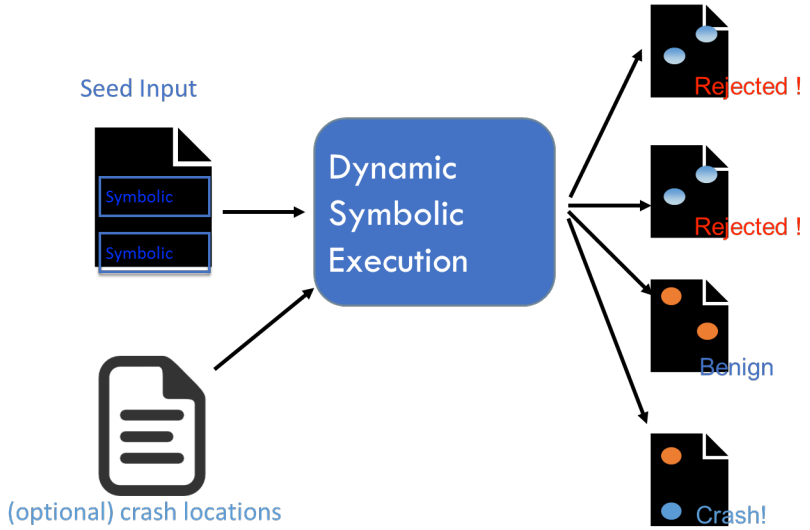
- Instead take a legal well-formed PDF file and mutate it!

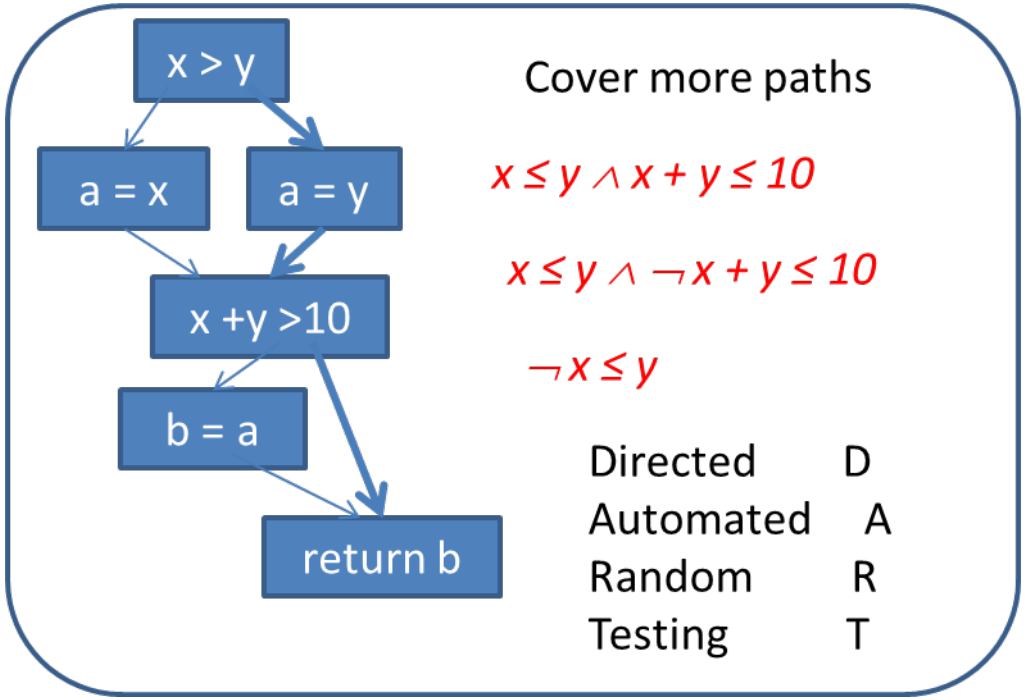
Why depend on mutations?

- Principle of mutation fuzzing
  - Take a well-formed input which does not crash.
  - Minimally modify or mutate it to generate a "slightly abnormal" input
  - See if the "slightly abnormal" input crashes.

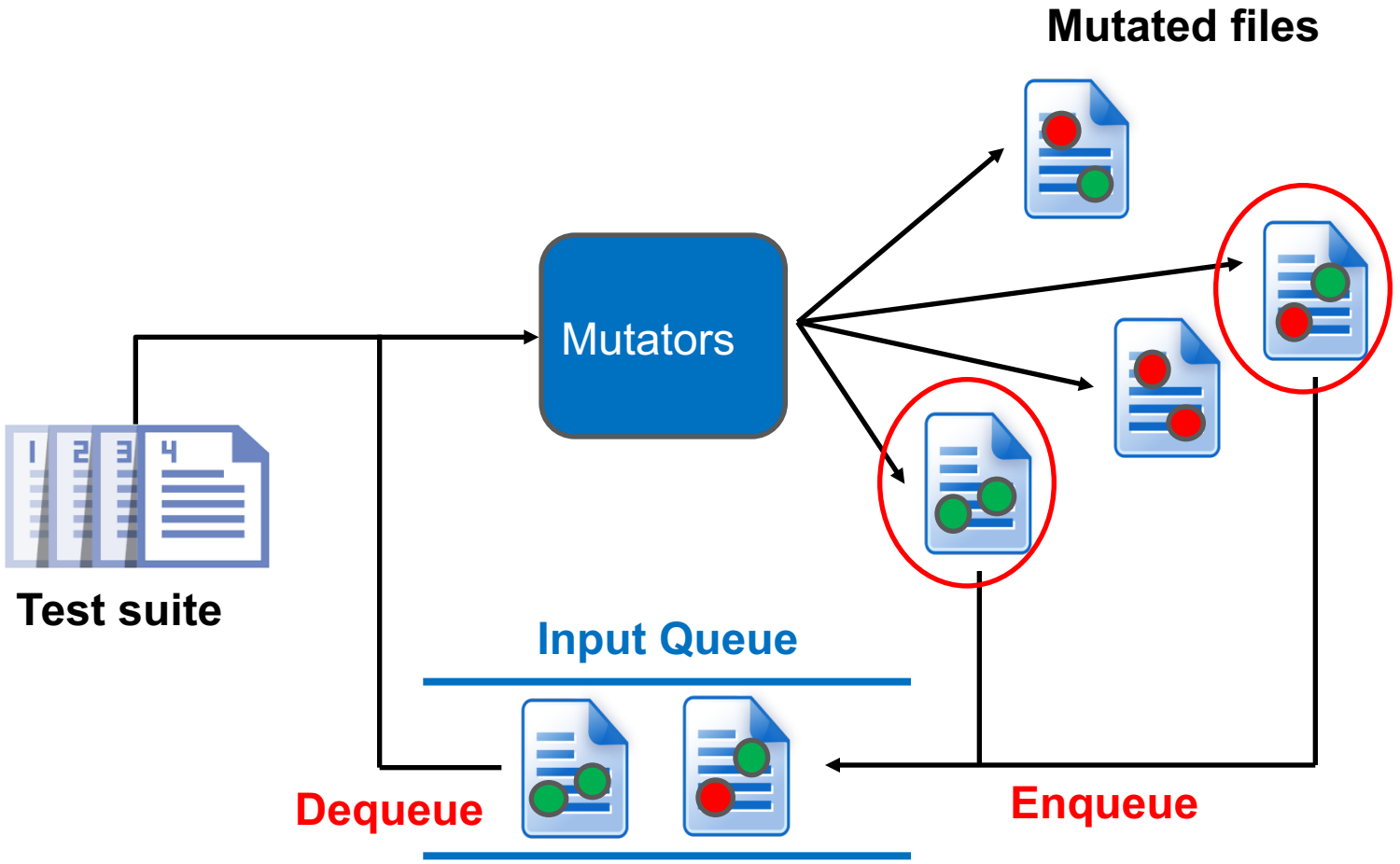
- Salient features
  - Does not depend on program at all [nature of BB fuzzing]
  - Does not even depend on input structure.
  - Yet can leverage complex input structure by starting with a well-formed seed and minimally modifying it.

White-box Fuzzing





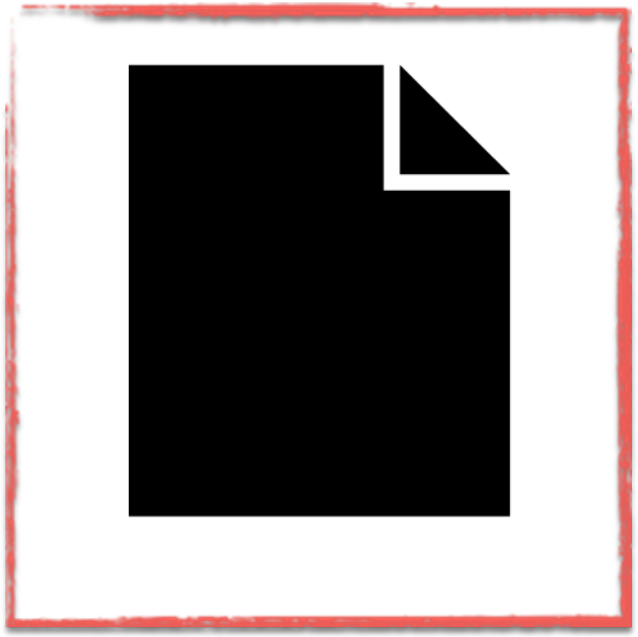
Grey-box Fuzzing, as in AFL



Mutations

Mutation Operators:

- Bitflips
- Boundary Values
  `(0,1,-1,INT_MAX,INT_MIN)`
- Simple arithmetic
  `(add/subtract 1)`
- Block deletion
- Block insertion



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

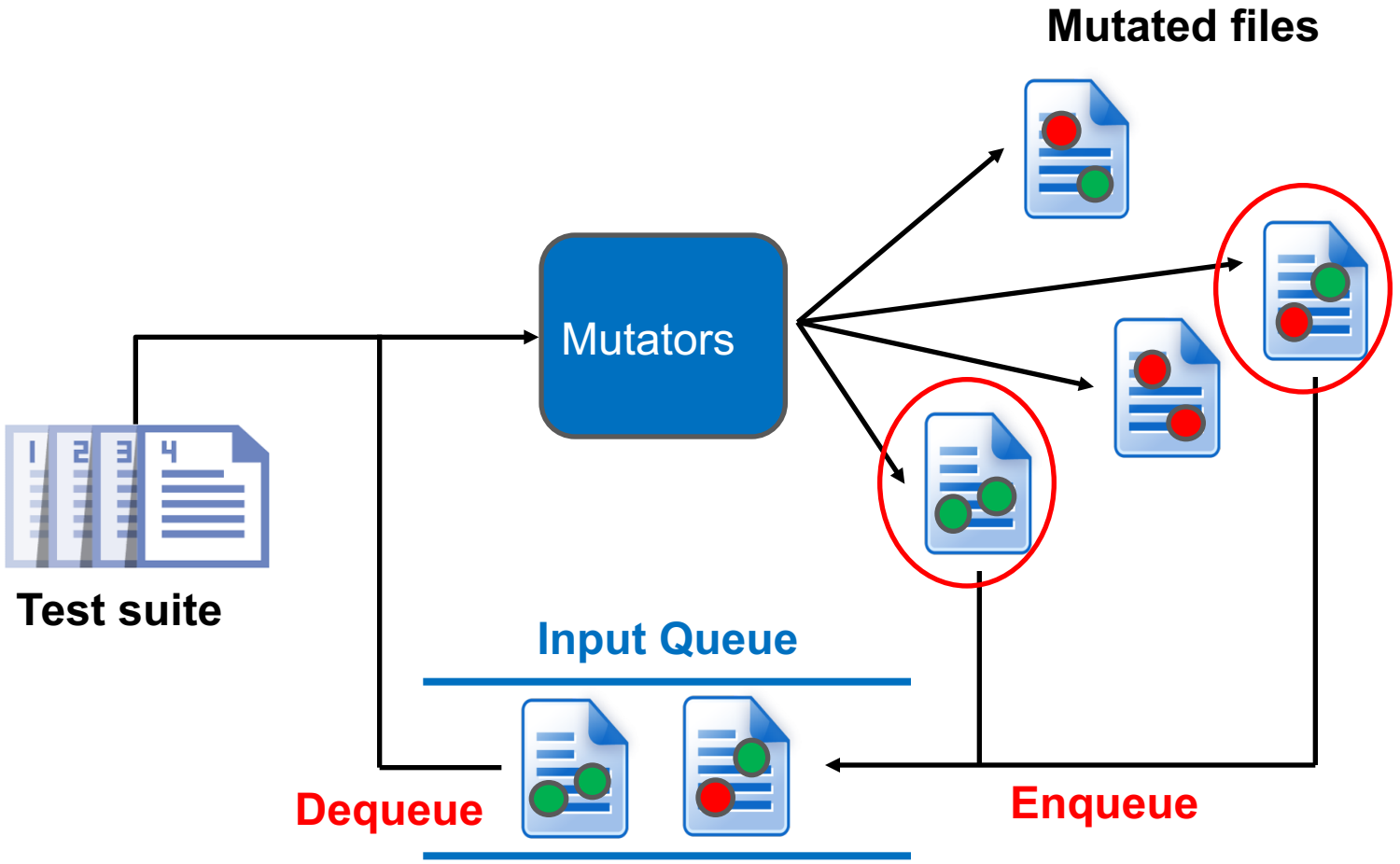
- Enhance the effectiveness of search techniques, with symbolic execution as inspiration

Symbolic Execution

- Explore capabilities of symbolic execution beyond search

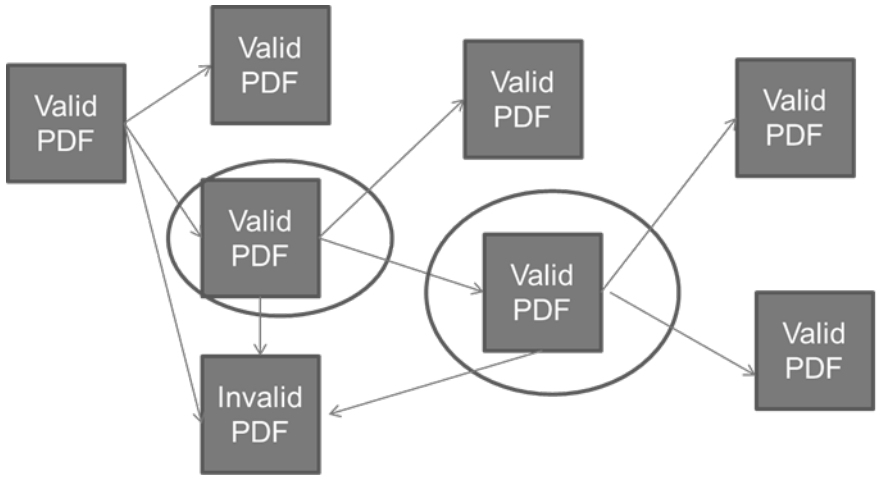
- **Systematic Fuzz Testing**

Grey-box Fuzzing, as in AFL



Grey-box Fuzzing Algorithm

- Input: Seed Inputs S
- 1: $T_{\times} = \varnothing$
- 2: T = S
- 3: if T = ∅ 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 $T_{\times}$
- 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 $T_{\times}$



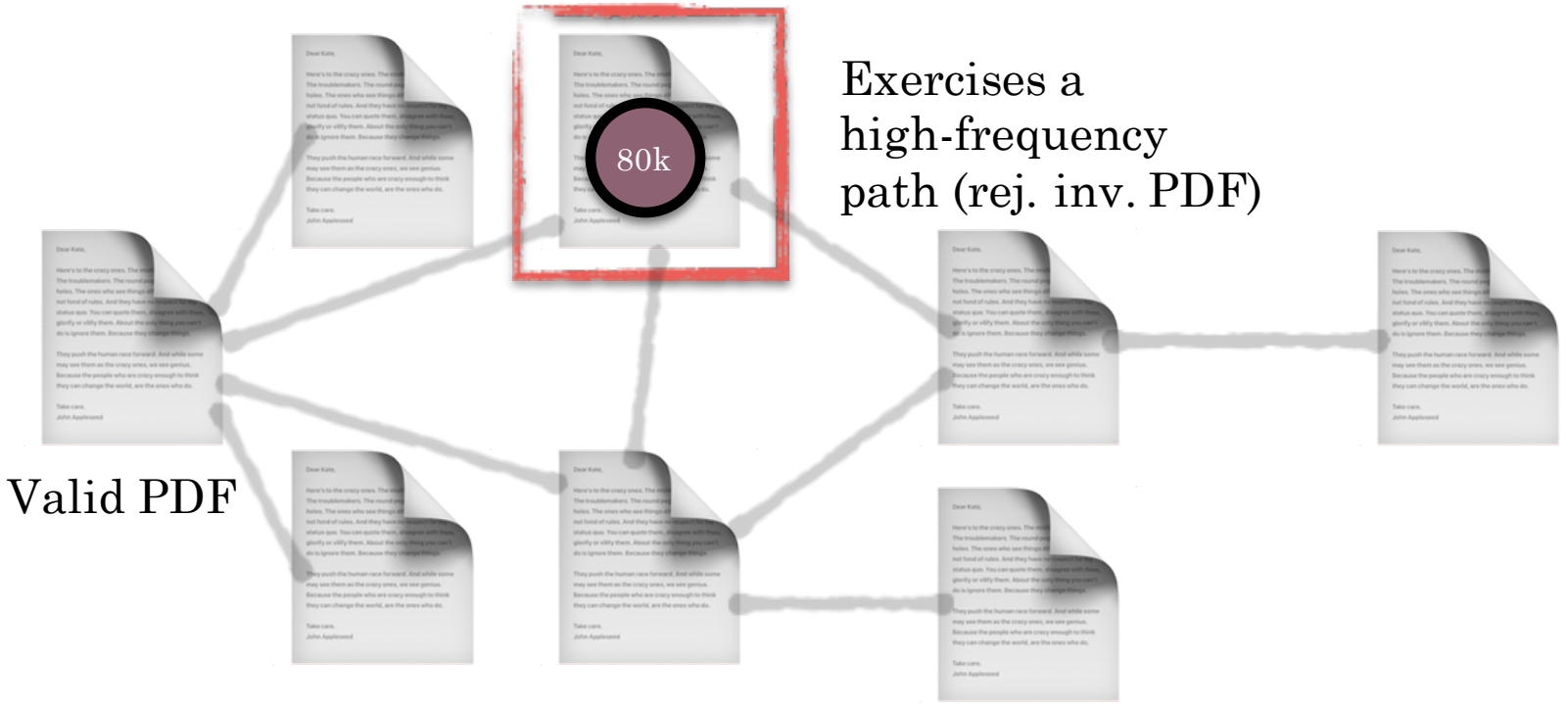
Programming by experienced people

Schematic

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

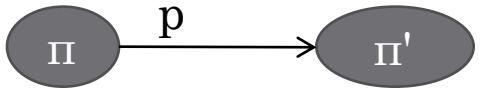
Core intuition

- AFL's power schedule is *constant* in the number of times s(i) the seed has been chosen for fuzzing.
- AFL's power schedule always assigns *high* energy



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'



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

```
1    void crashme (char* s) {
2          if (s[0] == 'b')
3                if (s[1] == 'a')
4                      if (s[2] == 'd')
5                            if (s[3] == '!')
6                                  abort ();
7    }
```

Power-Schedules

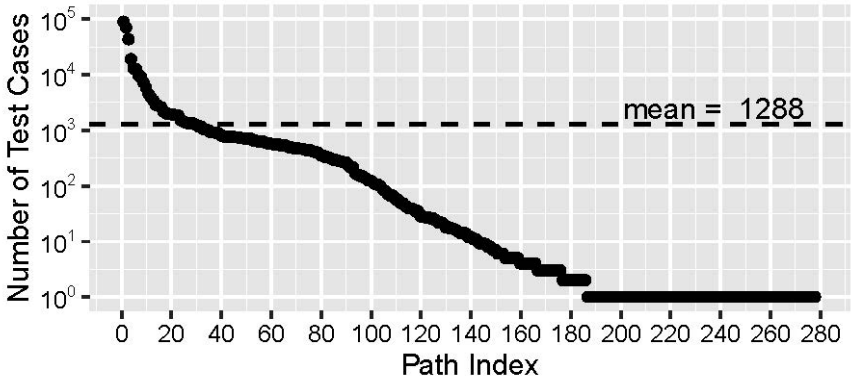
- Constant: $p(i) = \alpha(i)$
  - AFL uses this schedule (fuzzing ~1 minute)
  - $\alpha(i)$ .. how AFL judges fuzzing time for the test exercising path i

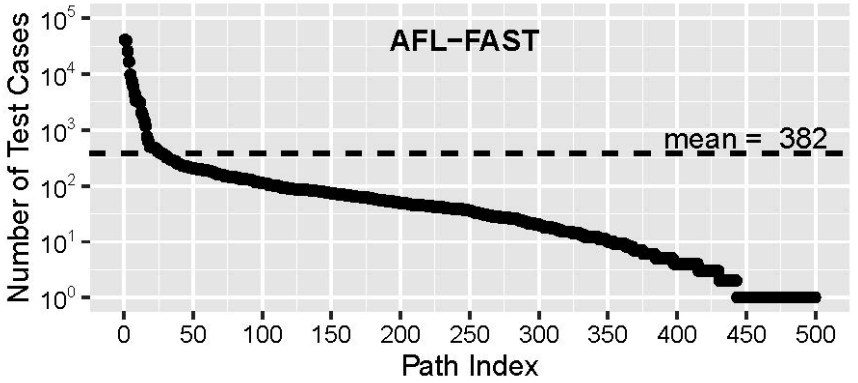
- Cut-off Exponential:

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

ß 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

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 [-]

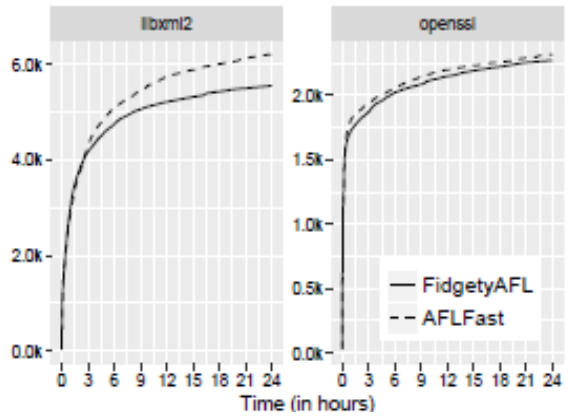
The question of whether randomized testing or program analysis gives you more coverage of a program is a really interesting one. Böhme actually has an earlier paper that addresses this question: https://www.comp.nus.edu.sg/~mboehme/paper/FSE14.pdf

Independent Evaluation

- An independent evaluation by team Codejitsu from Berkeley found that AFLFast exposes errors in the benchmark binaries of the DARPA Cyber Grand Challenge 19x faster than AFL.

Independent Evaluation and Deployment

- Picked up by Zalewski@AFL, with following observations, paraphrased
  - *AFLFAST assigns substantially less energy in the beginning of the fuzzing campaign.*
  - *Most of the cycles that AFLFAST carries out, are in fact very short. This causes the queue to be cycled very rapidly, which in turn causes new retained inputs to be fuzzed almost immediately. In other words, because AFLFAST assigns less energy, it can process the complete queue substantially faster. We say it starts by exploration rather than by exploitation*
- Implemented inside AFL (version 2.33b, FidgetyAFL), and distributed approximately within one year of publication



There remain differences between the two in terms of path discovered. More experiments may be needed.

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 …

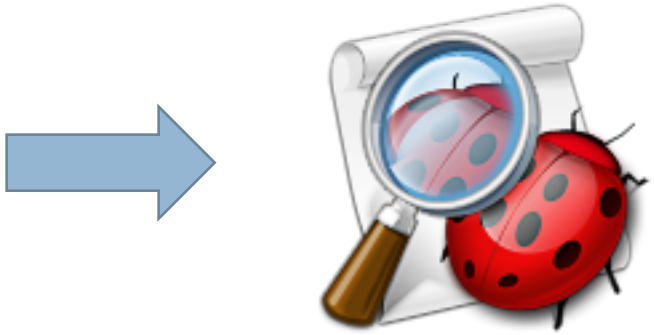
Search

- 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

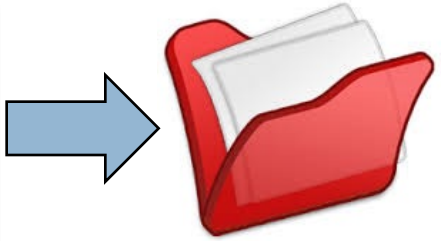
Using symbolic execution

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

Program binary

(Crash instruction, loaded modules, call stack, register values)

Hercules Toolset

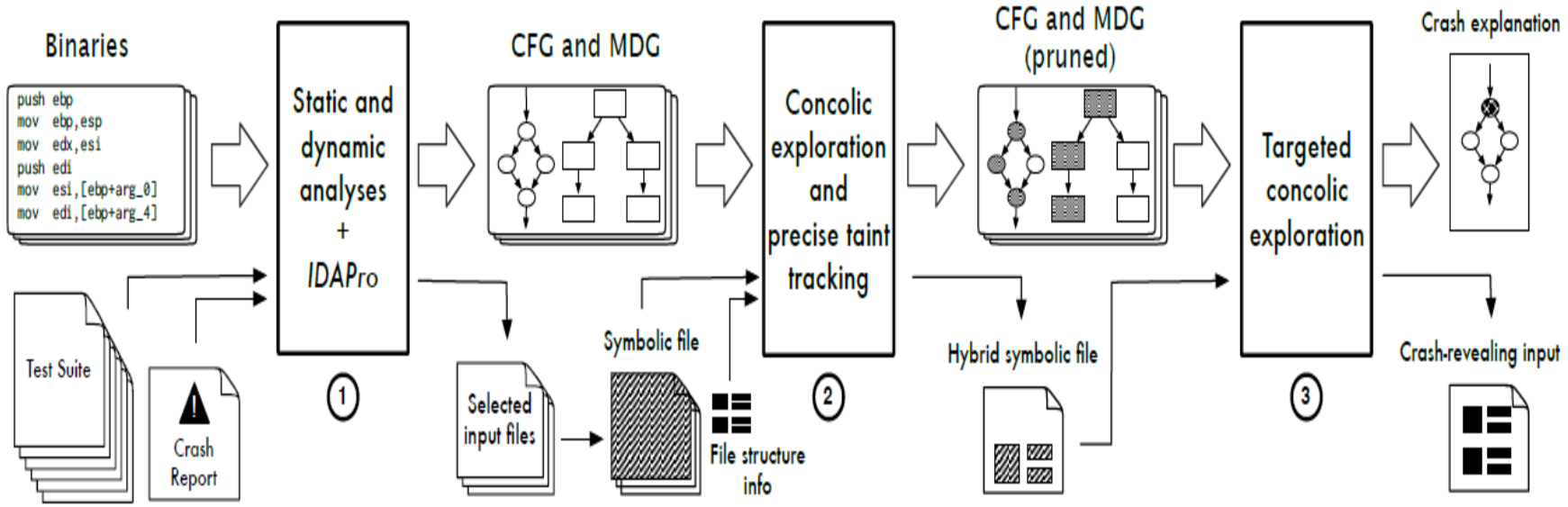


Crash input files

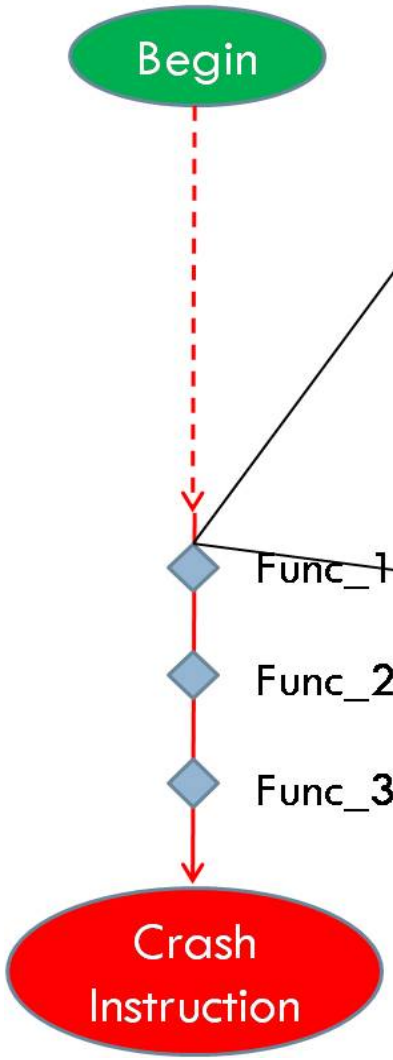
Benign input files

1. Directed Search Algorithm
2. Guided Selective Symbolic Execution

Symbolic Analyzer



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



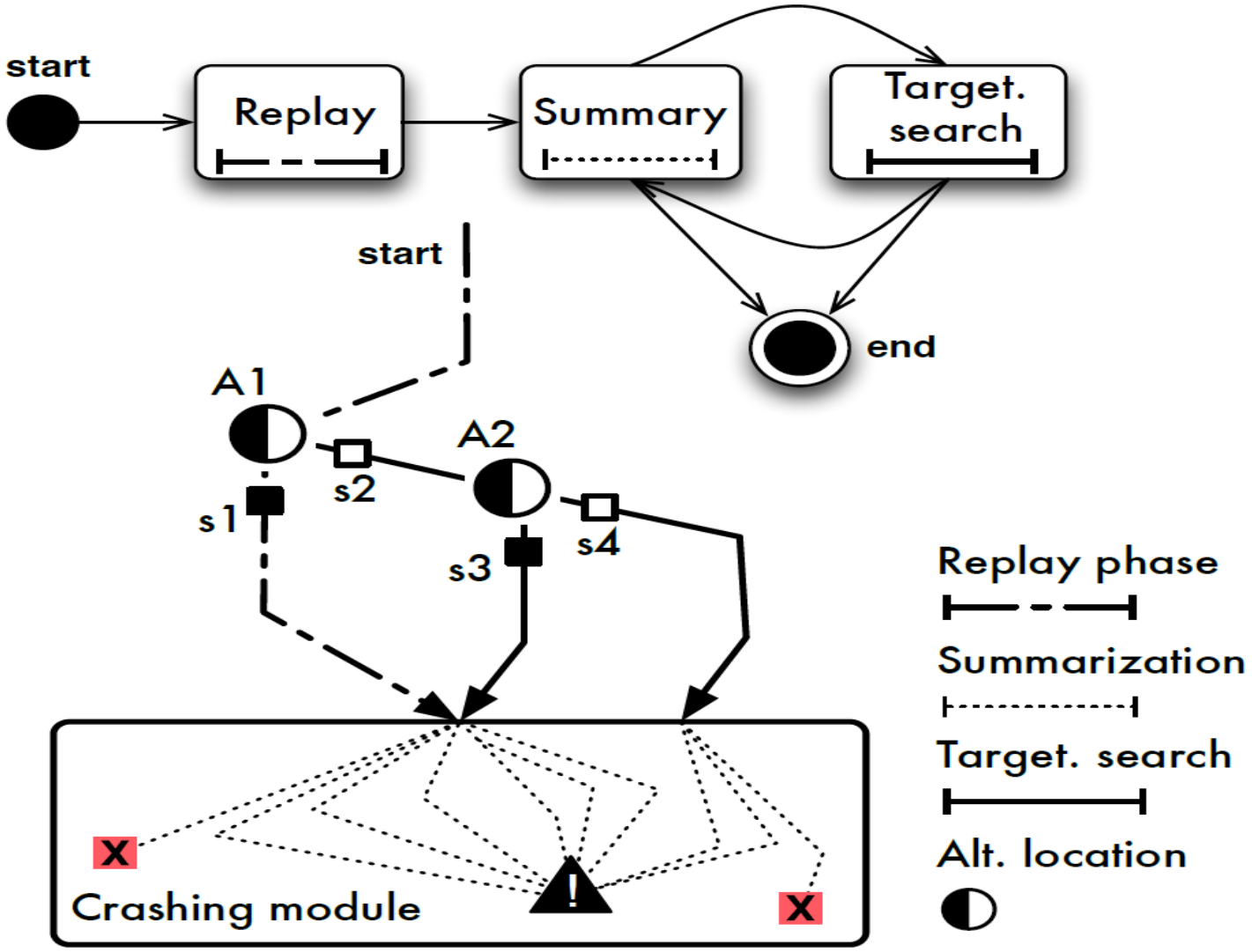
Path constraints at Func_1

```
...
(Eq (w32 0x46464952)
  (Concat w32 (Read w8 0x0 v3_sym_byte_3)
    (Concat w24 (Read w8 0x0 v2_sym_byte_2)
      (Concat w16 (Read w8 0x0 v1_sym_byte_1)
(Read w8 0x0 v0_sym_byte_0)))))
Constraint (Eq false
  (Eq (w32 0x44494d52)
    (Concat w32 (Read w8 0x0 v11_sym_byte_11)
       (Concat w24 (Read w8 0x0 v10_sym_byte_10)
         (Concat w16 (Read w8 0x0 v9_sym_byte_9)
(Read w8 0x0 v8_sym_byte_8))))))
...
```

Hybrid Symbolic File

```
...
4D 54 68 64 00 00 00 06 00 01 00 0B 00 F0 4D 54   MThd..........ðMT
72 6B 00 00 00 13 00 FF 58 04 04 02 18 08 00 FF   rk.....ÿX......ÿ
...
```

Hercules Targeted Search



Reaching a location

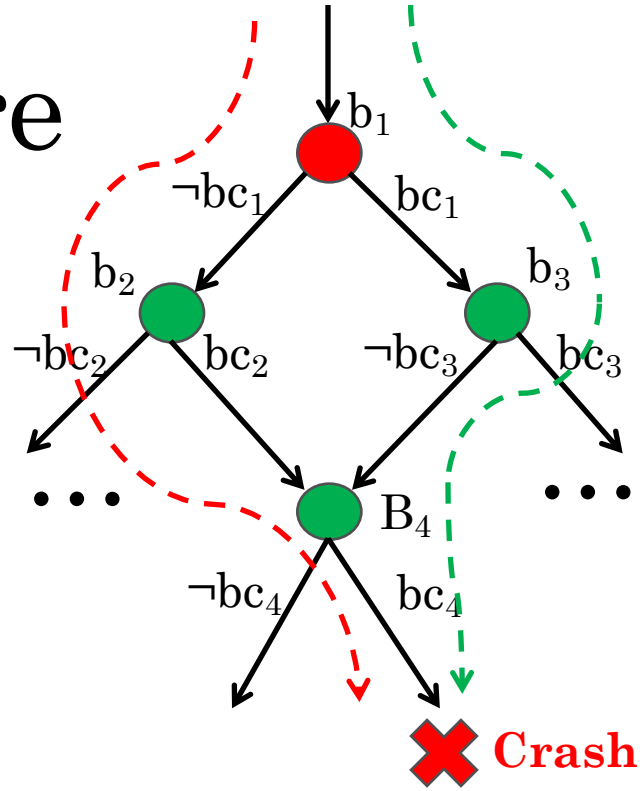
PC ^ CC

Challenges:

- Incomplete program structures
- Multi-module program
- The input file formats are complex

- Operands of the Crash instruction is *"not tainted"*
- *Example:* div ecx

UNSAT-core



Notations:
b_x: branch instruction
bc_x: branch condition at b_x
PC: path condition
CC: crash condition

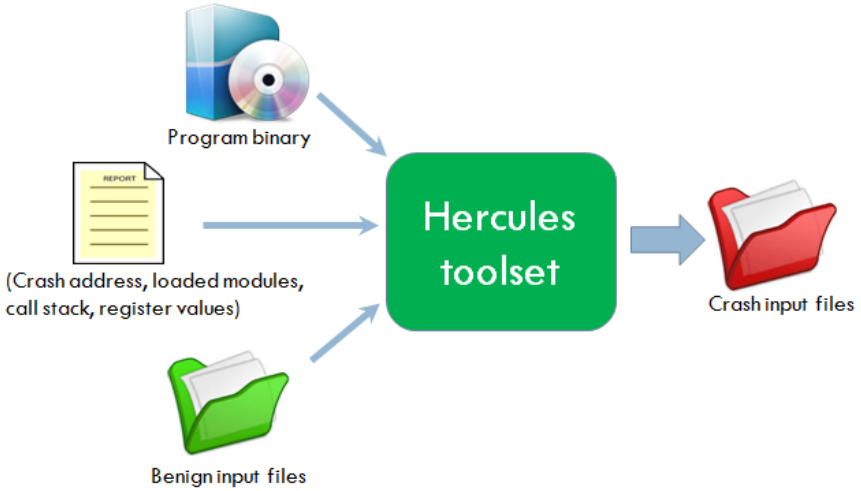
Crash instruction

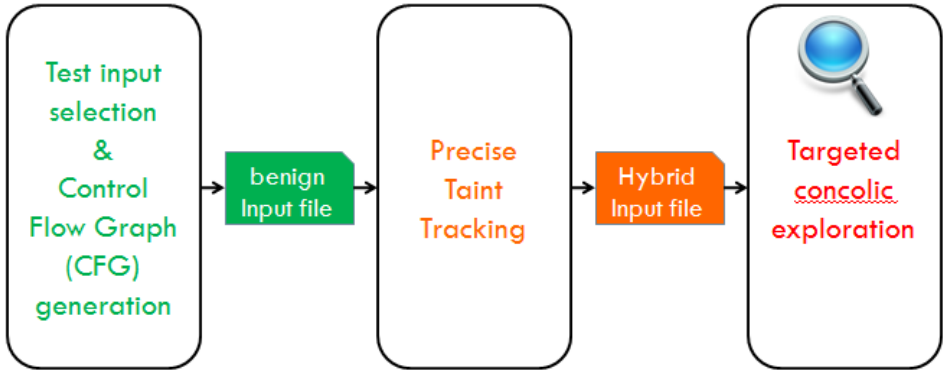
First attempt:
PC = bc_1 ^ ¬bc_3 ^ bc_4
PC ^ CC == UNSAT
bc_1 contradicts CC

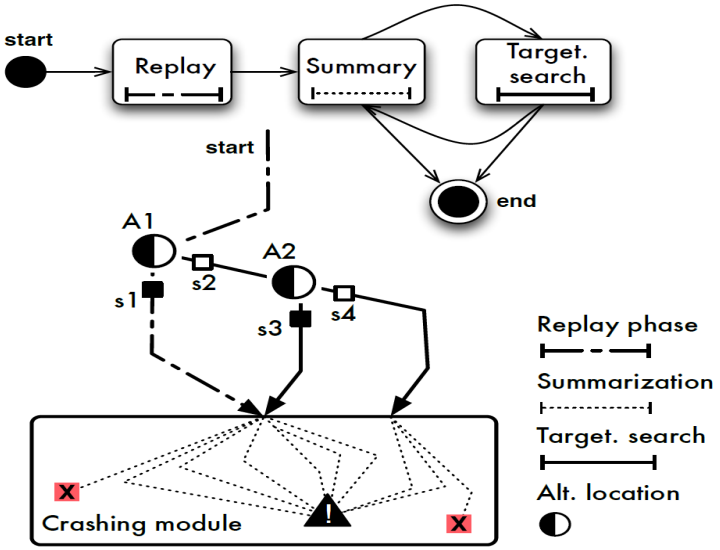
1) Backtrack to b_1
2) Take another branch

Second attempt:
PC' = ¬bc_1 ^ bc_2 ^ bc_4
PC' ^ CC == SAT

Hercules!



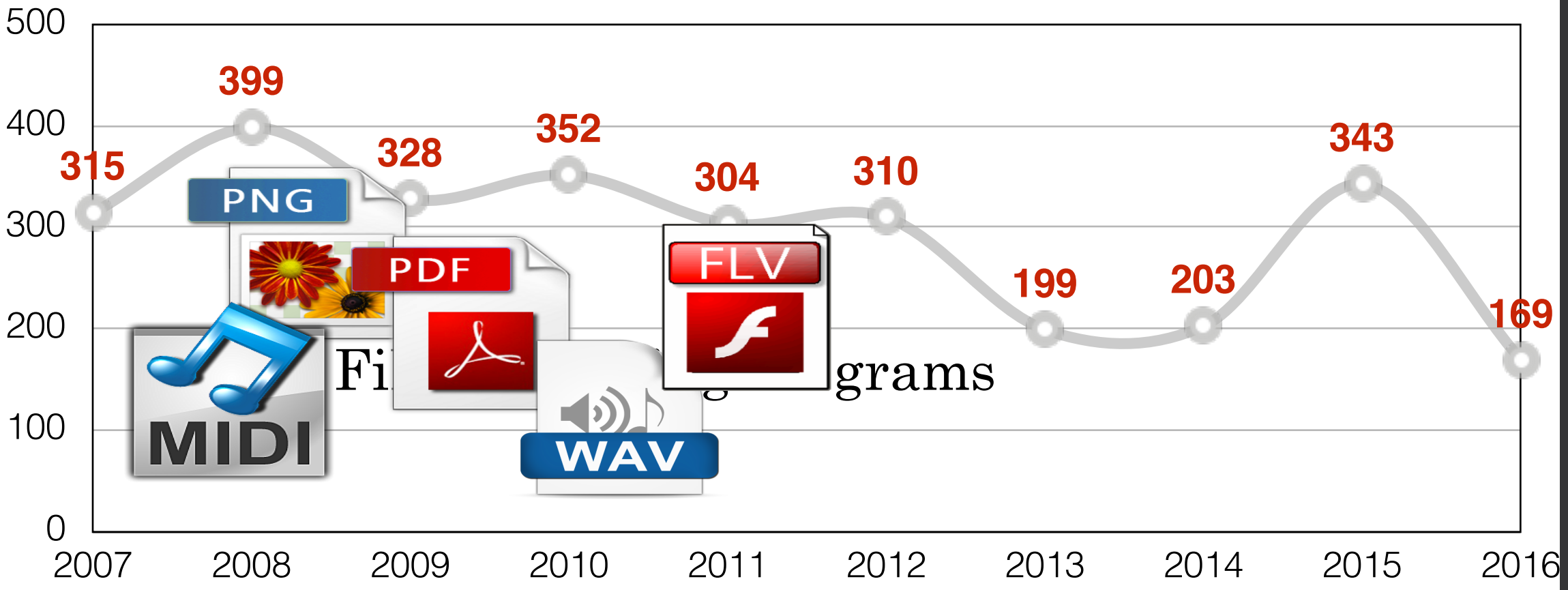




Case studies	Selected modules	S2E (DFS and Random)	Peach Fuzzer - mutation	Hercules
CVE-2010-2204 (Adobe Reader) Memory Access Violation	2 / 78	✗	✗	(*) ✓
CVE-2010-3000 (Real Player) Integer Overflow	2 / 129	✗	✗	✓
CVE-2014-2671 (Windows Media Player) Division by Zero	4 / 84	✗	✗	✓
CVE-2010-0718 (Windows Media Player) Buffer overflow	3 / 86	✗	✗	✓
CVE-2010-0688 (Orbital Viewer) Buffer overflow	2 / 49	✗	✓	✓
CVE-2011-0502 (MAM player) Null pointer reference	1 / 51	✓	✓	(*) ✓

Vulnerabilities in file-processing programs

#CVE-assigned vulnerabilities by year



(US National Vulnerability Database)

(By 30/8)

Motivating Example

A PNG file triggers a crash in VLC media player

```
"\x89\x50\x4E\x47\x0D\x0A\x1A\x0A" # PNG signature
"\x00\x00\x00\x0D" # IHDR size
"\x49\x48\x44\x52" # IHDR chunk
"\x7F\xFF\xFF\xFF" # width
"\x00\x00\x01\x02" # height
"\x01" # bit depth
"\x03" # color type
"\x00" # compression method
"\x00" # filter method
"\x00" # interlace method
"\xBA\x1B\xD8\x84" # IHDR chunk CRC
"\x00\x00\x00\x03" # PLTE size
"\x50\x4C\x54\x45" # PLTE chunk
"\xFF" # red
"\xFF" # green
"\xFF" # blue
"\xA7\xC4\x1B\xC8" # PLTE chunk CRC
[redacted]
"\x00\x00\x00\x01" # IDAT size
"\x49\x44\x41\x54" # IDAT chunk
"\xFF" # image data
"\x05\x3A\x92\x65" # IDAT chunk CRC
"\x00\x00\x00\x00" # IEND size
"\x49\x45\x4E\x44" # IEND chunk
"\xAE\x42\x60\x82" # IEND chunk CRC
```

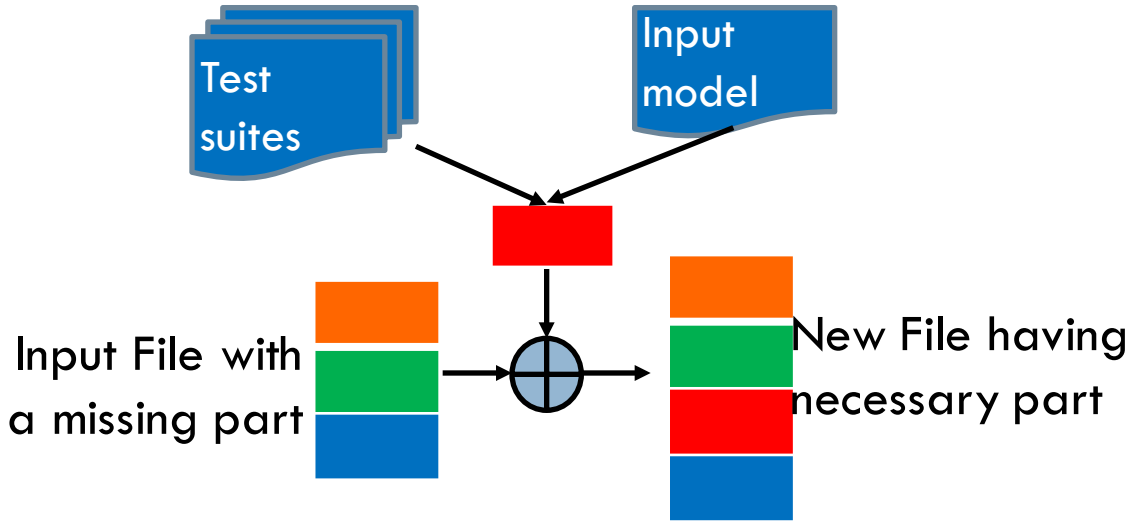
Requires specific values for some data fields

MOBF & WF are very unlikely to generate the crashing input IF the selected seed file does not have optional tRNS data chunk

Requires an optional data chunk

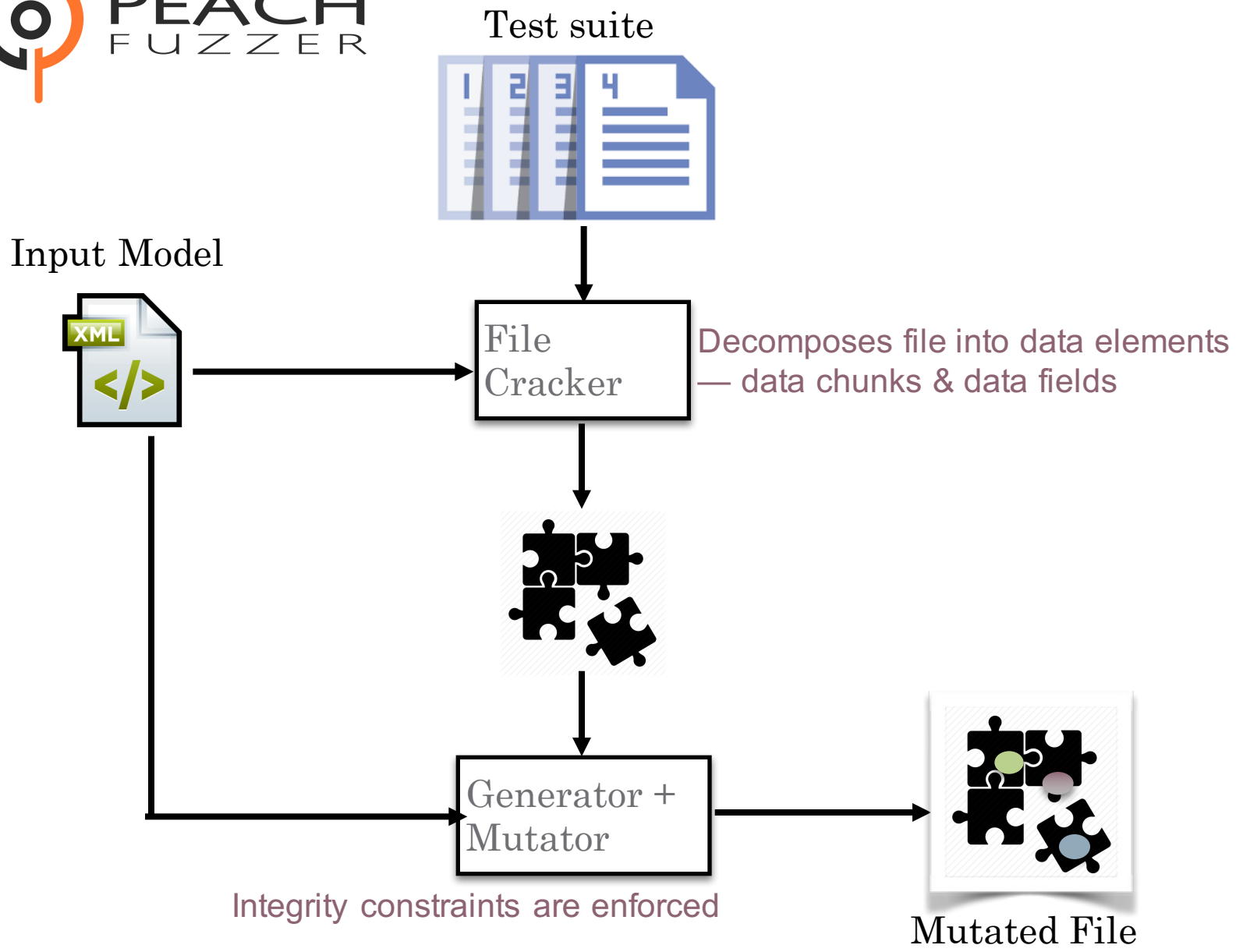
Observation & Solution

- A missing data chunk can be obtained from other seed inputs in the test suite
- OR it can be directly instantiated from the ***input model***

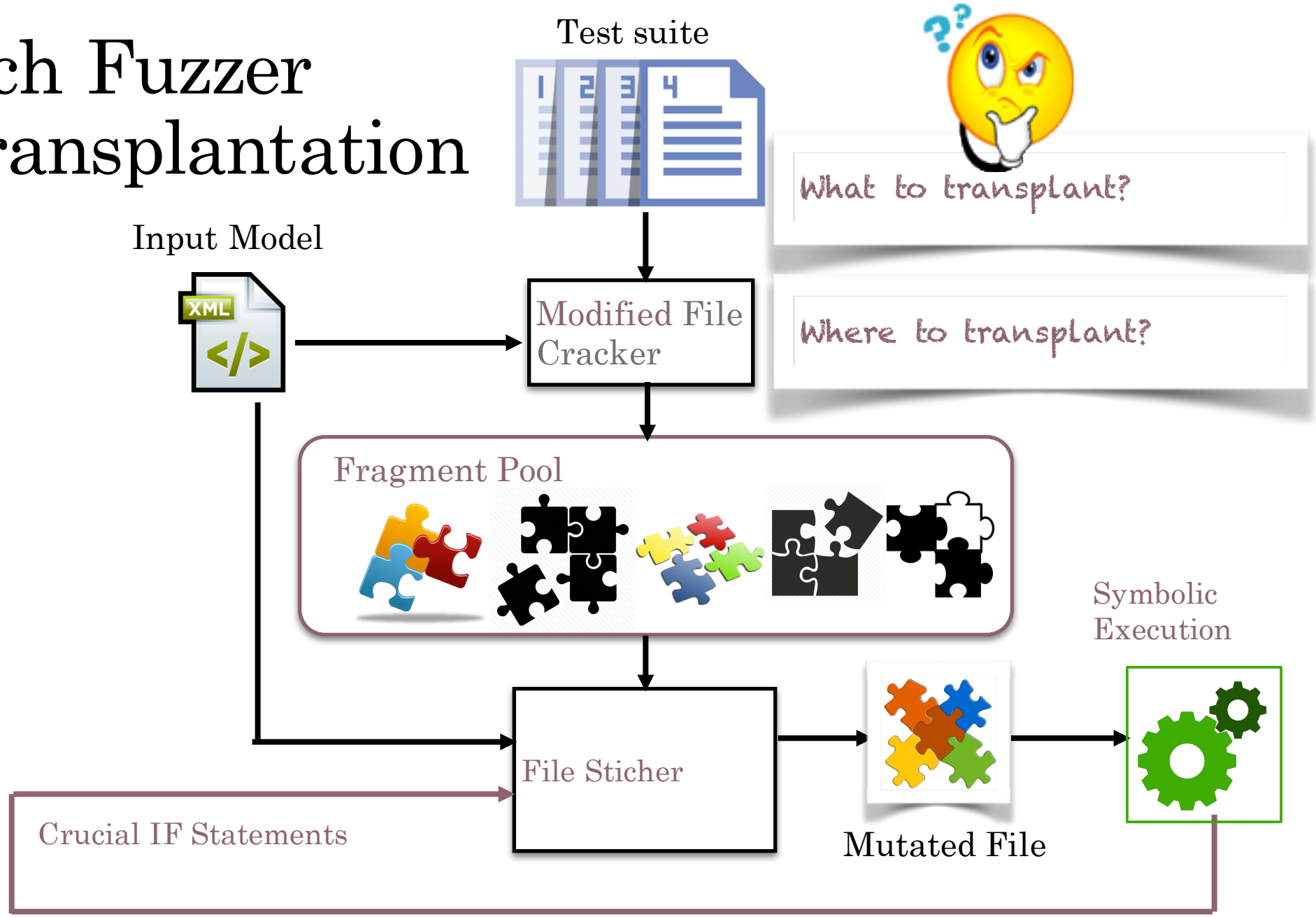


Data chunk Transplantation

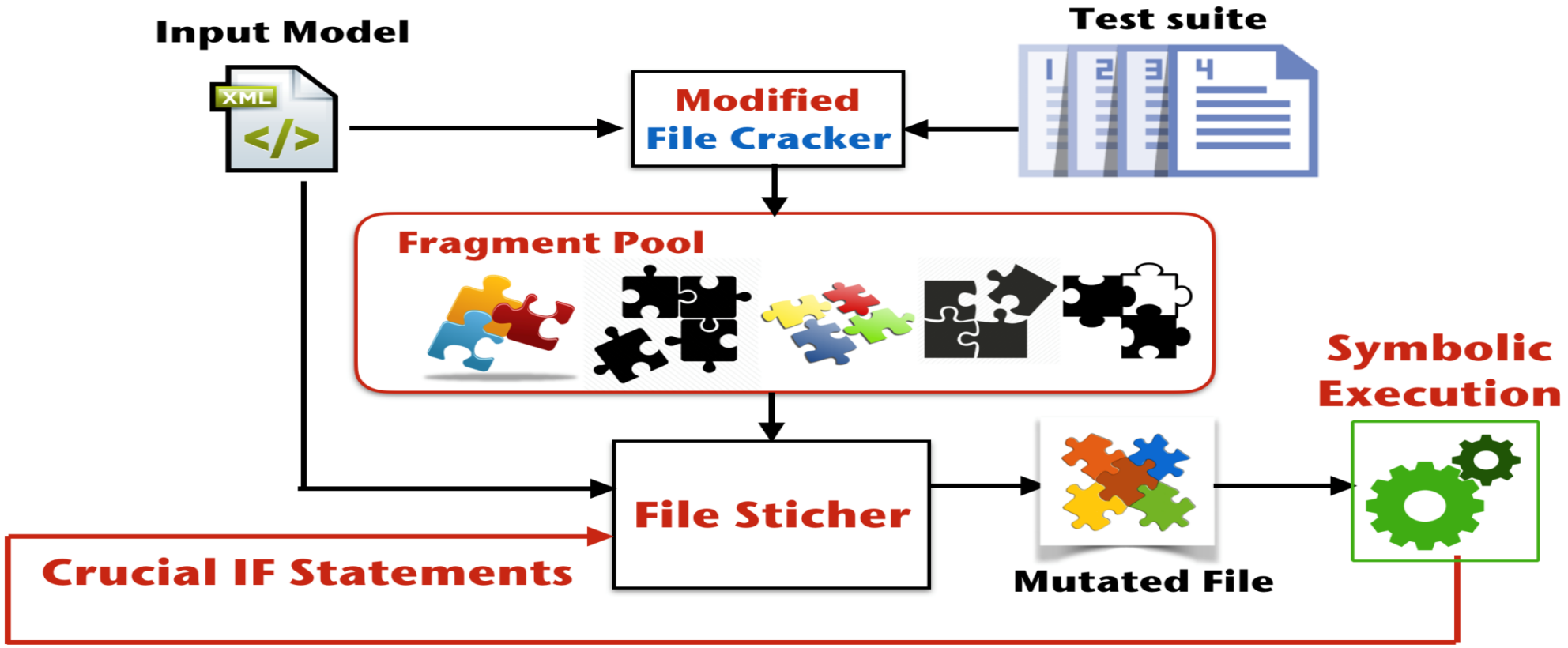
PEACH FUZZER



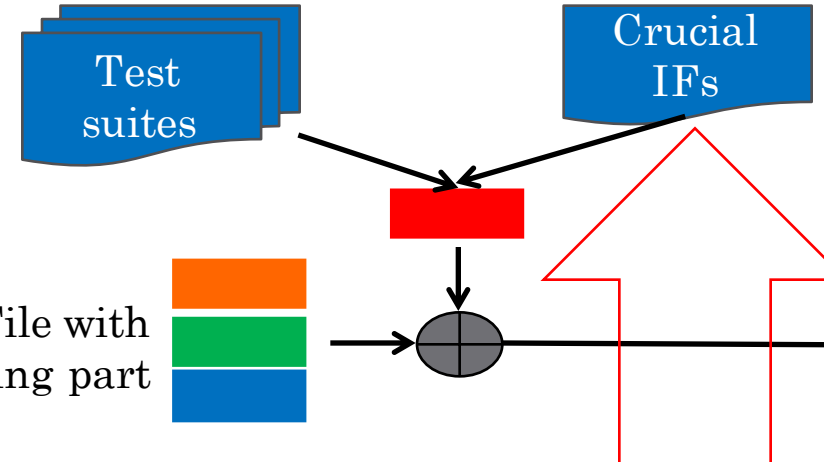
Peach Fuzzer + Transplantation



Combination

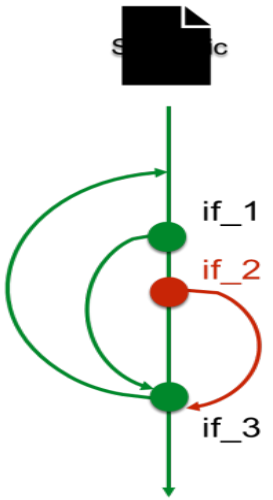


Crucial IF




```
1  // read chunks' info before first IDAT chunk
2  void png_read_info(png_structp ptr)
3  {
4  // read and check the PNG file signature
5  read_sig(f);
6  for (;;)
7  {
8     // get current chunk's information
9     uint_32 length = read_chunk_header(ptr);
10    uint_32 chunk_name = ptr->chunk_name;
11    // mandatory chunks
12    if (chunk_name == png_IHDR)
13       handle_IHDR(ptr, length);
14    else if (chunk_name == png_IEND)
15       handle_IEND(ptr, length);
16    else if (chunk_name == png_PLTE)
17       handle_PLTE(ptr, length);
18    else if (chunk_name == png_IDAT)
19    {
20       ptr->idat_size = length;
21       break;
22    }
23    // optional chunks
24    else if ...
25    else if (chunk_name == png_tRNS)
26       handle_tRNS(ptr, length);
27    else if ...
28 }
29 }
30 // initialize row buffer for reading data from file
31 void png_read_start_row(png_structp ptr)
32 {
33    size_t buf_size;
34    ...
35    buf_size = calculateBufSize(ptr);
36    ptr->row_buf = png_malloc(ptr, buf_size);
37    png_memset(ptr->row_buf, 0, ptr->rowbytes);
38 }
```

- **Step 1.** Mark input file (partially) symbolic
- **Step 2.** Concolically execute program in one path - *same path as concrete input*
- **Step 3.** Collect branch conditions of IF statements at which *only one branch has been taken (e.g., if_2)*
- **Step 4.** Use symbolic-execution-based *taint analysis & input model* to analyse branch conditions (at *if_2*) to validate crucial IFs statements



Experimental Results

Program	Advisory ID	Input Model	#Seed files	Hercules++	Peach	Hercules
VLC 2.0.7	OSVDB-95632	PNG	0 – 10	✓	✗	✗
VLC 2.0.3	CVE-2012-5470	PNG	0 – 10	✓	✗	✗
LTP 1.5.4	CVE-2011-3328	PNG	0 – 10	✓	✗	✗
XNV1.98	Unknown-1	PNG	0 – 10	✓	✓	✗
XNV1.98	Unknown-2	PNG	0 – 10	✓	✓	✗
XNV1.98	Unknown-3	PNG	0 – 10	✓	✓	✗
WMP 9.0	Unknown-4	WAV	10	✓	✓	✗
WMP 9.0	CVE-2014-2671	WAV	10	✓	✗	✓
WMP 9.0	CVE-2010-0718	MIDI	0 – 10	✓	✗	✓
AR 9.2	CVE-2010-2204	PDF	10	✓	✗	✓
RP 1.0	CVE-2010-3000	FLV	10	✓	✗	✓
MP 0.35	CVE-2011-0502	MIDI	0 – 10	✓	✓	✓
OV 1.04	CVE-2010-0688	ORB	0 – 10	✓	✓	✓

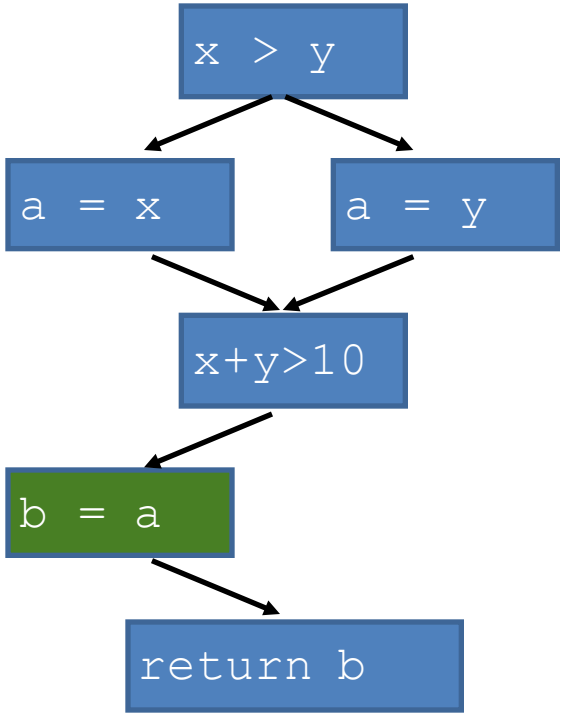
Evaluation - Seed Input Dependence

Program	Advisory ID	Input Model	#Seed files	Hercules++
VLC 2.0.7	OSVDB-95632	PNG	0	✓
VLC 2.0.3	CVE-2012-5470	PNG	0	✓
LTP 1.5.4	CVE-2011-3328	PNG	0	✓
XNV1.98	Unknown-1	PNG	0	✓
XNV1.98	Unknown-2	PNG	0	✓
XNV1.98	Unknown-3	PNG	0	✓
WMP 9.0	Unknown-4	WAV	0	✗
WMP 9.0	CVE-2014-2671	WAV	0	✗
WMP 9.0	CVE-2010-0718	MIDI	0	✓
AR 9.2	CVE-2010-2204	PDF	0	✗
RP 1.0	CVE-2010-3000	FLV	0	✗
MP 0.35	CVE-2011-0502	MIDI	0	✓
OV 1.04	CVE-2010-0688	ORB	0	✓

No seed file is needed

(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.**



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

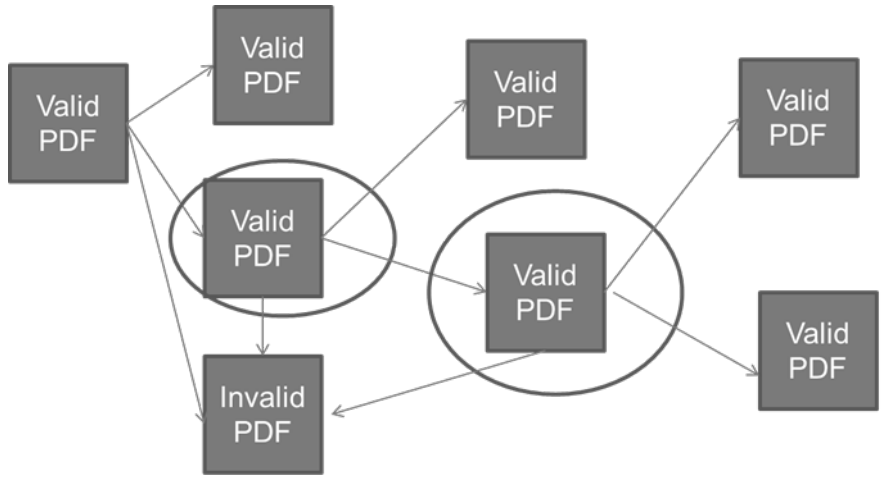
$\varphi_2 = \neg(x>y) \wedge (x+y>10)$

(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 = ∅ 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

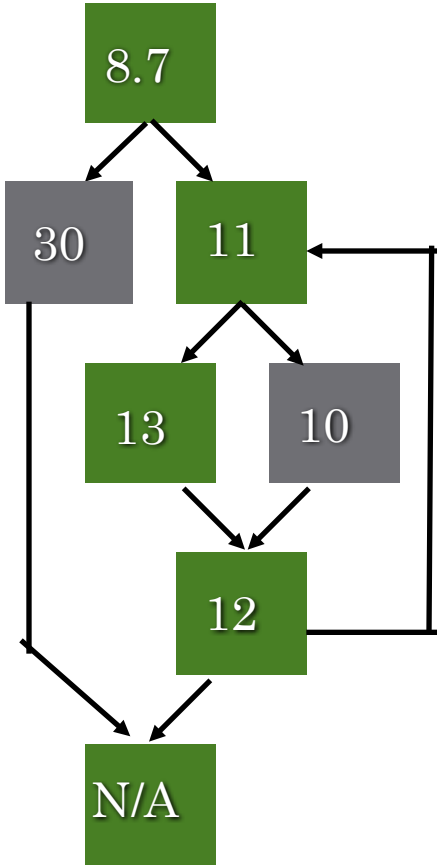


Exercises common path that rejects invalid PDF

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

Instrumentation

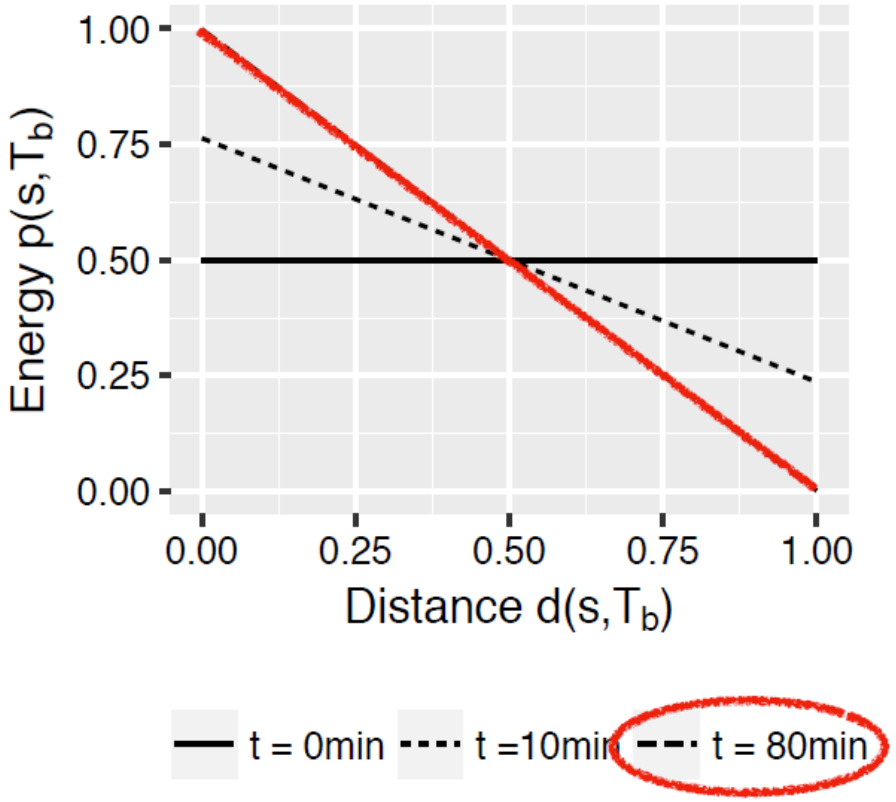
- **Function-level target distance** using call graph (CG)
- **BB-level target distance** using control-flow graph (CFG)
  1. Identify **target BBs** and assign **distance 0**
  2. Identify BBs that call **functions** and assign **10*FLTD**
  3. For **each BB**, compute harmonic mean of (length of shortest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



Results

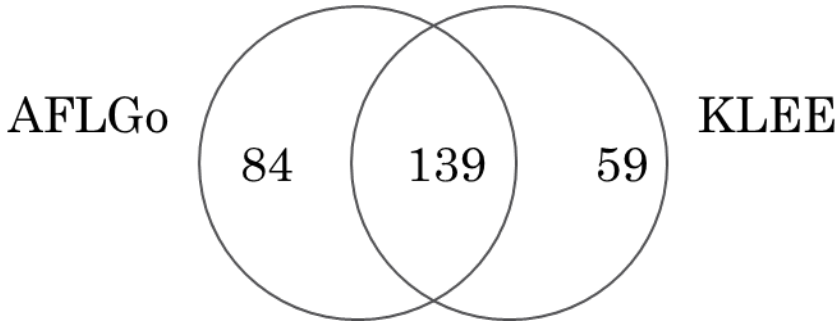
- **Patch Testing**: Reach changed statements
  - State-of-the-art in patch testing
    - **KATCH** (based on Klee symbolic exec. tool)
  - Experimental Setup
    - Reuse original **KATCH**-benchmark
    - Measure patch coverage (#changed BBs reached)
    - Measure vuln. detection (#errors discovered)

`175` patches in `diffutils`

`181` patches in `binutils`

Results

- **Patch Testing**: Reach changed statements
  - State-of-the-art in patch testing
    - **KATCH** (based on Klee symbolic exec. tool)
  - Patch Coverage (#changed BBs reached)
    - While we would expect Klee to take a substantial lead, **AFLGo outperforms KATCH** in terms of patch coverage.
    - **BUT**: **Together** they cover **42%** and **26%** more than **AFLGo** and **KATCH** individually. **They complement each other!** **AFLGo found 13 previously unreported bugs (7 CVEs)** in addition to 4 of the 7 bugs that were found by **KATCH**.



Crash Reproduction

Crash Reproduction: Exercise stack trace

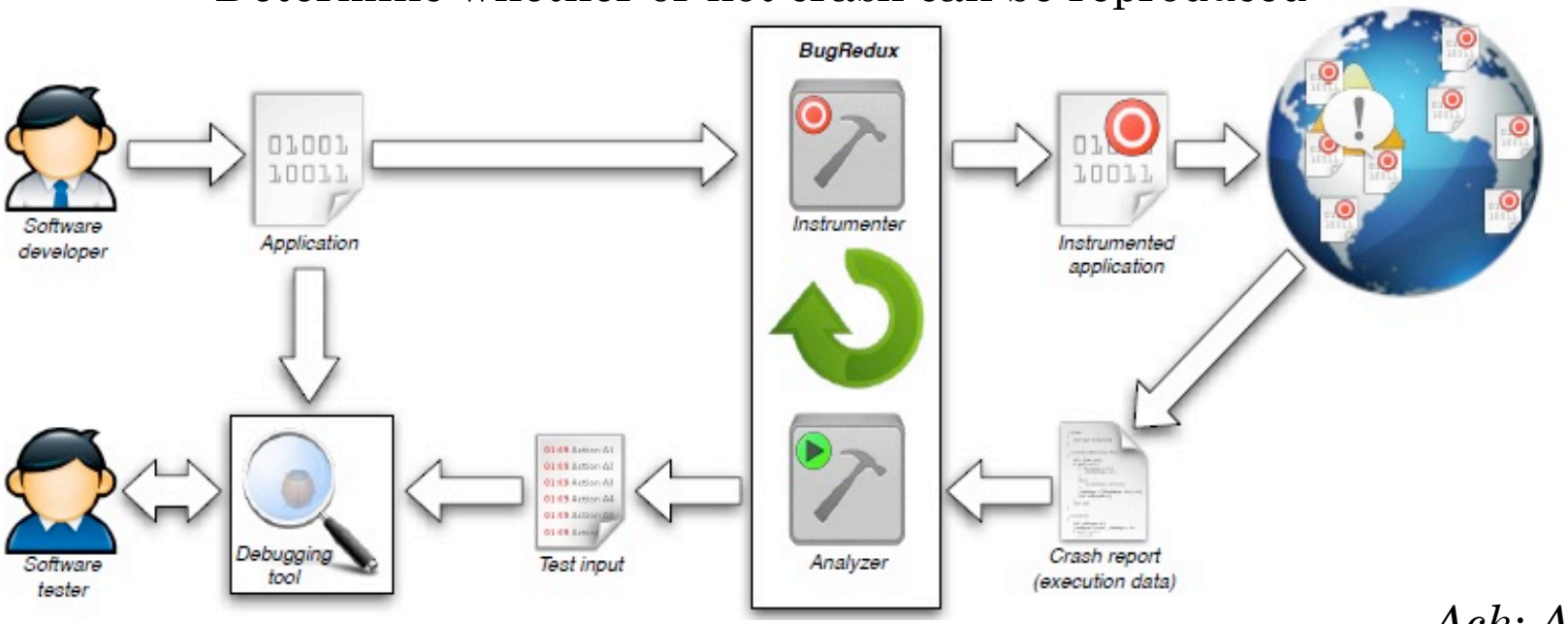
State-of-the-art in crash reproduction

BugRedux (based on Klee symbolic exec. tool)

Experimental Setup

Reuse original **BugRedux**-benchmark

Determine whether or not crash can be reproduced



Ack: Alex Orso (GATech)

Crash Reproduction

Crash Reproduction: Exercise stack trace

State-of-the-art in crash reproduction

BugRedux (based on Klee symbolic exec. tool)

Experimental Setup

Reuse original **BugRedux**-benchmark

Determine whether or not crash can be reproduced

Subjects	BugRedux	AFLGo	Comments
sed.fault1	✗	✗	Takes two files as input
sed.fault2	✗	✓	
grep	✗	✓	
gzip.fault1	✗	✓	
gzip.fault2	✗	✓	
ncompress	✓	✓	
polymorph	✓	✓	

AFLGo reproduces 3 times more crashes!

Summary of Results

- **Directed greybox fuzzer** (AFLGo) **outperforms symbolic execution-based directed fuzzers** (KATCH & BugRedux)
  - in terms of **reaching more target locations** and
  - in terms of **detecting more vulnerabilities**,
  - on their own, original benchmark sets.

- **Integrated as OSS-Fuzz fork** (AFLGo for Continuous Fuzzing)
- **17 CVEs** reported (e.g., libxml)
- **39 bugs** found in security-critical libraries

Details in CCS17 paper: Directed Grey-box Fuzzing

https://github.com/aflgo/aflgo

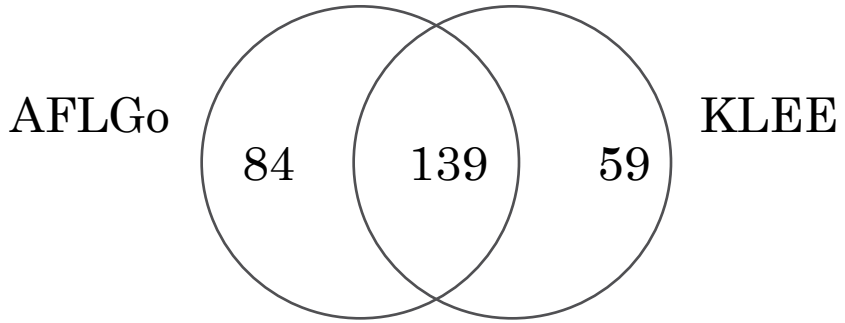
In this talk …

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 search**



Grey-box and White-box!

Utility	ELOC	AFL	FAST	KLEE	Utility	ELOC	AFL	FAST	KLEE	Utility	ELOC	AFL	FAST	KLEE	Utility	ELOC	AFL	FAST	KLEE	Utility	ELOC	AFL	FAST	KLEE
base64	104	90.0	92.0		echo	97	64.0	81.0		logname	25	96.0	96.0		pwd	129	33.0	33.0		tty	32	96.0	100.0	
basename	55	100.0	100.0		expand	72	94.0	94.0		ls	1547	61.7	33.0		seq	282	89.6	71.0		uname	85	82.0	91.0	
cat	219	83.8	78.0		expr	334	76.0	70.0		md5sum	297	53.4	69.0		shuf	223	86.0	60.0		unexpand	104	93.9	92.0	
cksum	61	86.0	91.0		factor	510	52.8	23.0		nl	194	84.6	94.0		sort	1702	68.1	48.0		uniq	195	89.6	89.0	
comm	146	95.0	94.0		fmt	315	90.0	88.0		od	637	89.1	87.0		split	641	76.3	30.0		uptime	70	92.0	92.0	
csplit	521	79.3	77.0		fold	104	94.0	96.0		paste	189	91.0	93.0		stat	555	68.1	61.0		users	53	97.1	90.0	
cut	188	87.3	93.0		head	382	68.0	70.0		pr	854	95.0	67.0		tac	238	70.1	71.0		wc	308	72.9	76.0	
df	664	74.3	49.0		hostid	23	100.0	100.0		printenv	41	100.0	100.0		tail	841	55.1	52.0		who	279	83.0	86.0	
dirname	35	100.0	100.0		id	160	85.0	83.0		printf	224	88.7	95.0		tr	620	74.8	75.0		whoami	28	96.0	96.0	
du	347	65.3	68.0		join	430	90.5	85.0		ptx	657	93.6	64.0		tsort	193	69.3	96.0		**Average**	**327**	**82.1**	**78.3**	

Similar coverage observed in both approaches for now.

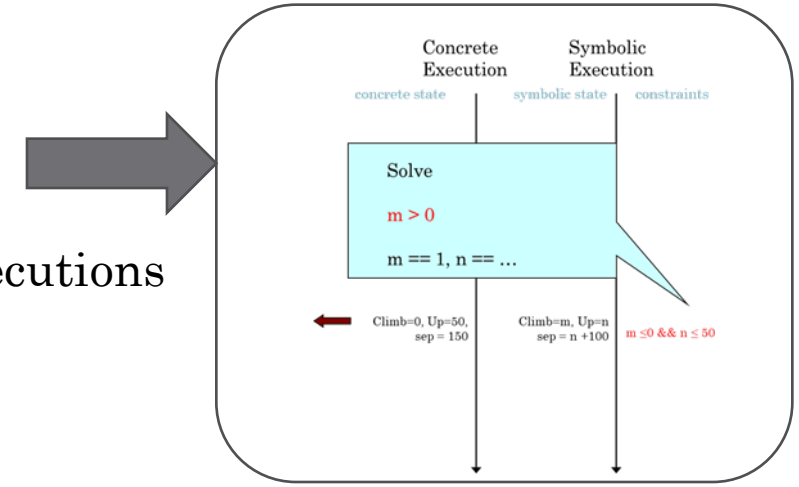
Role of benchmarks remains important, so that it is not over-fitted to one approach.

More details appear in the paper(s), including the TSE18 paper
http://www.comp.nus.edu.sg/~abhik/pdf/TSE18.pdf

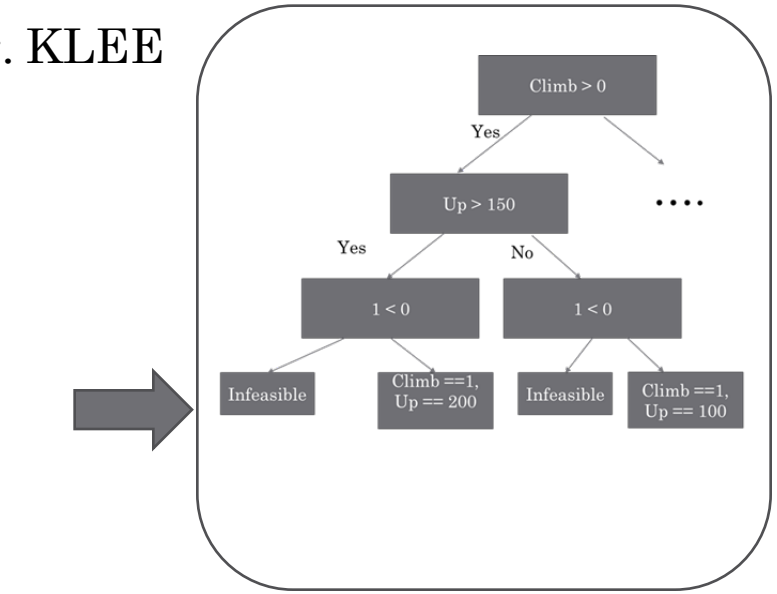
Reflections on Symbolic Execution

Bug Finding

- Concolic execution: supporting *real* executions [Directed Automated Random Testing]

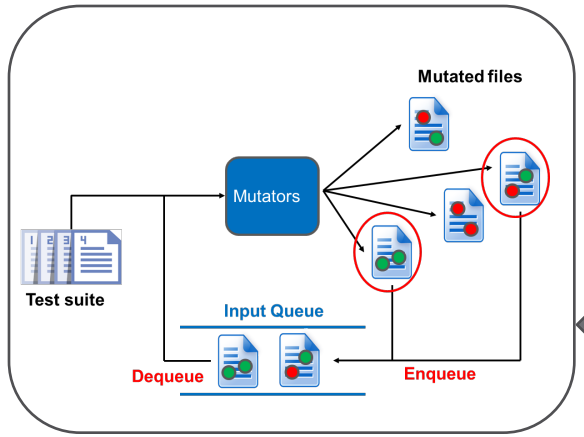


- Symbolic execution tree construction e.g. KLEE [Modeling system environment]



- Grey-box fuzz testing for systematic path exploration inspired by concolic execution

AFLFast

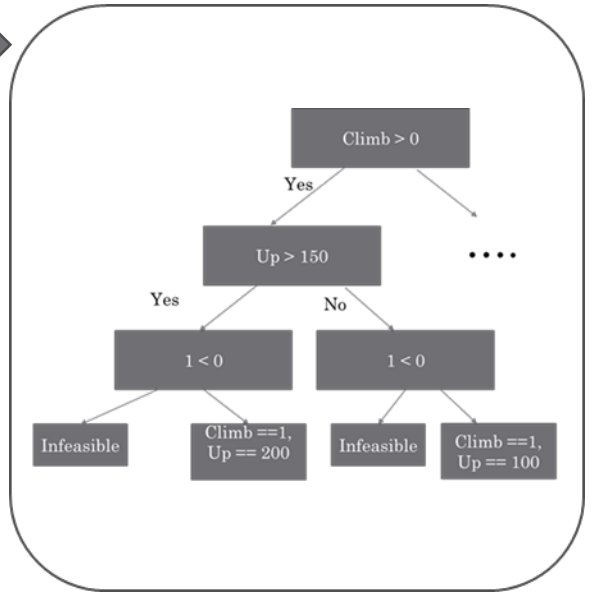


Reflections on Symbolic Execution

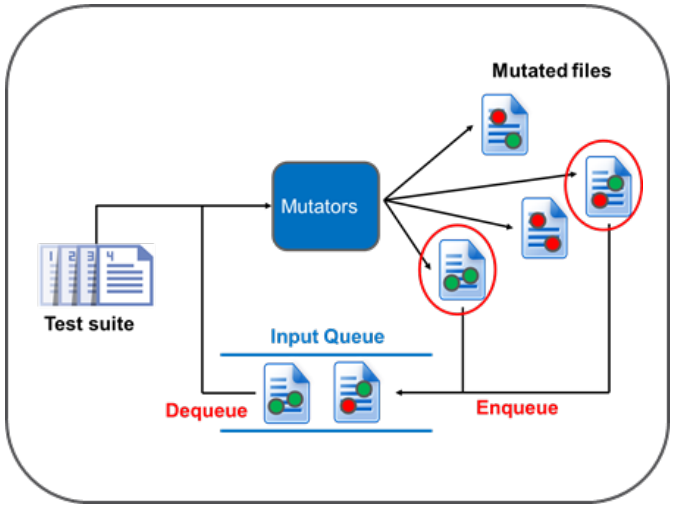
Reachability Analysis

Reachability of a location in the program

- Traverse the symbolic execution tree using search strategies e.g. KATCH



- Encode it as an optimization problem inside the genetic search of grey-box fuzzing AFLGo

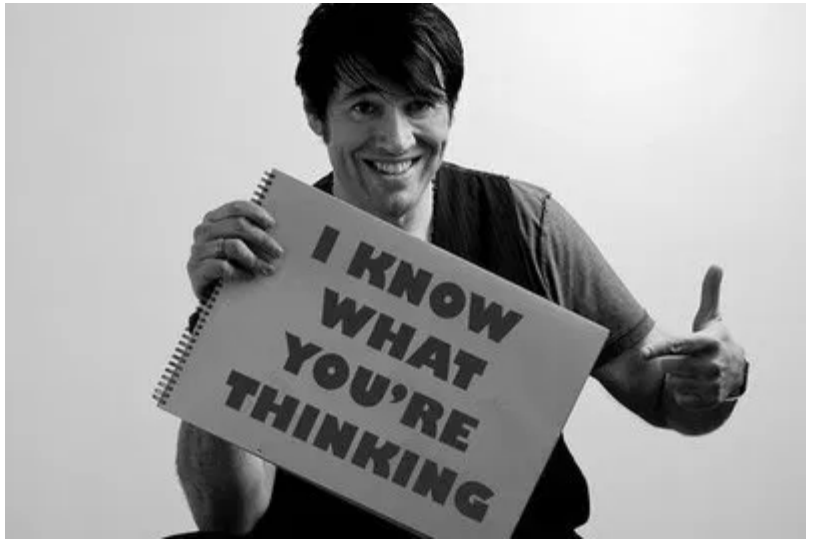


Reflections on Symbolic Execution

Specification Inference

(application: localization, self-healing)

In the absence of formal specifications, analyze the buggy program and its artifacts such as execution traces via various heuristics to glean a specification about how it can pass tests and what could have gone wrong!



Relevant Research Results

50 CVEs in well-fuzzed programs like FFMPEG.

Directed Greybox Fuzzing (PDF)
Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, Abhik Roychoudhury
24th ACM Conference on Computer and Communications Security (CCS) 2017.

Coverage-based Greybox Fuzzing as Markov Chain (PDF)
Marcel Böhme, Van Thuan Pham, Abhik Roychoudhury
23rd ACM Conference on Computer and Communications Security (CCS) 2016, Also in IEEE Transactions in Software Engineering (TSE) 2018, paper

Model-based Whitebox Fuzzing for Program Binaries (pdf)
Van Thuan Pham, Marcel Böhme, Abhik Roychoudhury
IEEE/ACM International Conference on Automated Software Engineering (ASE) 2016.

Hercules: Reproducing Crashes in Real-World Application Binaries (PDF)
Van Thuan Pham, Wei Boon Ng, Konstantin Rubinov, Abhik Roychoudhury
ACM/IEEE International Conference on Software Engineering (ICSE) 2015.

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

ACKNOWLEDGEMENT: National Cyber Security Research program from NRF Singapore http://www.comp.nus.edu.sg/~tsunami/ and DSO National Labs

A note for all students here

Happy to talk to you now, or later by email **abhik@comp.nus.edu.sg**

You can look up my webpage **http://www.comp.nus.edu.sg/~abhik**

I am happy to discuss my past as well as ongoing projects with you.

Will again talk on Wednesday morning – on using symbolic execution for program debugging and repair. The slides have been shared with you, and you can get a sneak preview of this research from
http://www.comp.nus.edu.sg/~abhik/projects/Repair/index.html

Let us catch up.