Enhancing Symbolic Fuzzing with Learning

Kang Li
University of Georgia

IMG source: www.computerhope.com

Joint work with Qixue Xiao, Yu Chen, Chenggang Wu, Deyue Zhang

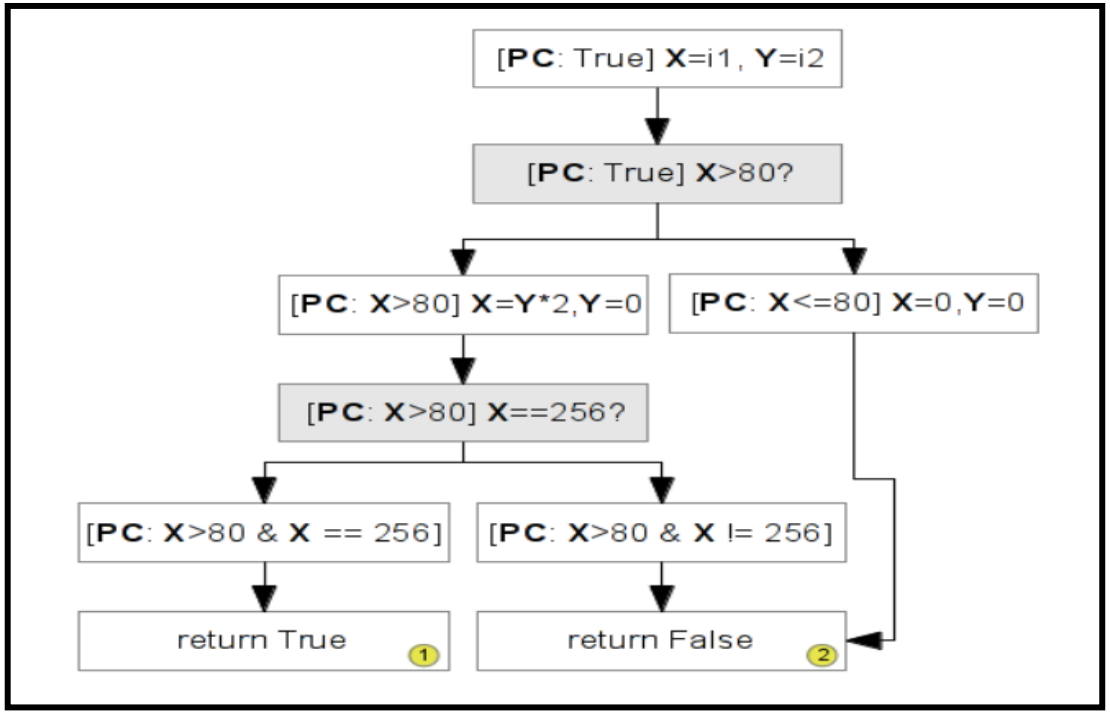
About Me

- Professor at the University of Georgia
- Founder of the *Disekt, SecDawgs* CTF Teams
- Founding Mentor of *xCTF* and *Blue-lotus* Team
- 2016 DARPA Cyber Grand Challenge Finalist

disekt
CTF Team

Concrete vs. Symbolic Execution

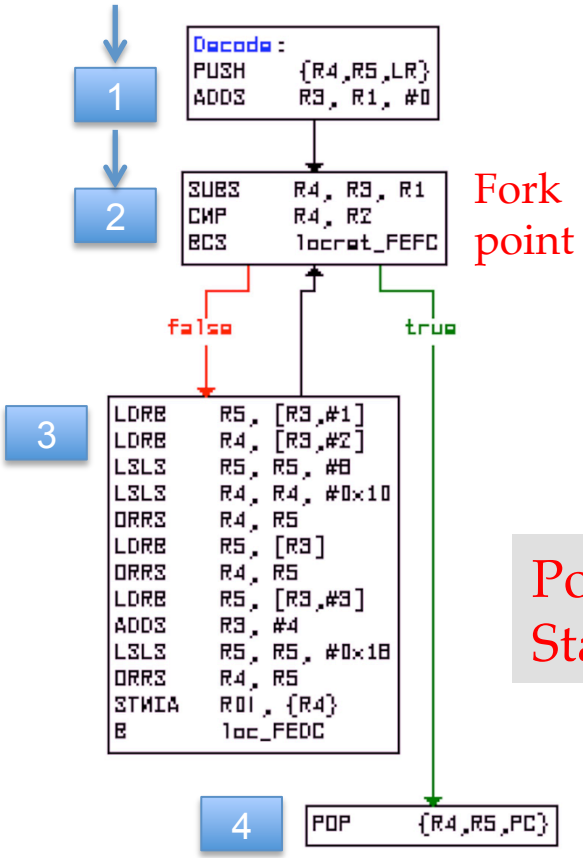
```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

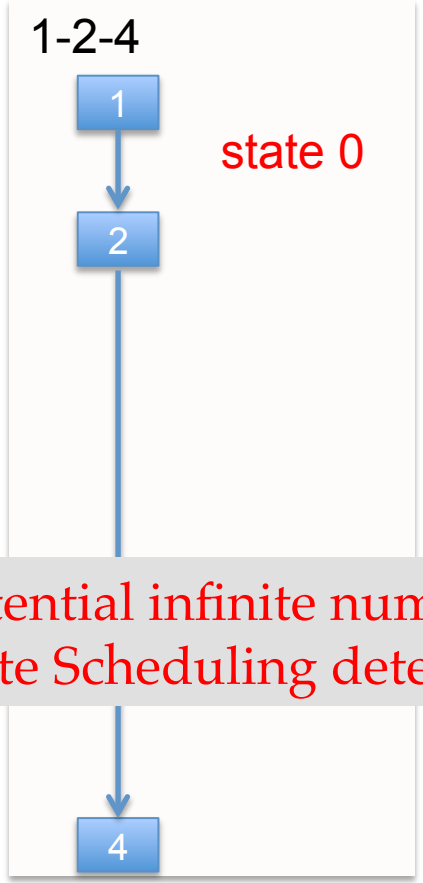


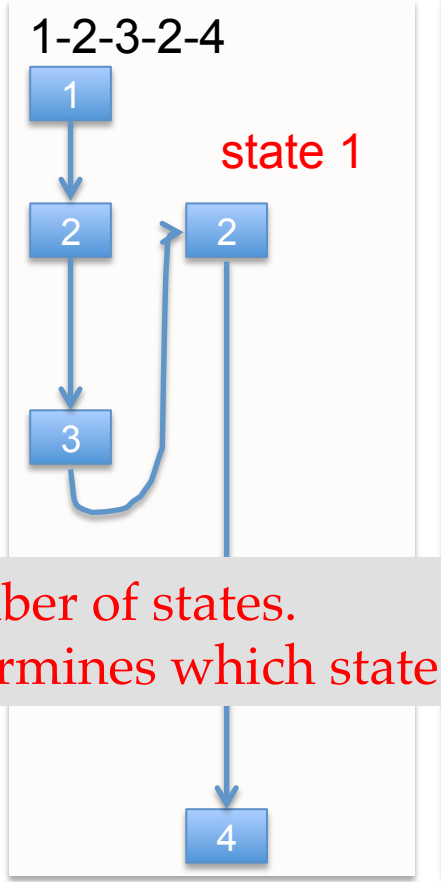
Symbolic Fuzzing (SymExec for Bug Finding)

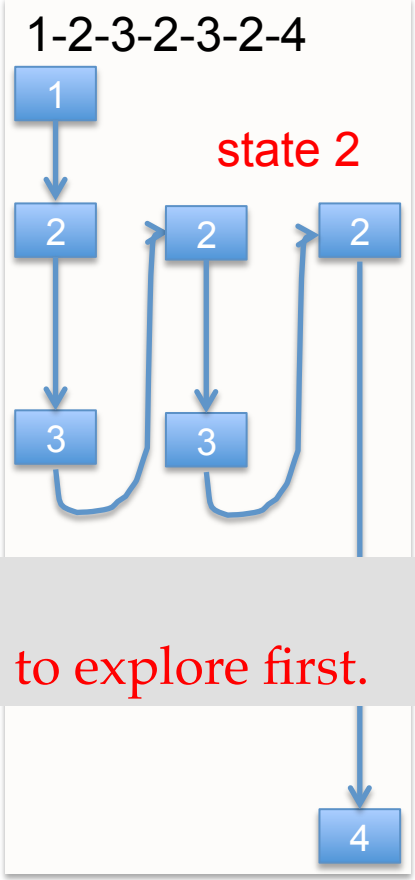
- Use symbolic variables as inputs
  - Propagation symbolic inputs
- Construct path constraints
  - Evaluate branch condition at potential branch points
  - Rely on constraint solver to determine path feasibility
  - Fork on viable branch, append path constraint
- Explore feasible paths & check for property violations
  - Exploration order determined by scheduling algorithms

Symbolic Exec Engine and State Scheduling

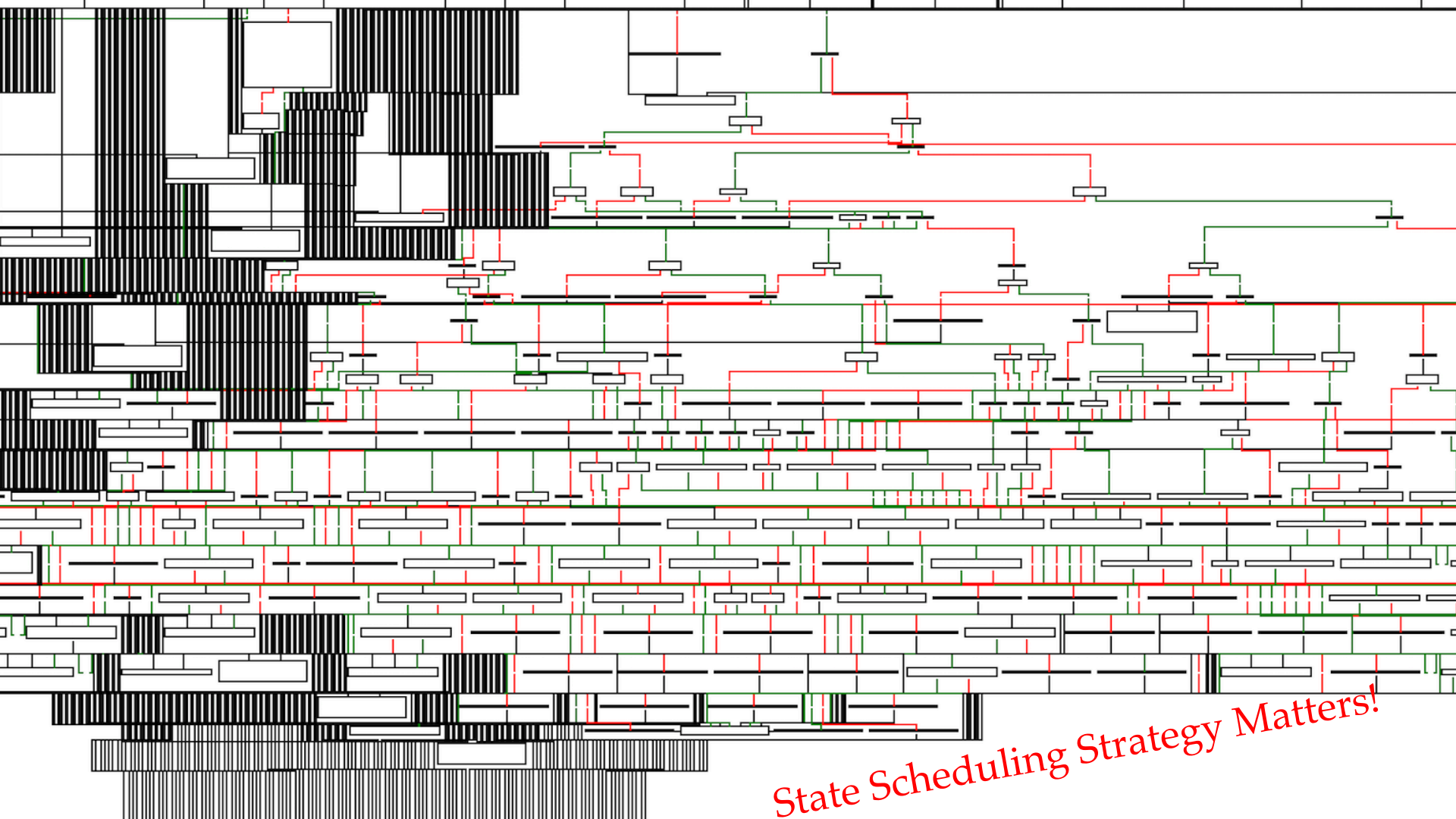








Potential infinite number of states.
State Scheduling determines which state to explore first.


State Scheduling Strategy Matters!

A Practical Challenge to Symbolic Exec

```
void foo()
{
    ...

    for (i = 0; i < sym2; i++)
        cnt++;

    if (cnt > len)
        bar();
    else
        zoo();

    ...
}
```

Sym2 is a symbolic input

The loop bound depends on symbolic variable
→ path explosion !

Won't reach bar() when the loop
finishes with relatively fewer iterations

- Difficult to reach high code coverage in a timely manner

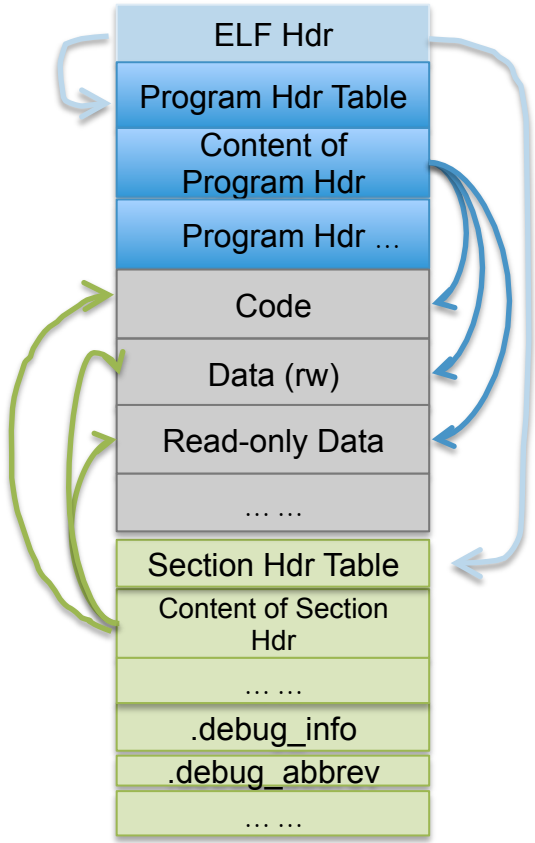
Programs often Process Input in Phases (dwarfdump)

1. ELF Header Parsing
   elf_begin() @ dwarfdump.c
   is_it_known_elf_header() etc. @dwarfdump.c

2. Section Header Parsing
   process_section_groups() @ dwarfdump.c

3. Processing CU header
   dwarf_next_cu_header_d() @ print_die.c

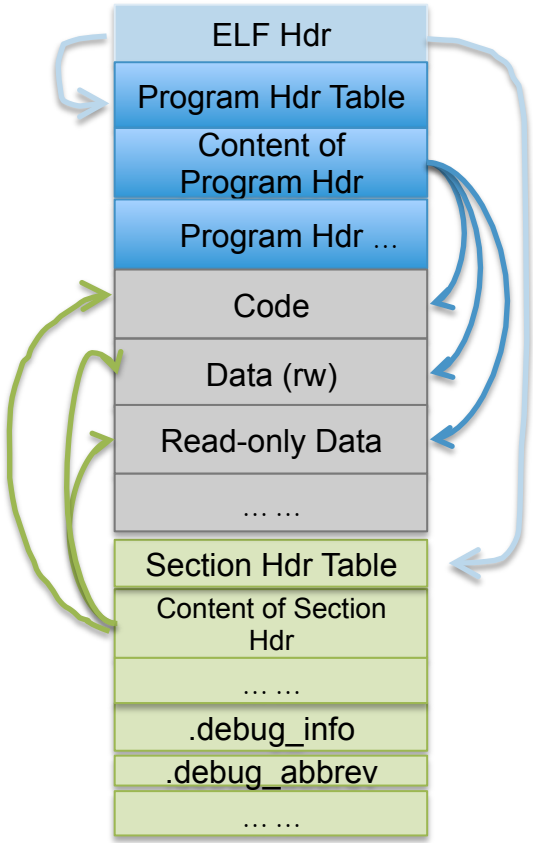
4. Debug Symbol Array Parsing
   dwarf_get_abbrev_code() @ print_abbrevs.c



Trapping Phase Example (Parsing ELF files)

```
6003 process_section_groups (FILE * file)
6004 {
6005   Elf_Internal_Shdr * section;
6014
6015   /* Don't process section groups unless needed. */
6016   if (!do_unwind && !do_section_groups)
6017     return 1;
6018
6019   if (elf_header.e_shnum == 0)
6020    {
6021      if (do_section_groups)
6022        printf (_("\nThere are no sections to group.\n"));
6023
6024      return 1;
6025    }
...
6046   for (i = 0, section = section_headers;
6047        i < elf_header.e_shnum;
6048        i++, section++) {}
...
6272 }
```

Symbolic Var as loop bound!

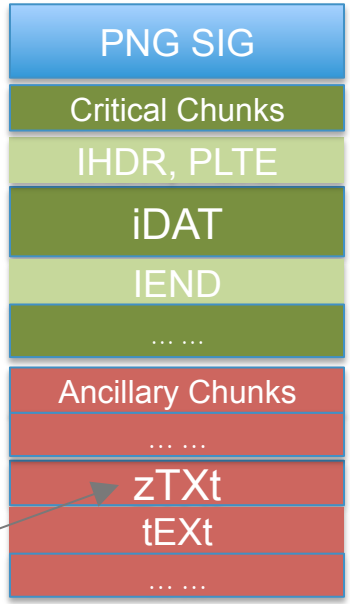


PNG File Parsing

```
1231 png_size_t /* PRIVATE */
1232 png_check_keyword(png_structp png_ptr,
         png_charp key, png_charpp new_key)
1233 {
1234    png_size_t key_len;
1235    png_charp kp, dp;
......
1259    for (kp = key, dp = *new_key; *kp != '\0'; kp++, dp++)
1260    {
......
1279    }

1283    kp = *new_key + key_len - 1;
1284    if (*kp == ' ')
1285    {
1288       while (*kp == ' ')
1289       {
1290            *(kp--) = '\0';
1291            key_len--;
1292       }
1293    }
```

***kp is determined by input file, thus symbolic bound!**

Programs often execute in phases; Symbolic exec often traps in a phase



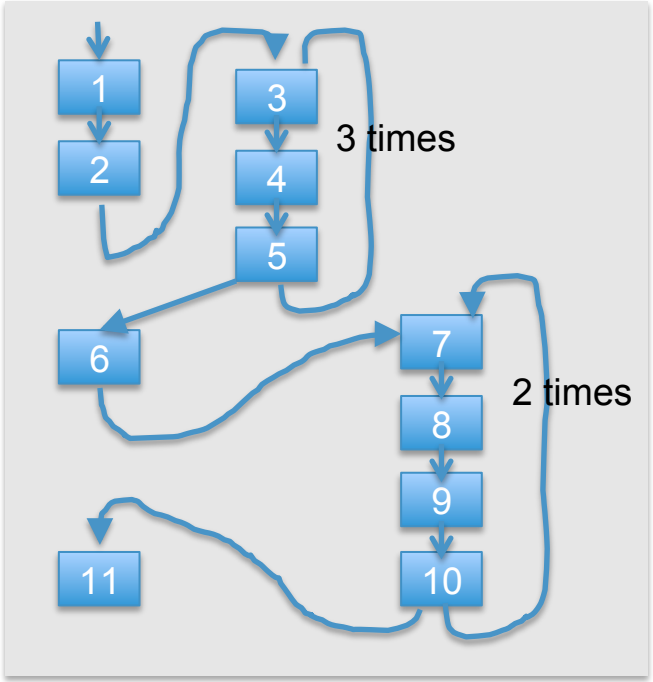
Keyword	Null separator	Compression flag	Compression method	Language tag	Null separator	Translated keyword	Null separator	Text
1~79 bytes	1 byte	1 byte	1 byte	0 or more bytes	1 byte	0 or more bytes	1 byte	0 or more bytes

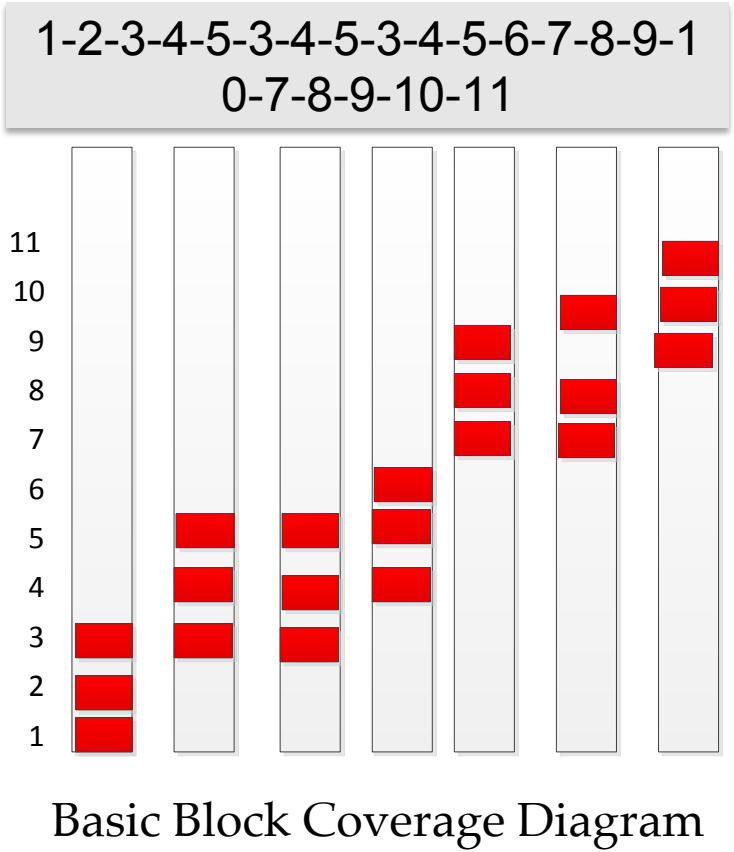
Symbolic Exec Exploration Strategies

- Common
  - DFS, BFS
  - Random
    - Path
    - State
- Heuristic
  - Coverage-guided
    e.g. distance to uncovered code
  - Random + Timeout
  - Random + Covnew

None of These Strategy Ensure High Code Coverage!
(as shown by our empirical results)

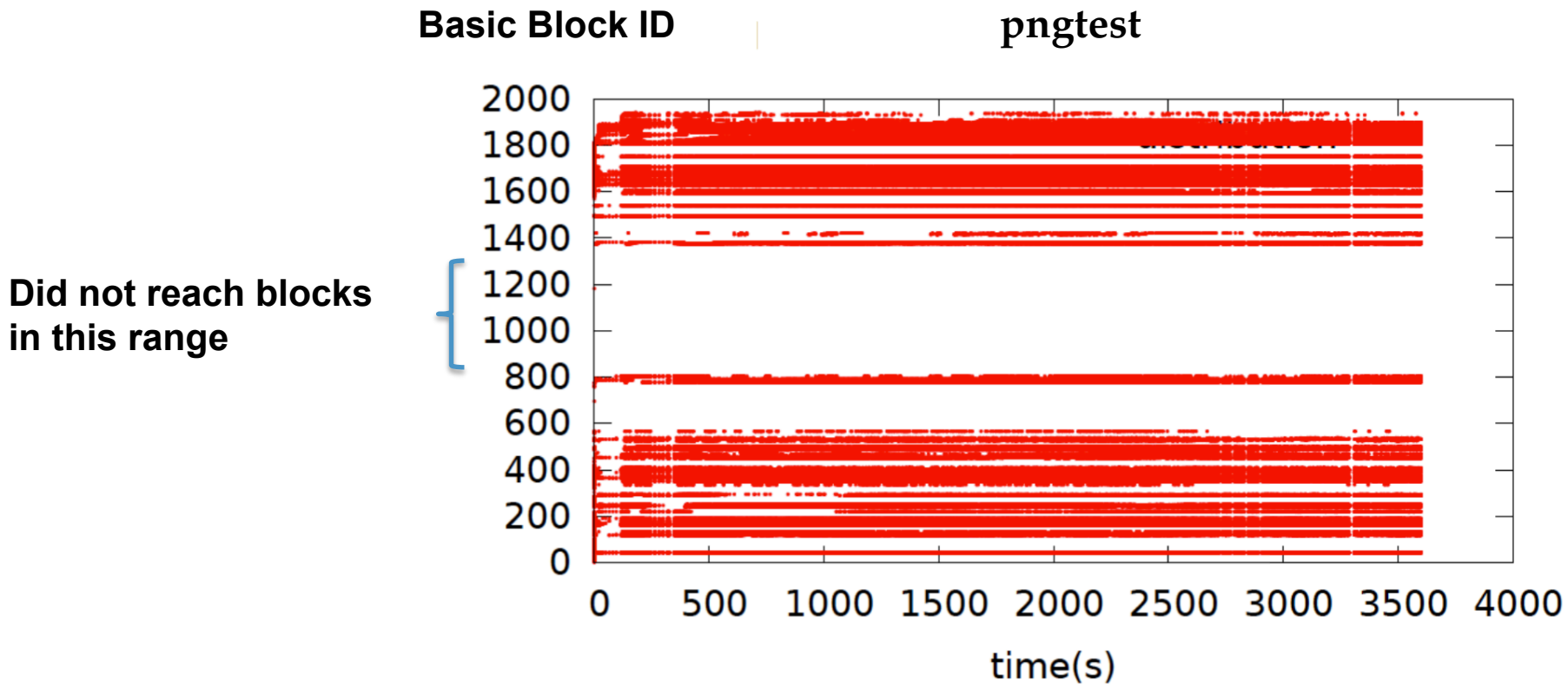
Measuring Test Coverage



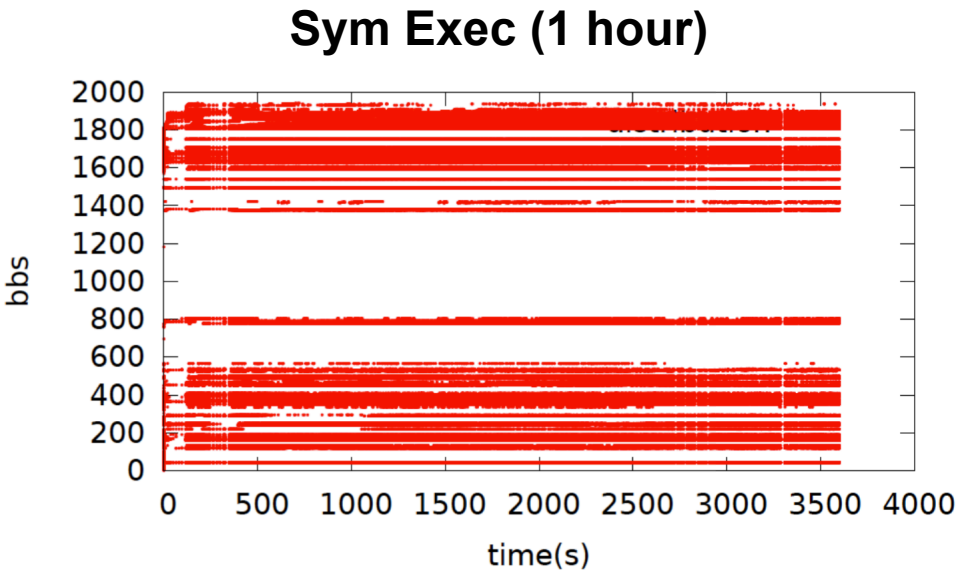


Basic Block Coverage Diagram

Basic Block Coverage by Symbolic Exec (pngtest)



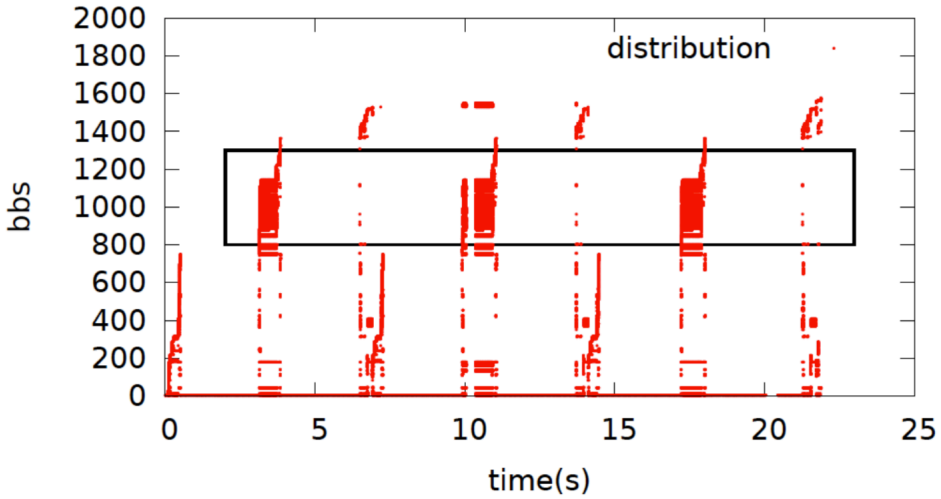
Coverage Comparison (pngtest)



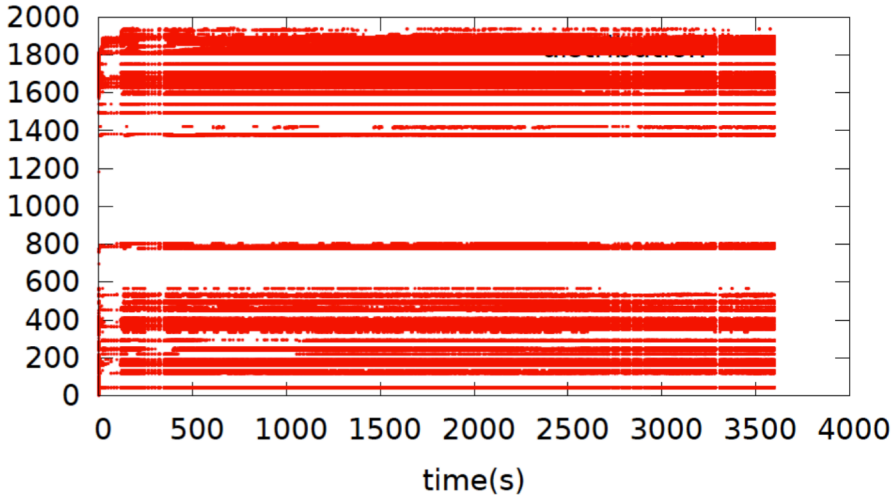
Coverage Comparison (pngtest)

20+ seconds vs. 1 hour

Concrete Exec



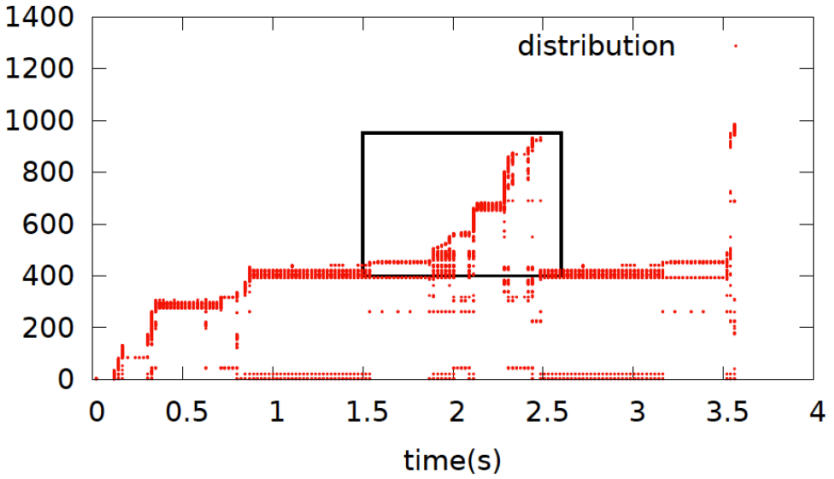
Sym Exec



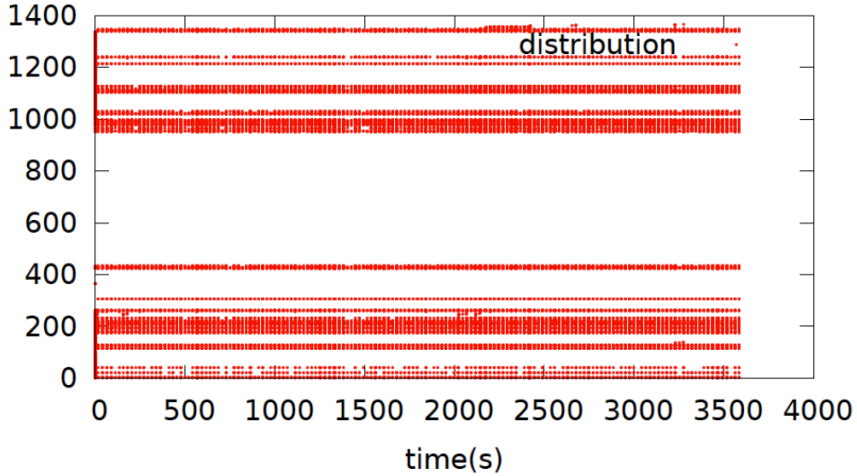
Coverage Comparison (gif2tiff)

3.5 seconds vs. 1 hour

Concrete Exec

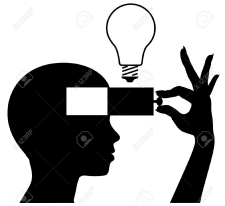


Sym Exec

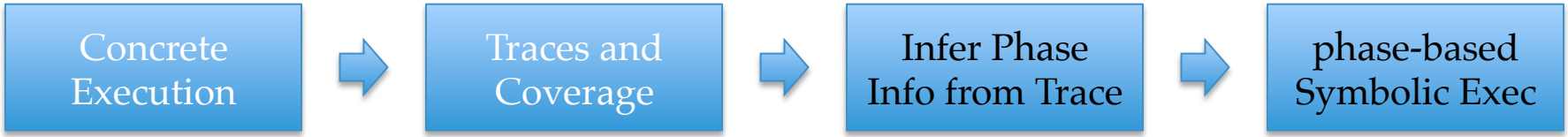


Symbolic execution fails to cover blocks reached by concrete execution

Infer Program Phases



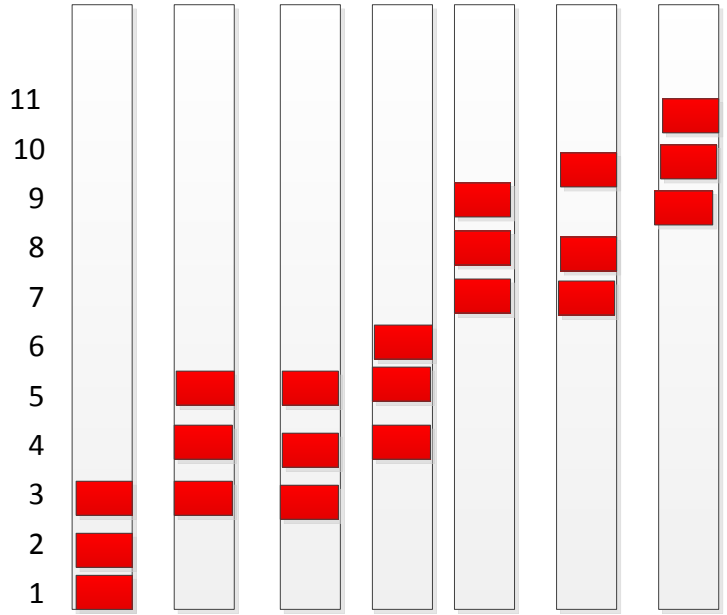
- Learn program phase information from concrete execution
- Phase represents the temporal locality of a program execution



Learn Phase Info from Concrete Execution

- Log concrete execution trace (in the form of basic block transitions)

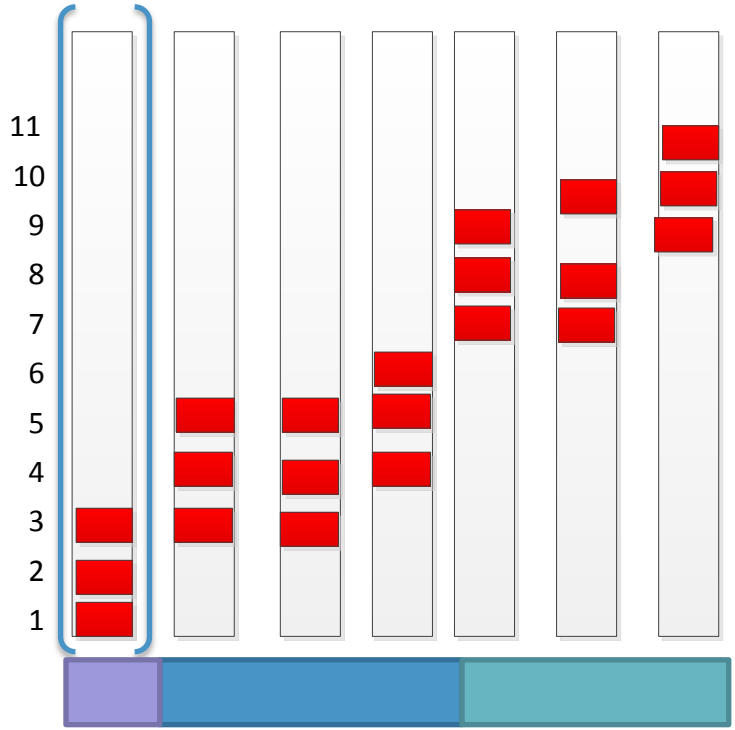
1-2-3-4-5-3-4-5-3-4-5-6-7-8-9-1
0-7-8-9-10-11



Phase Inference Algorithm

Coverage info

C_0	C_1	C_2	C_3	C_4	C_5	C_6
0	0	0	0	0	0	1
0	0	0	0	0	1	1
0	0	0	0	1	0	1
0	0	0	0	1	1	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	1	1	1	0	0	0
0	1	1	1	0	0	0
1	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	0	0	0	0	0



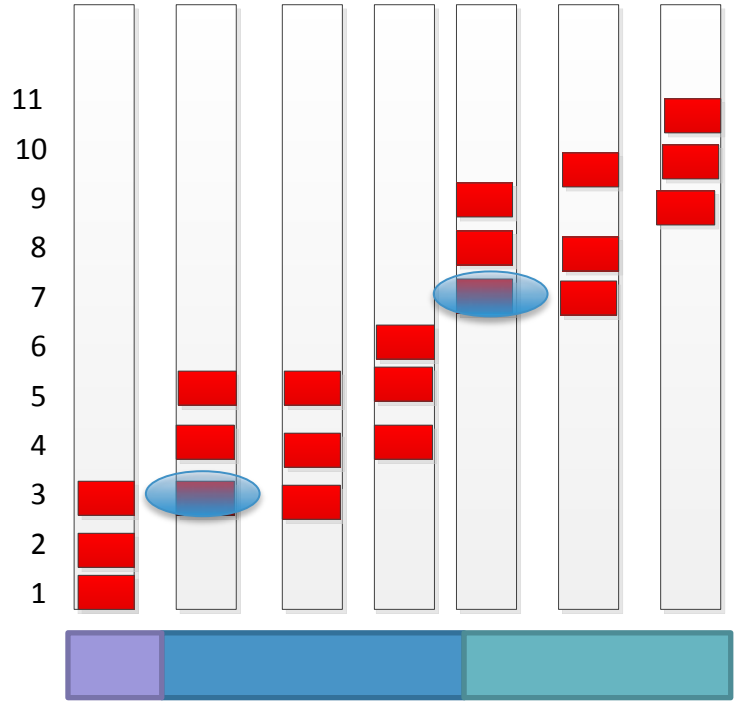
Apply **k-means** to cluster these vectors

Select Key State to Represent Each Phase

- Key State:

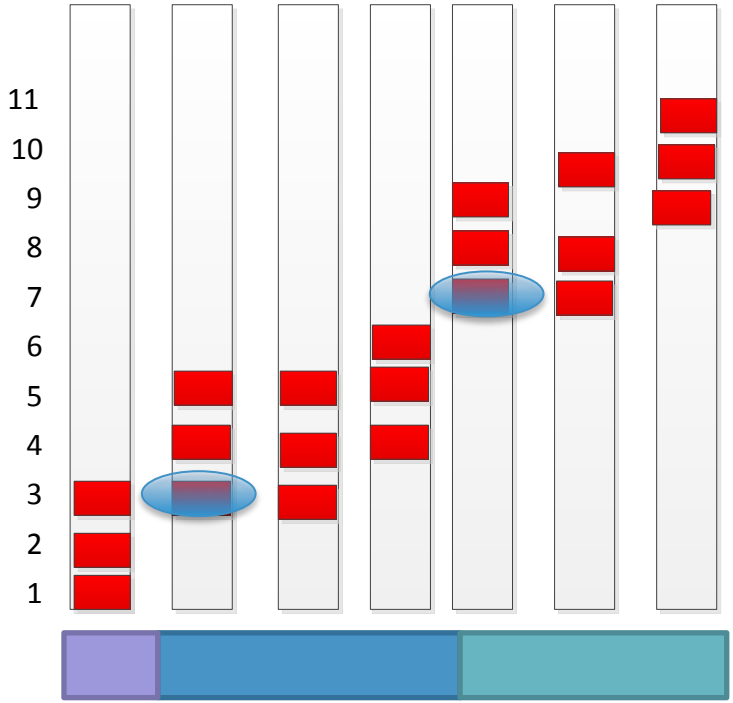
Input that can lead a program to leave a phase and enter a new phase.

Represented by a pair of path constraint and a concrete input.

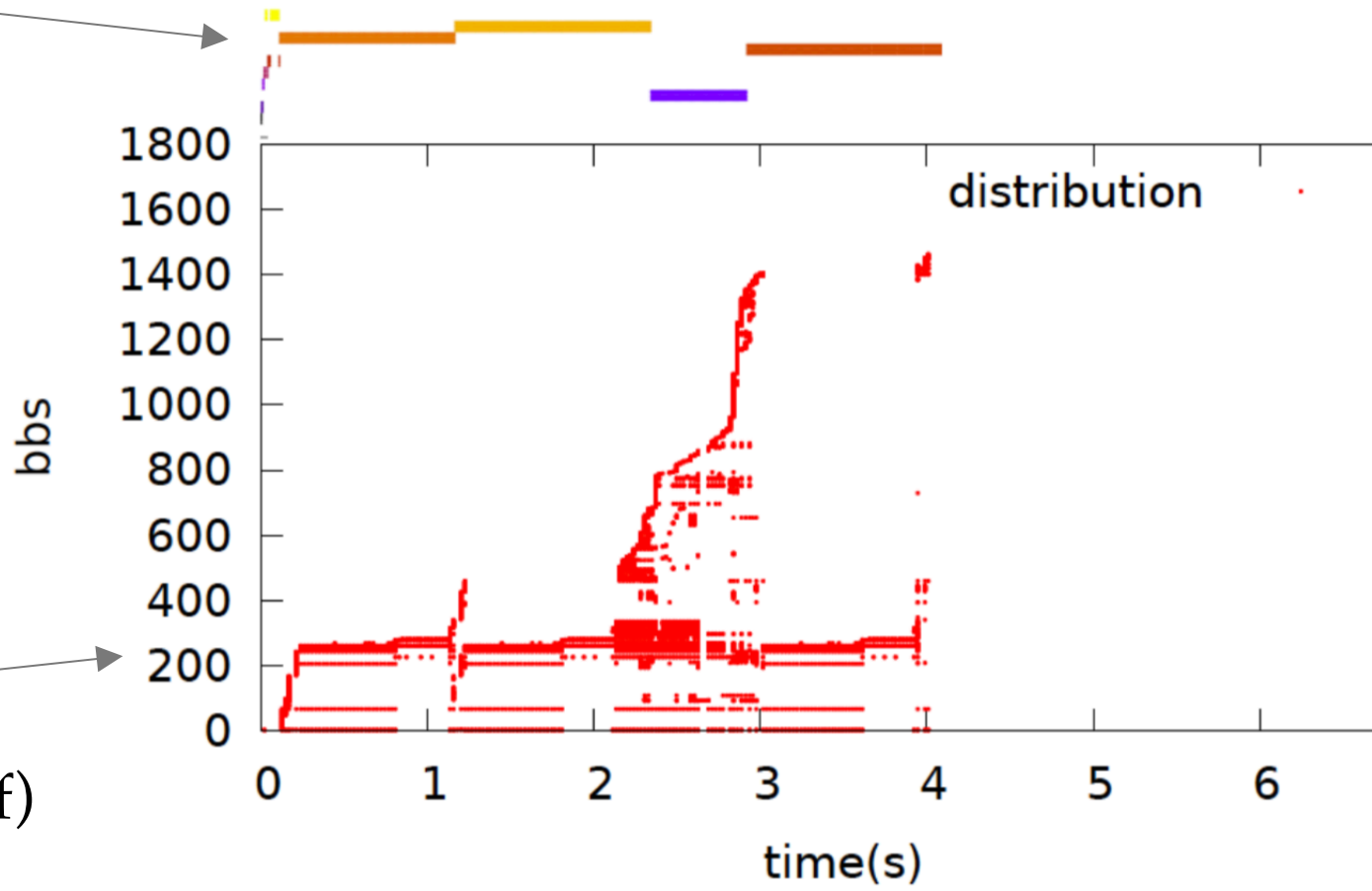


pbSE: phase-based Symbolic Execution

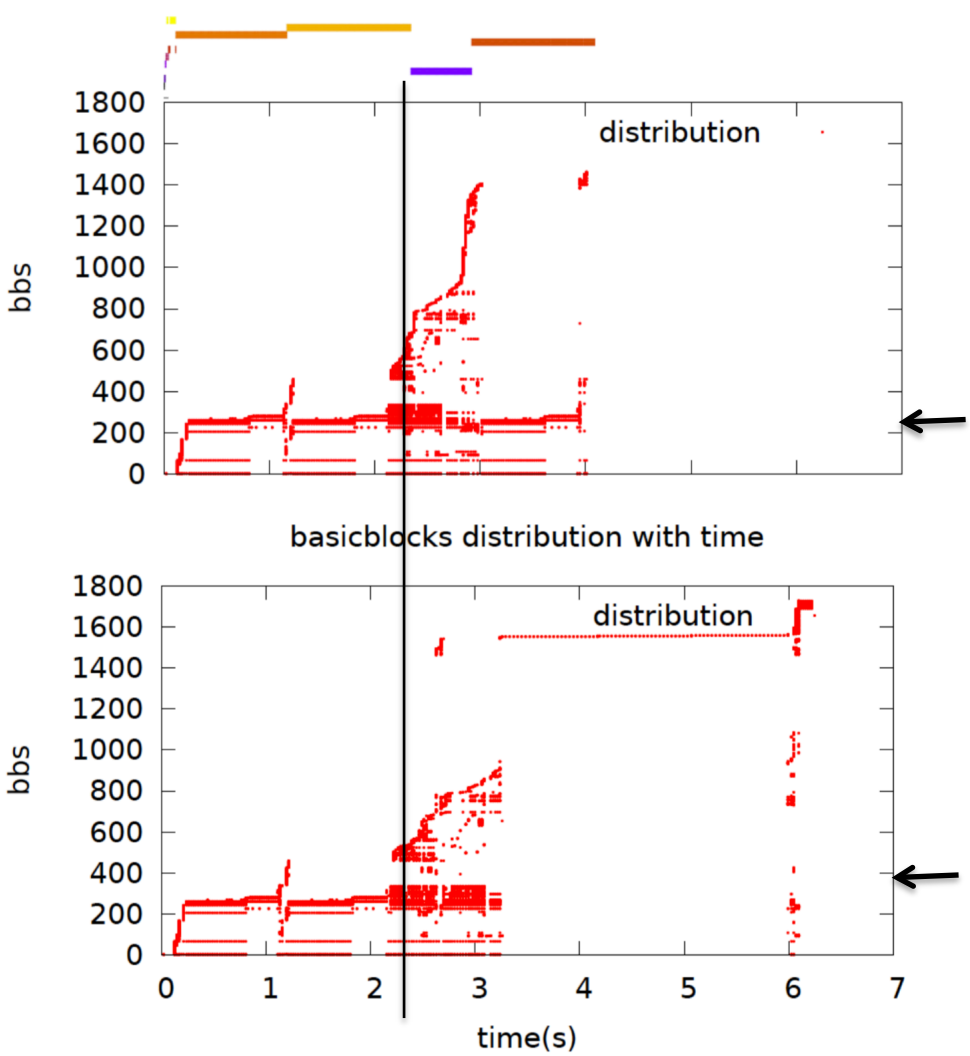
- Concolic Execution to these critical transition blocks without forking
- Symbolic exec start after reaching these critical blocks
- Related Work:
  - SAGE from Microsoft -- concolic execution for test case generation, but does not strategically select fork points



The phases inferred from trace



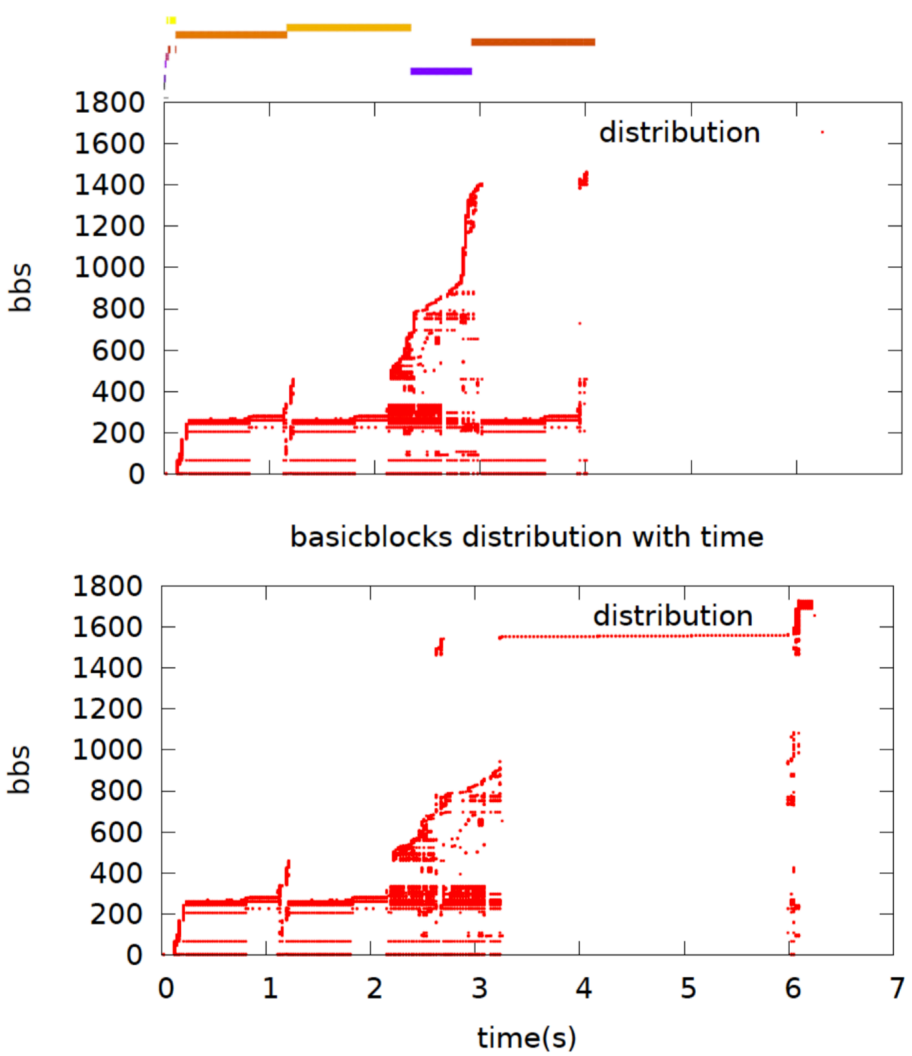
A concrete execution trace of tiff2rgba (libtiff)



Bug detected in libtiff by pbSE (starting from a later phase)

The initial execution trace from which pbSE learned phase information

The execution trace generated using the detected “buggy input”

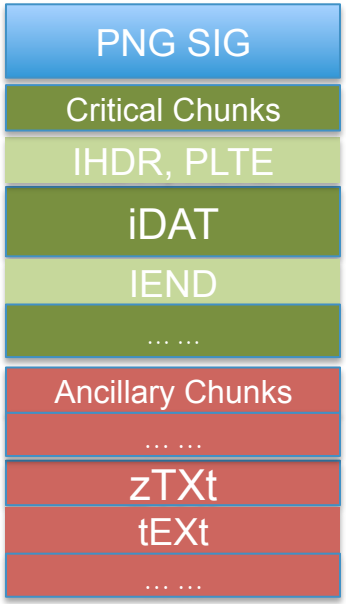



```
 1 #define DECLAREContigPutFunc(name) \
 2 static void name(\
 3     TIFFRGBAImage* img, \
 4     uint32* cp, \
 5     uint32 x, uint32 y, \
 6     uint32 w, uint32 h, \
 7     int32 fromskew, int32 toskew, \
 8     unsigned char* pp \
 9 )

1640 DECLAREContigPutFunc(putcontig8bitCIELab)
1641 {
1642     float X, Y, Z;
1643     uint32 r, g, b;
1644     (void) y;
1645     fromskew *= 3;
1646     while (h-- > 0) {
1647         for (x = w; x-- > 0;) {
1648             TIFFCIELabToXYZ(img->cielab,
1649                     (unsigned char)pp[0],
1650                     (signed char)pp[1],
1651                     (signed char)pp[2],
1652                     &X, &Y, &Z);
1653             TIFFXYZToRGB(img->cielab, X, Y, Z, &r, &g, &b);
1654             *cp++ = PACK(r, g, b);
1655             pp += 3;
1656         }
1657         cp += toskew;
1658         pp += fromskew;
1659     }
1660 }
```

- Variable *h,w* are symbolic.
- Overflow occurs when *h*w*3 > pp allocation size*

Code snippets of libtiff.c

PNG File Format




```
1231 png_size_t /* PRIVATE */
1232 png_check_keyword(png_structp png_ptr,
         png_charp key, png_charpp new_key)
1233 {
1234    png_size_t key_len;
1235    png_charp kp, dp;
......
1259    for (kp = key, dp = *new_key; *kp != '\0'; kp++, dp++)
1260    {
......
1279    }

1283    kp = *new_key + key_len - 1;
1284    if (*kp == ' ')
1285    {
1288       while (*kp == ' ')
1289       {
1290            *(kp--) = '\0';
1291            key_len--;
1292       }
1293    }
```

trapping phase

underflow occurs when *kp is just " "

Code snippets of libpng.c

Test Results of phase-based Symbolic Exec

Target Software	Test Driver	Bug Phase / Total Phases	# of Unique Bugs
libpng	pngtest	5 / 9	**2**
libtiff	gif2tiff, tiff2rgba, tiff2bw	3 / 5	**5**
libdwarf	dwarfdump	6 / 10	**10**
binutils	readelf	5 / 7	**4**

- 21 previous unknown bugs were discovered over 2015~2017
- All bugs occurred in deep paths (on a relatively late phase)
- Details are in an upcoming paper in IEEE/IFIP DSN, June 2017

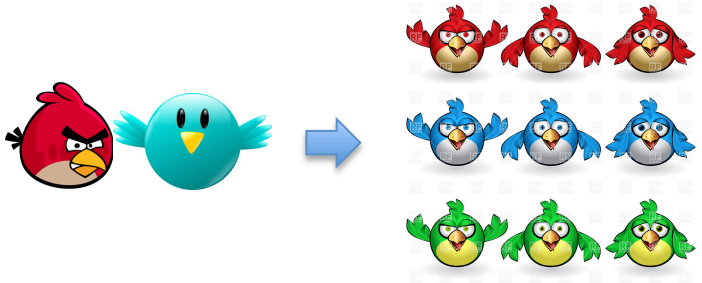
"pbSE: Phase-based Symbolic Execution", by Qixue Xiao, Yu Chen, Chenggang Wu, Kang Li et.al.

Where Can We Apply Learning?

- In the vulnerability discovery context
  - Concrete Testing
  - Symbolic Testing  *bird*
    - Program phase information

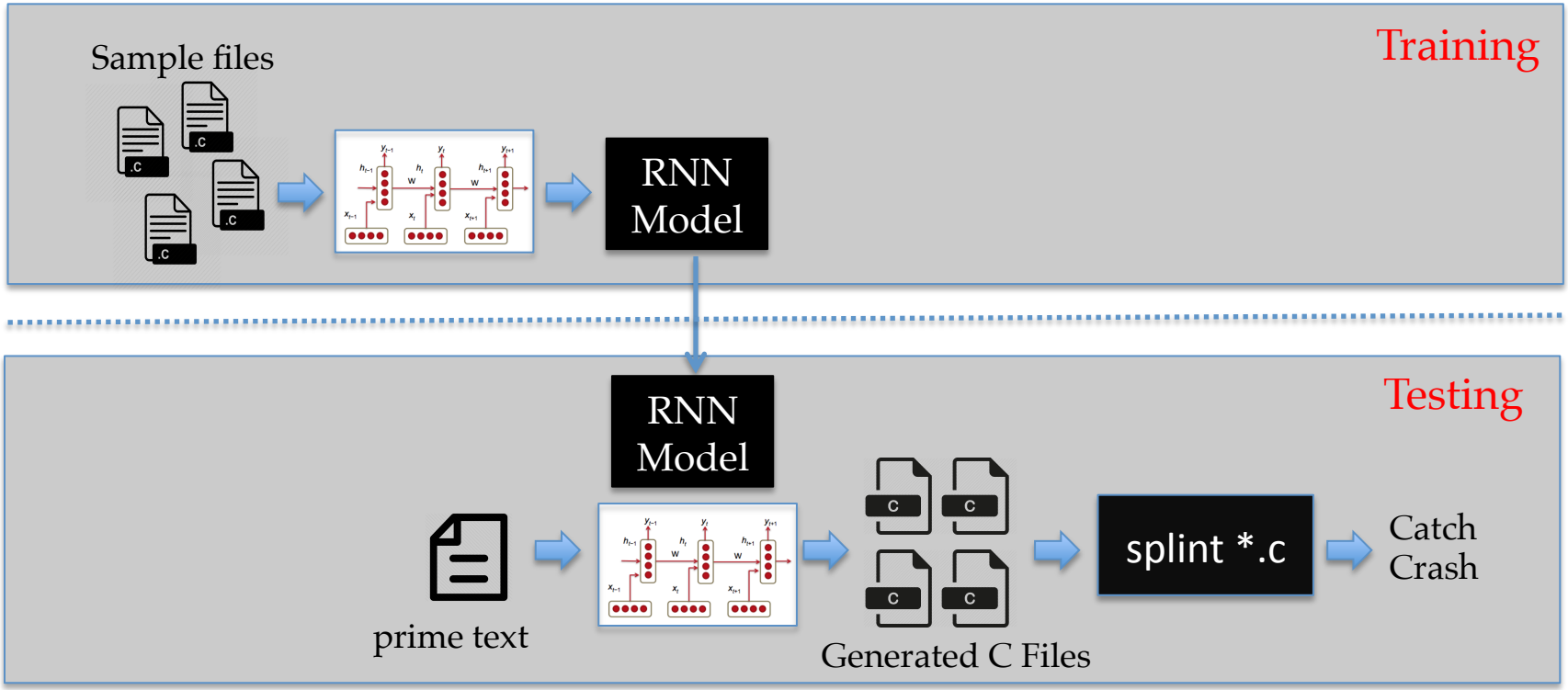
Where Can We Apply Learning?

- In the vulnerability discovery context
  - Concrete Testing
    - Input generation



  - Symbolic Testing
    - Program phase information

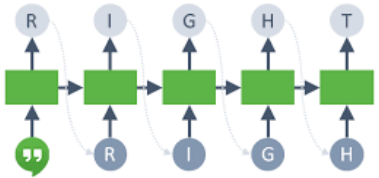
Input Generation Workflow



Platforms

Torch

- https://github.com/torch/distro.git



Char-RNN implementation

- https://github.com/karpathy/char-rnn
- https://github.com/jcjohnson/torch-rnn

Coverage Information

- https://github.com/DynamoRIO/dynamorio

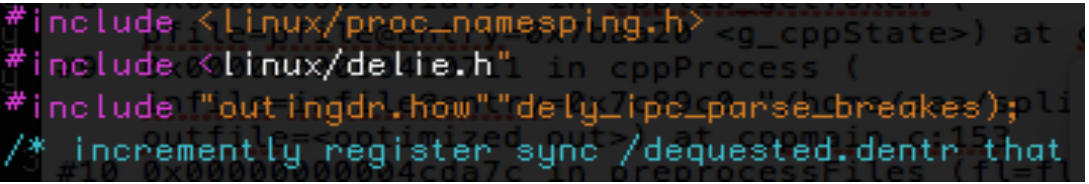
Coverage Comparison

- Dataset
  - Base: 10,000 c files (from Linux kernel)
  - Random: 10,000 randomly generated text files
  - RNN: 10,000 sample files generated from RNN

	Basic block covered (unique blocks not covered by regular C files)	crashes	Unique bugs
C files from Linux kernel	24865 (0)	0	0
Random generated Files	8366 (107)	0	0
RNN generated pseudo C Files	7846 (792)	8	4

Crash Examples

Generated C File, 654.c




```
$ splint /tmp/654.c

… …

/tmp/654.c:3:1: Missing '>' in "#include <FILENAME>"
/tmp/654.c:18:1: Junk at end of #include
/tmp/654.c:18:1: Cannot find include file or chara on search path:

*** Segmentation Violation
*** Location (not trusted): /tmp/654.c:2:1
… …

*** Please report bug to submit@bugs.debian.org (via reportbug)
*** A useful bug report should include everything we need to reproduce the bug.
… …
```

Gdb backtrace

```
Program received signal SIGSEGV, Segmentation fault.
… …
#0  strlen () at ../sysdeps/x86_64/strlen.S:106
#1  0x00000000004a6445 in ?? ()
#2  0x00000000004763c8 in ?? ()
#3  0x0000000000476539 in ?? ()
#4  0x0000000000412ed0 in ?? ()
#5  0x00000000004130dc in ?? ()
#6  0x0000000000419bac in ?? ()
#7  0x000000000041a7d6 in ?? ()
#8  0x00000000004169e3 in ?? ()
#9  0x000000000040f581 in ?? ()
#10 0x00000000004d5c9b in ?? ()
#11 0x00000000004d6743 in ?? ()
#12 0x00007ffff7a33f45 in __libc_start_main
(main=0x4d5f10, argc=2, argv=0x7fffffffe618,
   init=<optimized out>, fini=<optimized out>,
rtld_fini=<optimized out>, stack_end=0x7fffffffe608)
   at libc-start.c:287
#13 0x0000000000401939 in ?? ()
… …
```

Snippets from Other Crash Examples

Consecutive broken include

```
#include <scmd
#include "likuc.h>
```

*** Segmentation Violation
*** Last code point: llmain.c:659
*** Previous code point: llmain.c:588

Empty include name

```
#include <linux/alloc_processes.h>
#include <
```

*** Segmentation Violation
*** Error in `splint': malloc(): memory corruption

Unknown symbol + function pointer

```
static int destroy_same(unsigned int clock_program_probe)
{
    preport_currenter(cfs_rq, BPF_FREE_PERFINED);

    now = (cpuset_index());

    return node;
}
```

*** Segmentation Violation
*** Last code point: transferChecks.c:4415
*** Previous code point: transferChecks.c:4002

Tension between Learn and Fuzz

AFL found crashes with invalid characters

```
static struct {
        struct#.2xin",;
        boo_*cache_fi^?ggg;i
}
"^@^@hlu4^["atim<80>c".udVg^Y);+
```

```
$ splint /tmp/860_45.c

… …
860_45.c:6:16: Invalid character (ascii: 127), skipping character
  Code cannot be parsed.  For help on parse errors, see splint -help
  parseerrors. (Use -syntax to inhibit warning)
860_45.c:6:17: Invalid character (ascii: -1), skipping character
860_45.c:6:18: Invalid character (ascii: -1), skipping character
860_45.c:6:19: Invalid character (ascii: -1), skipping character
860_45.c:6:21: Invalid character (ascii: -95), skipping character
*** Segmentation Violation
*** Error in `splint': malloc(): memory corruption: 0x0000000001a78290 ***
Aborted
```

- RNN based generation won't generate this input
- Training set include only good samples
  ("legit" C programs)

Lesson Learned

- Benefit of Learning based input generation
  - RNN can generate “seemingly syntax correct” input
  - Error often occur when input does not strictly follow specification.
  - Tension between Learn and Fuzz

- Limitation of Learning based input generation
  - Neural Network can learn but can’t generate bytes that were not in the training set, whereas mutation based fuzzing can.

Summary

Two Exercises of Using Learning in Vulnerability Discovery

State Scheduling in Symbolic Execution

1. Many program processes data in phases
2. Symbolic Exec often traps in early phases
3. Learning from concrete execution to infer phases information, thus increases code coverage

Input Generation

1. Ability of mimic simple syntax using RNN
2. Perfect learning (thus perfect syntax) is not best for fuzzing

Questions

kangli.ctf@gmail.com