CE 528 Cloud Computing

Lecture 18: Testing and Verification
Spring 2026

Prof. Yigong Hu

Slides courtesy of Chang Lou and Robert Morrios

Administrivia

- The presentation schedule and peer evaluation form have been posted on Piazza

- Please upload your slides by **12:00 PM** on the day of your presentation

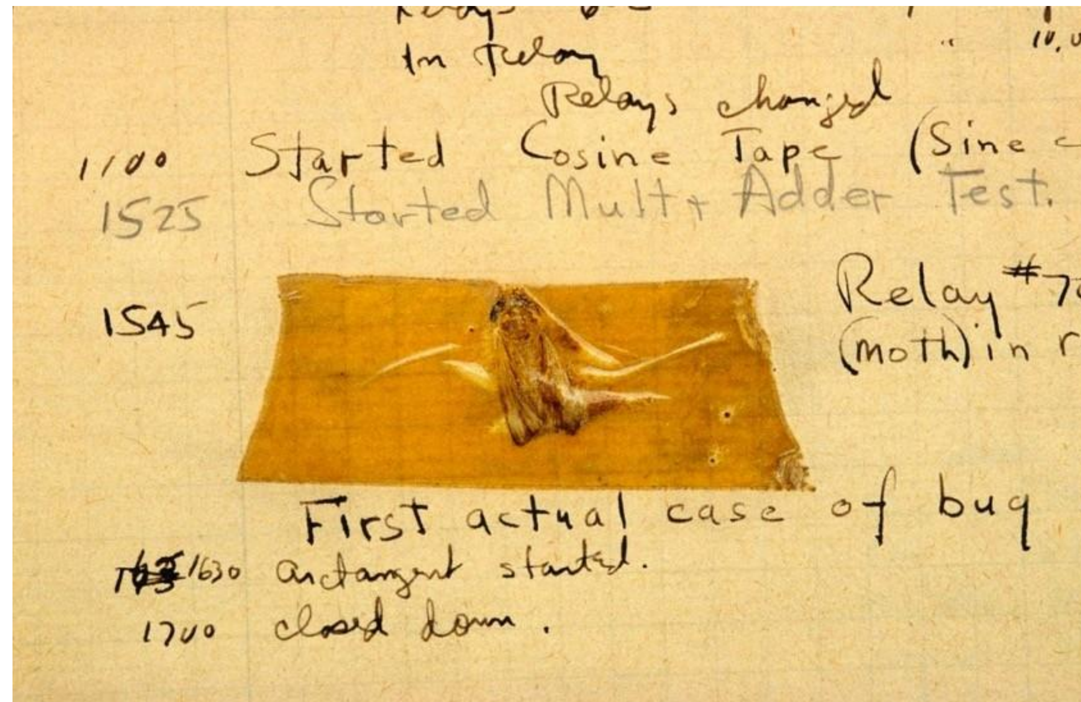
- No quiz question is required

- The course evaluation is now available

Bug

The origin of “bug” is literally a bug

- Coined by U.S. Navy Admiral and computer science pioneer, Grace Hopper
- A moth got into a mechanical relay of Mark II supercomputer, jamming the system.



How to Find a Bug

Testing

- Generate random input, run and see if it triggers a bug

Testing

```
func TestPersist12C(t *testing.T) {
	servers := 3
	cfg := make_config(t, servers, unreliable: false, snapshot: false)
	defer cfg.cleanup()

	cfg.begin( description: "Test (2C): basic persistence")

	cfg.one( cmd: 11, servers, retry: true)

	// crash and re-start all
	for i := 0; i < servers; i++ {
		cfg.start1(i, cfg.applier)
	}
	for i := 0; i < servers; i++ {
		cfg.disconnect(i)
		cfg.connect(i)
	}

	cfg.one( cmd: 12, servers, retry: true)

	leader1 := cfg.checkOneLeader()
	cfg.disconnect(leader1)
	cfg.start1(leader1, cfg.applier)
	cfg.connect(leader1)

	cfg.one( cmd: 13, servers, retry: true)
```

Testing

```go
func TestPersist12C(t *testing.T) {
    servers := 3
    cfg := make_config(t, servers, unreliable: false, snapshot: false)
    defer cfg.cleanup()

    cfg.begin( description: "Test (2C): basic persistence")

    cfg.one( cmd: 11, servers, retry: true)

    // crash and re-start all
    for i := 0; i < servers; i++ {
        cfg.start1(i, cfg.applier)
    }
    for i := 0; i < servers; i++ {
        cfg.disconnect(i)
        cfg.connect(i)
    }

    cfg.one( cmd: 12, servers, retry: true)
```

set input

one(11)
disconnect(1)
connect(1)
disconnect(2)..

Execute program

Check result

Testing

```go
func TestPersist12C(t *testing.T) {
    servers := 3
    cfg := make_config(t, servers, unreliable: false, snapshot: false)
    defer cfg.cleanup()

    cfg.begin( description: "Test (2C): basic persistence")

    cfg.one( cmd: 11, servers, retry: true)

    // crash and re-start all
    for i := 0; i < servers; i++ {
        cfg.start1(i, cfg.applier)
    }
    for i := 0; i < servers; i++ {
        cfg.disconnect(i)
        cfg.connect(i)
    }

    cfg.one( cmd: 12, servers, retry: true)
```

set input

one(11)
disconnect(1)
connect(1)
disconnect(2)..

Execute program

Check result

Test passed, does that mean your program has no bug?

tests only cover a small portion of possibilities!

Fuzz Testing

Goal:

- To find program inputs that reveal a bug

Approach:

- Generate inputs randomly until program reports errors

Fuzzing Testing Example

Standard HTTP GET request

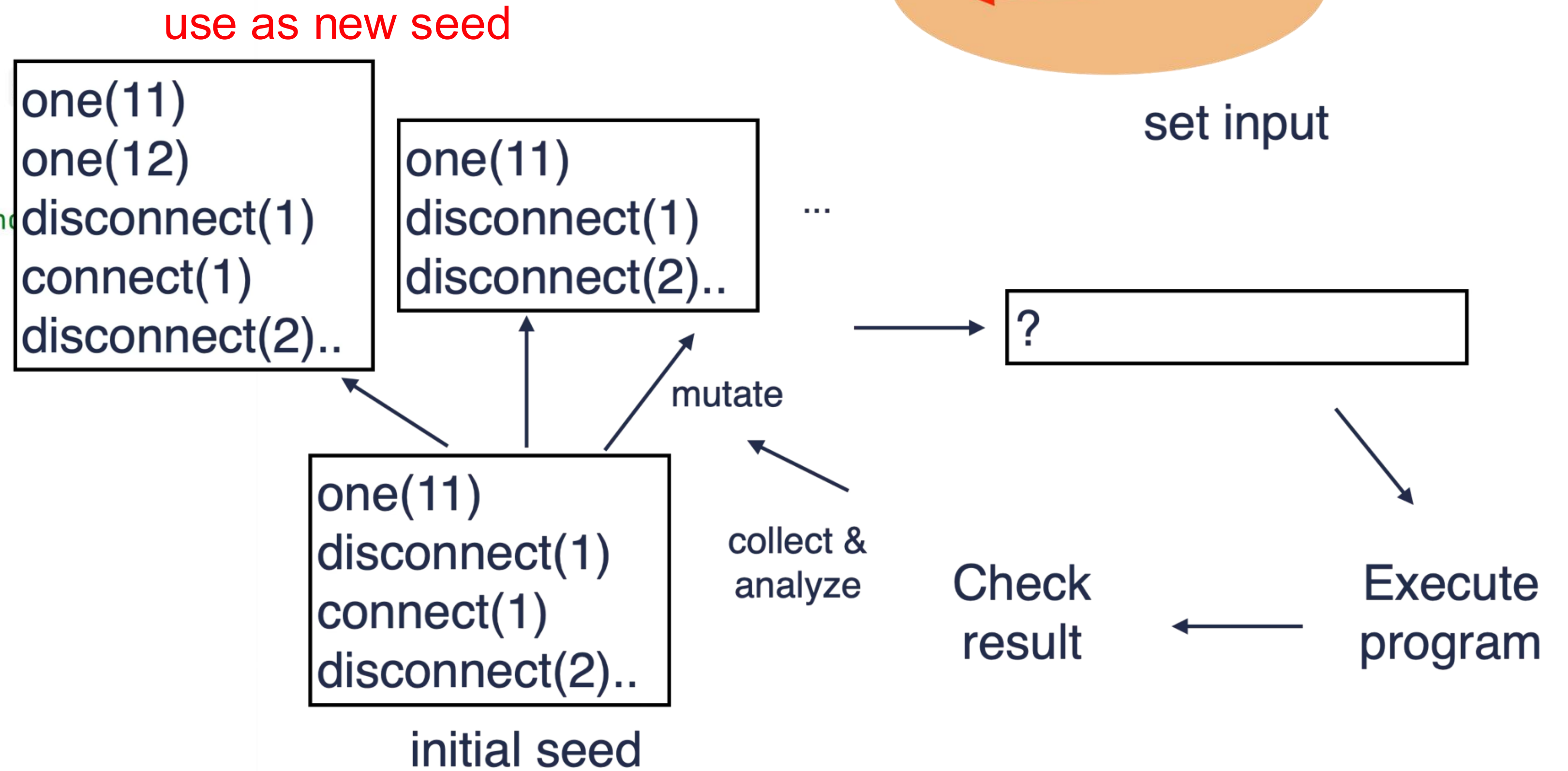
- § GET /index.html HTTP/1.1

Fuzzing HTTP GET request

- § AAAAAA...AAAA /index.html HTTP/1.1
- § GET ///////index.html HTTP/1.1
- § GET %n%n%n%n%n%n.html HTTP/1.1
- § GET /AAAAAAAAAAAAA.html HTTP/1.1
- § GET /index.html HTTTTTTTTTTTTTP/1.1

Fuzz Testing

```
func TestPersist12C(t *testing.T) {
    servers := 3
    cfg := make_config(t, servers, unreliable: false,
    defer cfg.cleanup()

    cfg.begin( description: "Test (2C): basic persisten

    cfg.one( cmd: 11, servers,  retry: true)

    // crash and re-start all
    for i := 0; i < servers; i++ {
        cfg.start1(i, cfg.applier)
    }
    for i := 0; i < servers; i++ {
        cfg.disconnect(i)
        cfg.connect(i)
    }

    cfg.one( cmd: 12, servers,  retry: true)
```



How to Find a Bug

Testing

- Generate random input, run and see if it triggers a bug

Static Analysis

- Analyze all possible behavior of a program without running it

What is a Static Analysis

A static analysis tool S analyzes the source code of a program P to determine whether it satisfies a property φ, such as:

- “P never deference a null pointer”
- “P does not leak file handles”
- “P does not divide by Zero”

Bad News

It is impossible to write such a tool!

- Why?
- **Rice Theorem:** Any nontrivial semantic property φ is undecidable, meaning it is impossible to construct a general automated method to determine whether P satisfies φ

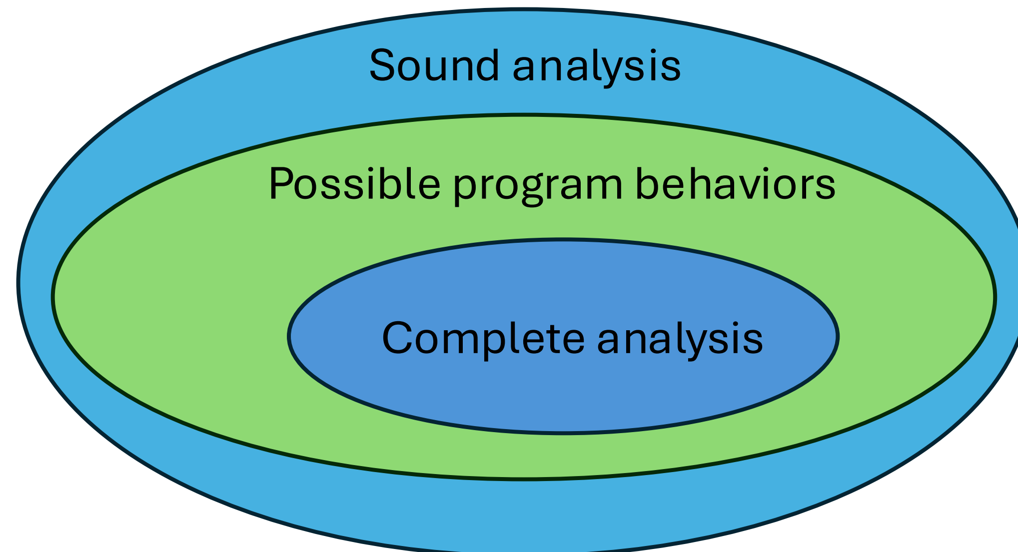
How to design a practical static analysis?

- Trade-off between soundness and completeness

Soundness VS Completeness

A practical static analysis tool analyzes whether a program satisfies a property φ, but can be wrong in one of two way:

- A sound static analysis never miss any violations but may **report false positives**
- A complete static analysis never report false positives, but it does not guarantee all violations will be reported (**false negative**)

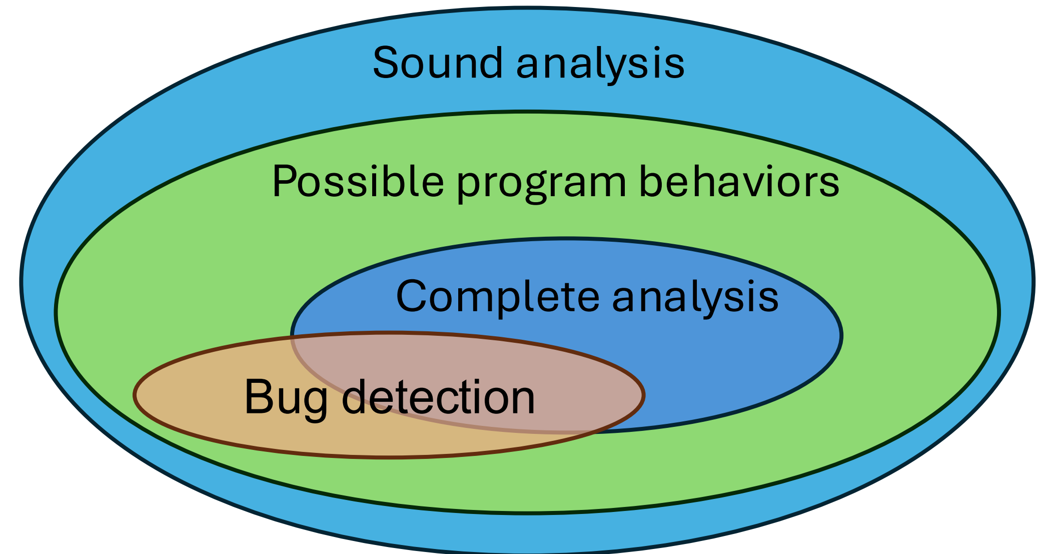


Applications of Static Analysis

Compiler (sound)

- Type checking, liveness analysis …

Verification (sound)



Bug finding?

- Usually complete
- But many bug detection tool for are neither sound nor complete

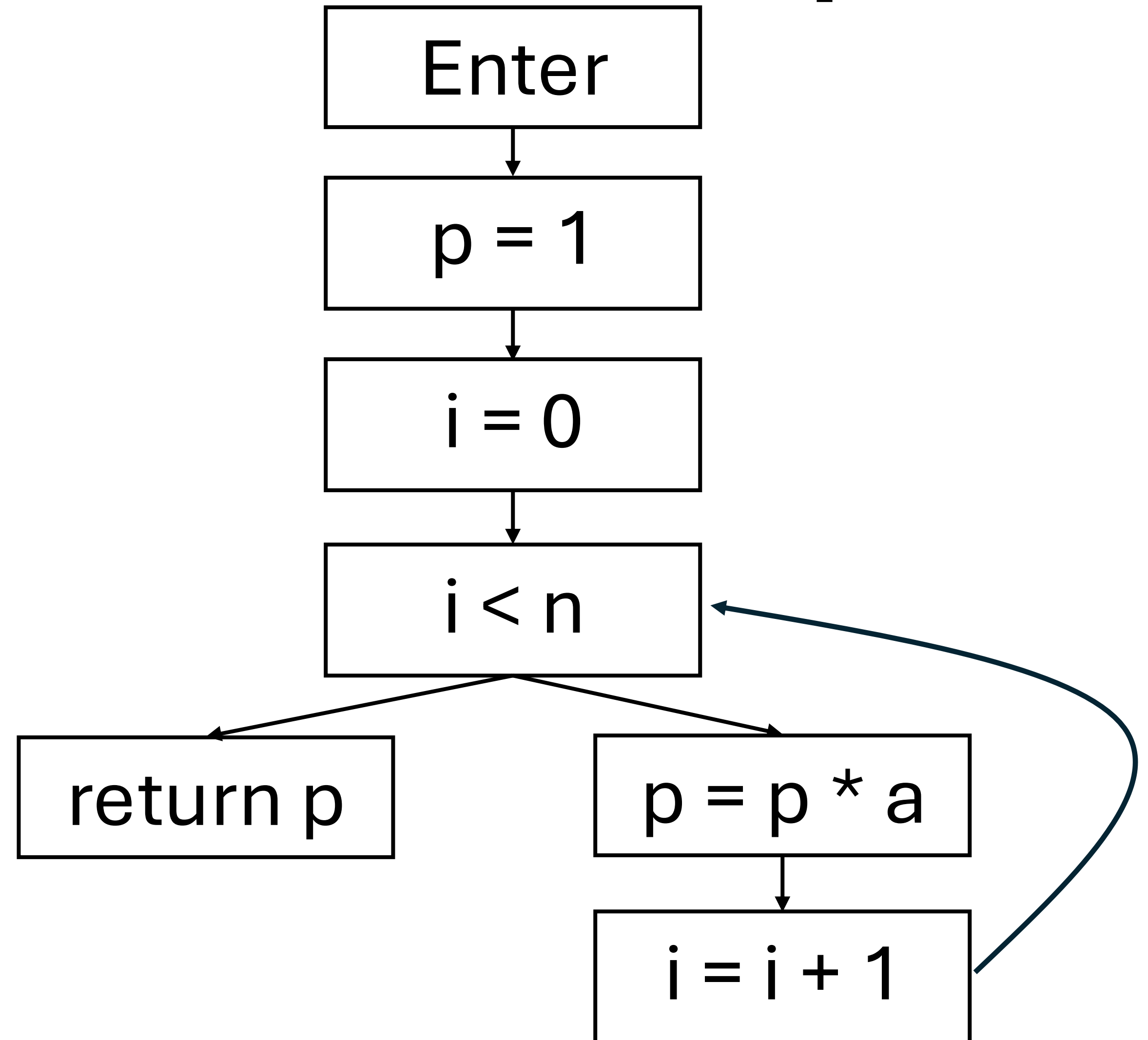
A Toy Static Analysis Example

```
int foo (int a, int n) {
    int p = 1;
    for (int i = 0; i < n; i++)
        p *= a;
    return p;
}
```

p = 1
i = 0
i < n
i = i + 1
p = p * a
return p

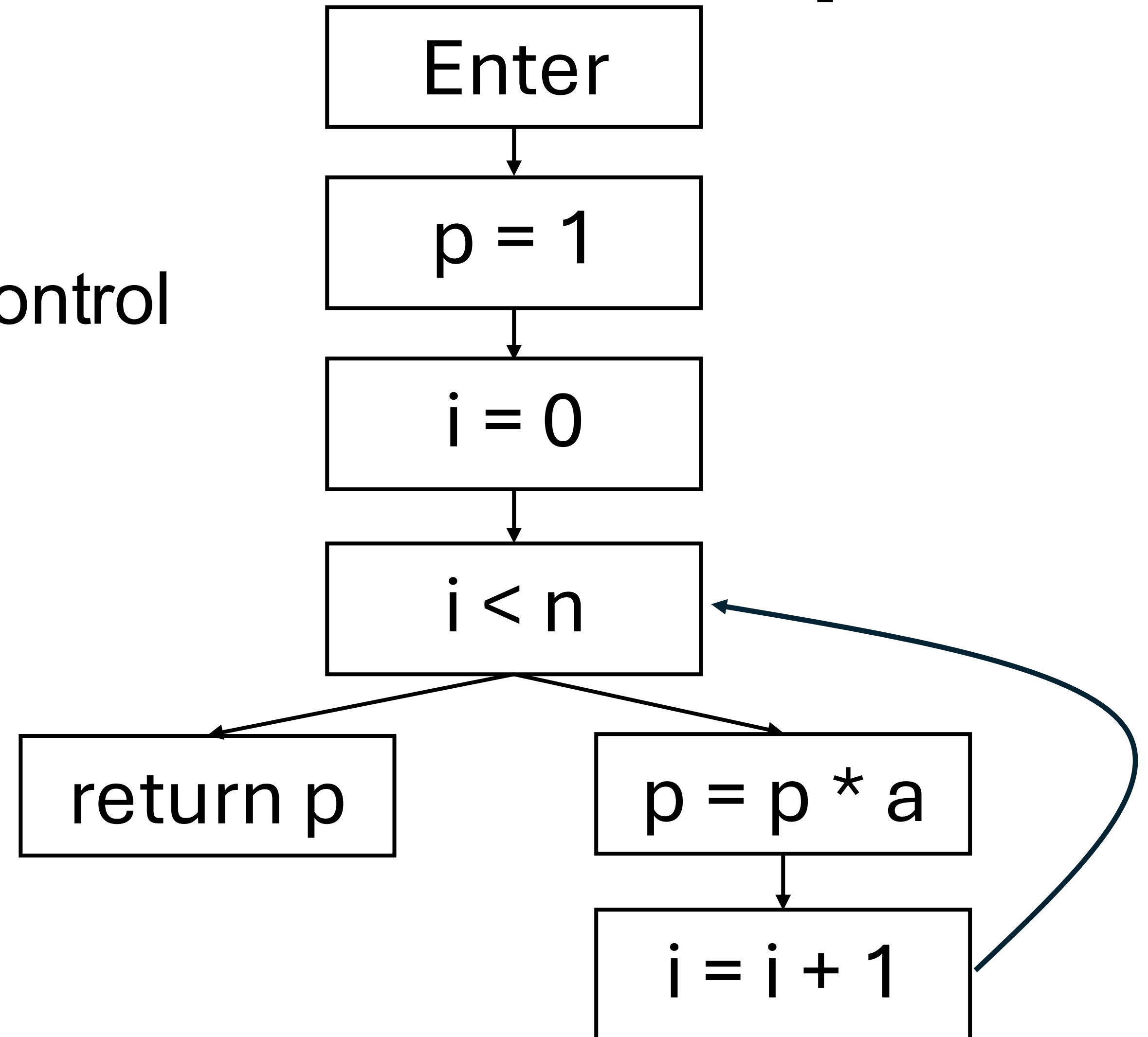
A Toy Example: Control-Flow Graph

```
int foo (int a, int n) {
    int p = 1;
    for (int i = 0; i < n; i++)
        p *= a;
    return p;
}
```



A Toy Example: Control-Flow Graph

- **Directed graph**
  - Each node is a statement
  - Each edges represent possible control flow
- **Statement may be**
  - Assignments
  - Branch
  - Enter/return
  - …



How to Find a Bug

Testing

- Generate random input, run and see if it triggers a bug

Static Analysis

- Analyze all possible behavior of a program without running it

Symbolic Execution

- “KLEE Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler

Why Not Static Analysis

Sound static analysis is great

- Can prove many important errors (leak, divided by zero)

But their false positive can be difficult to eliminate

- Static analysis can produce many false positives on large or unusual code bases
- Unless you are an expert, telling a false positive from a real bug can be hard

Symbolic Execution

Goal: a bug finding technique that is easy to use!

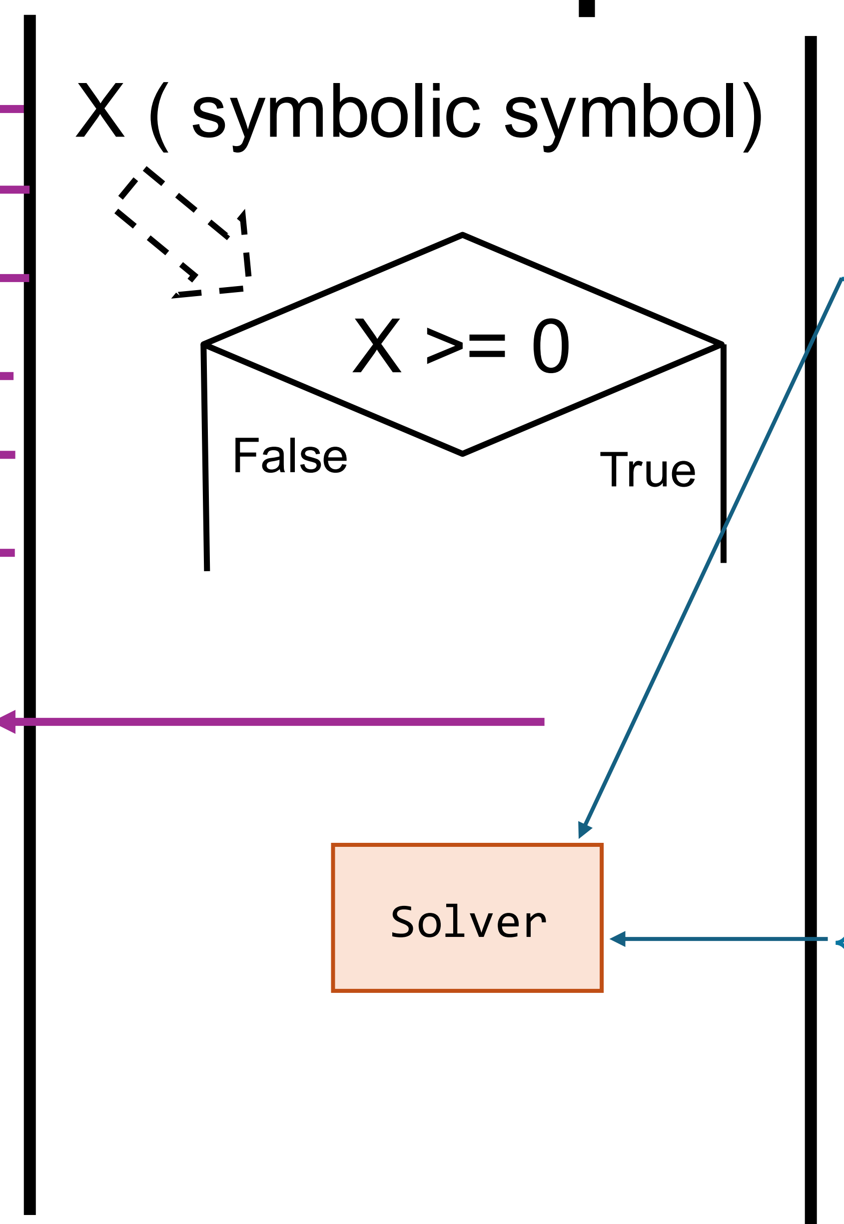
- No false positives
- Produces a concrete input on which the program will fail to meet the specification
- But can not prove the absence of errors

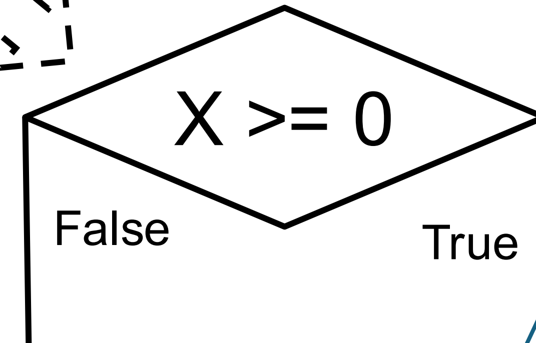
Key insight of Symbolic Execution

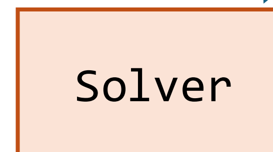
- Evaluate the program on symbolic input values
- Use an automated theorem prover to check whether there are corresponding concrete input values that make the program fail.

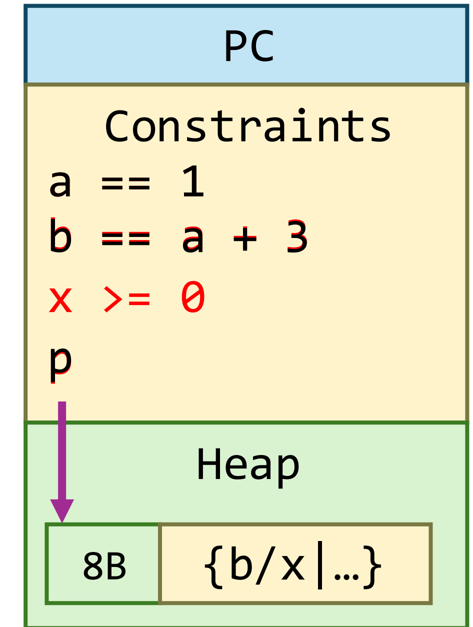
Symbolic Execution - an Example

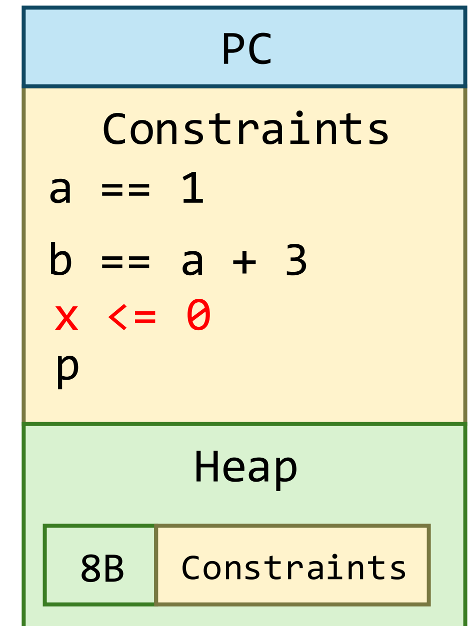
```
int foo (int x) {
    int a = 1;
    int b = a + 3;
    int *p = malloc(sizeof(int));
    if (x >= 0) {
        p *= b / x;
} else {
    FILE *fp = fopen(“a.txt”, “w”)
    fputc ('?’,fp);
}
```











What is KLEE

Extensible symbolic execution platform, used and extended by many groups in academia and industry

- Bug finding
- Test input generation
- Automated debugging
- …

304 papers used KLEE(link)

Why is KLEE so successful?

Easy to Use

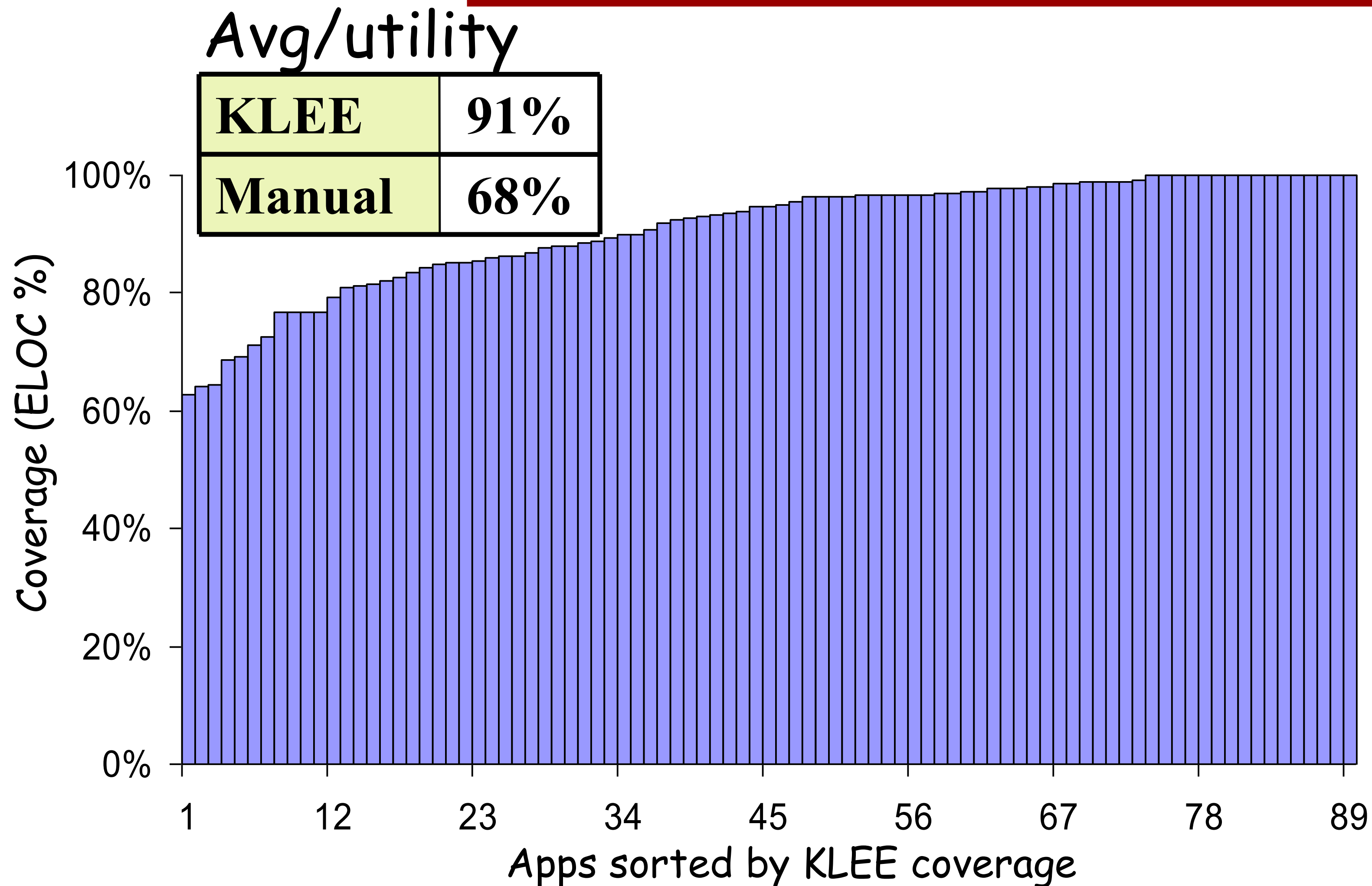
Demo

```
// #include directives
struct image_t {
  unsigned short magic;
  unsigned short h, sz; // height, size
  char pixels[1018];
};
int main(int argc, char** argv) {
  struct image_t img;
  int fd = open(argv[1], O_RDONLY);
  read(fd, &img, 1024);

  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  unsigned short w = img.sz / img.h;

  return w;
}
```

```
$ clang -emit-llvm -c -g image_viewer.c
$ klee --posix-runtime -write-pcs
  image_viewer.bc --sym-files 1 1024 A
...
KLEE: output directory = klee-out-1
(klee-last)
...
KLEE: ERROR: ... divide by zero
...
KLEE: done: generated tests = 4
```

High Code Coverage

(Coreutils, non-lib, 1h/utility = 89 h)



[Cadar, Dunbar, Engler OSDI 2008]

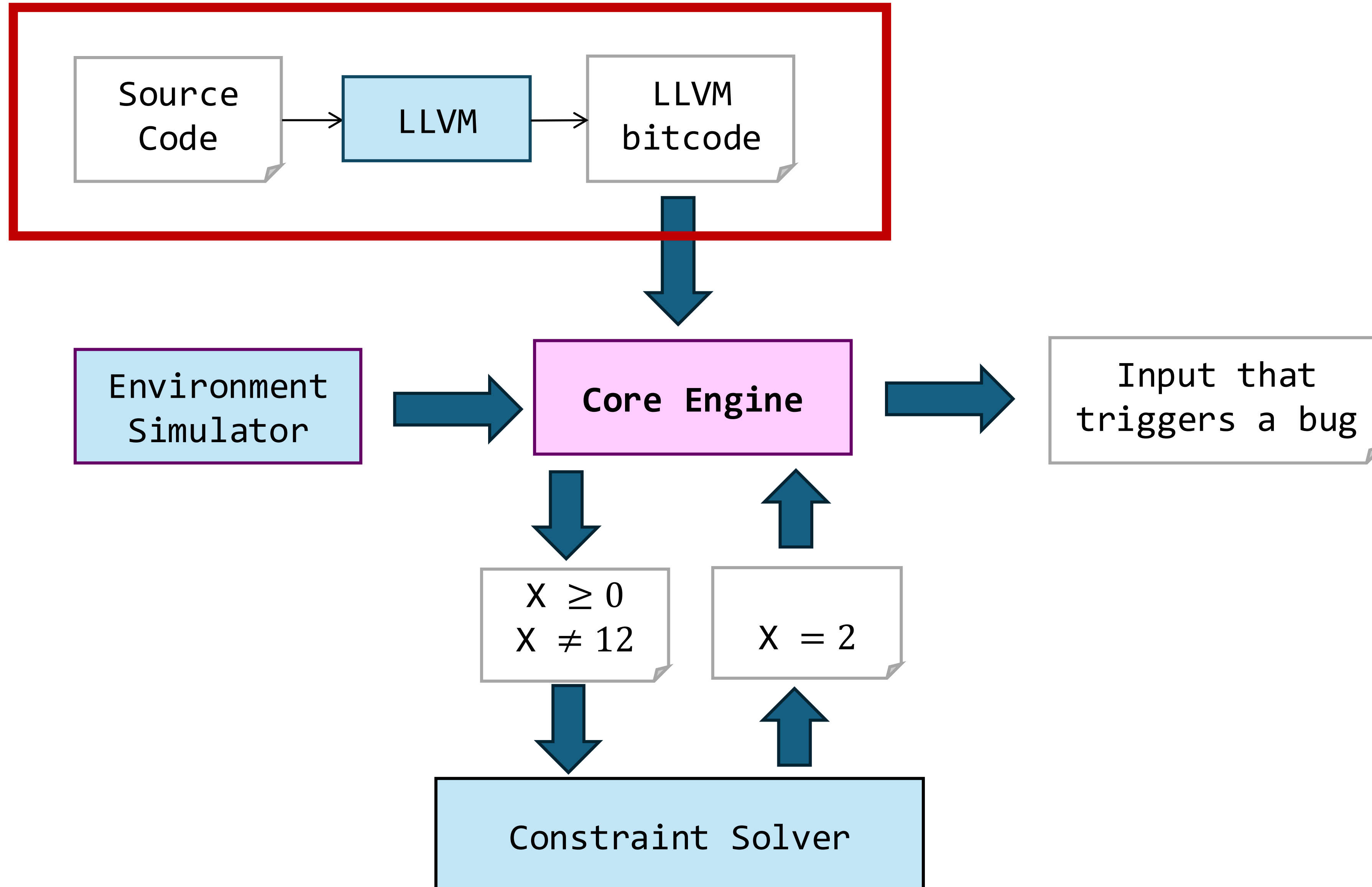
Bug Finding with KLEE

Focus on Systems and Security Critical Code

	Applications
UNIX utilities	Coreutils, Busybox, Minix (over 450 apps)
UNIX file systems	ext2, ext3, JFS
Network servers	Bonjour, Avahi, udhcpd, lighttpd, etc.
Library code	libdwarf, libelf, PCRE, uClibc, etc.
Packet filters	FreeBSD BPF, Linux BPF
MINIX device drivers	pci, lance, sb16
Kernel code	HiStar kernel
Computer vision code	OpenCV (filter, remap, resize, etc.)
OpenCL code	Parboil, Bullet, OP2

- Most bugs fixed promptly

KLEE Framework



KLEE's Input : Bitcode

Why LLVM?

- Mature framework – incorporated into commercial products
- Easy to design = analysis passes design + Useful program analysis
- Extensible: different front-end

KLEE runs LLVM, not C code

```
klee@5c7f9e1945aa:~$ clang -emit-llvm -c -g image_viewer.c
klee@5c7f9e1945aa:~$ ls
image_viewer.bc  image_viewer.c  klee_build  klee_src
klee@5c7f9e1945aa:~$ klee --posix-runtime image_viewer.bc --sym-files 1 1024 A
KLEE: NOTE: Using POSIX model: /tmp/klee_build130stp_z3/runtime/lib/libkleeRuntimePOSIX64_Debug+Asserts.bca
```

Limitation of LLVM

Fast changing, not-backward compatible API

- KLEE – clang 13.0 (latest LLVM 19.0)

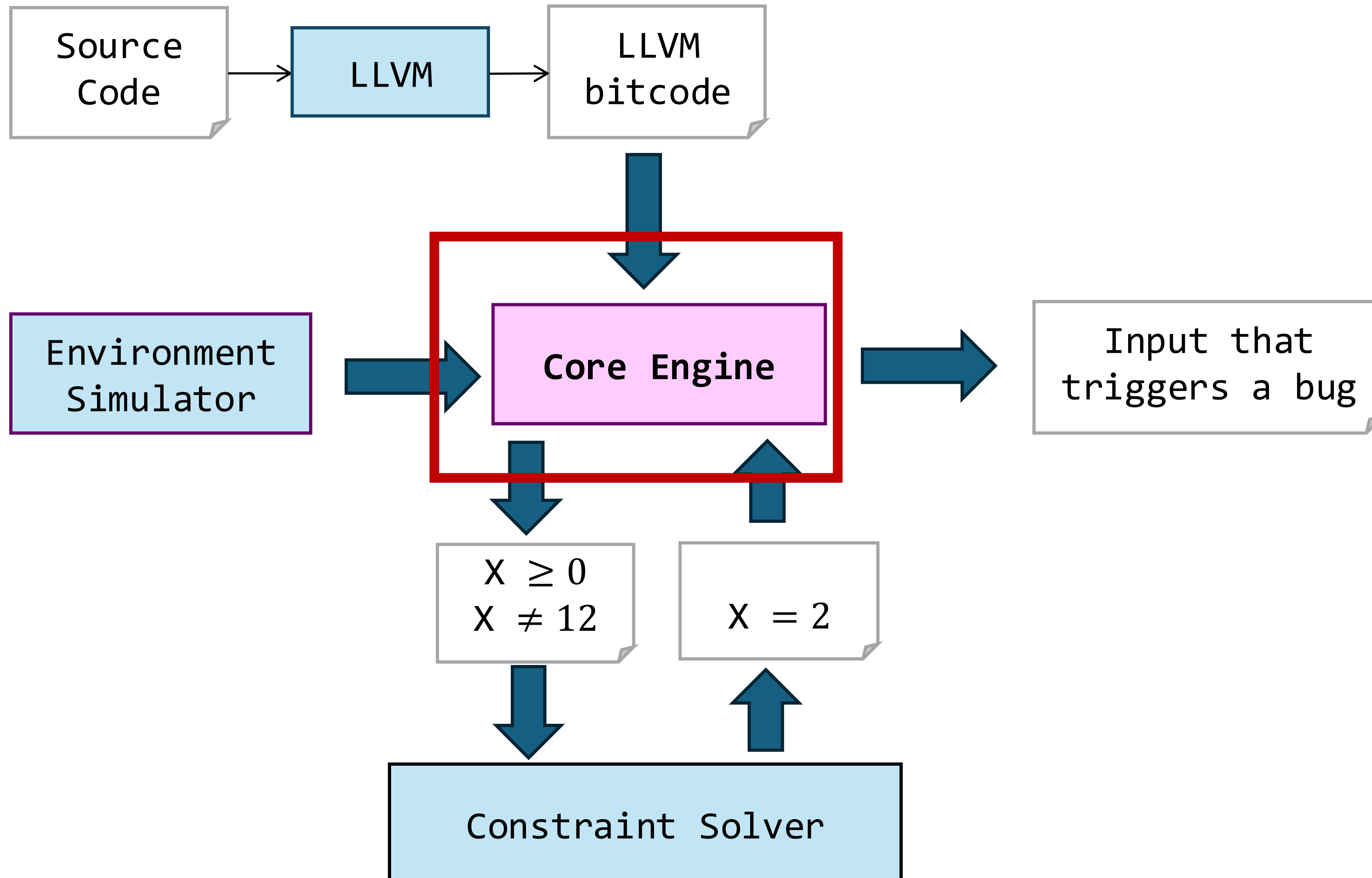
Compiling a software to LLVM bitcode is hard

- WLLVM

Executing LLVM bitcode is slow

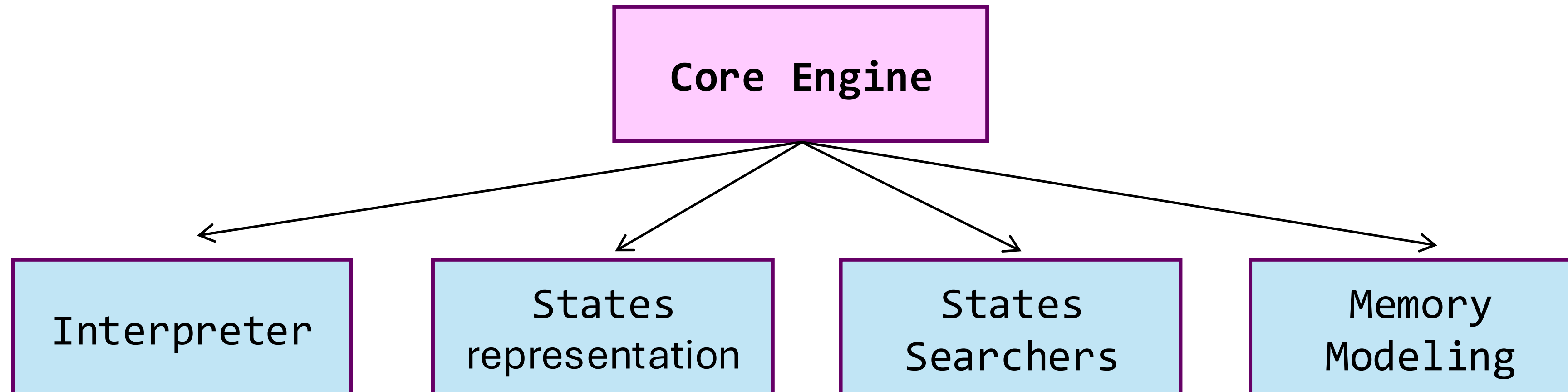
- Starting MySQL: 1 minutes in native machine; > 40 minutes in KLEE
- Limited to KLEE capability to large scale software

KLEE Framework



Core Engine

Core engine implements symbolic execution exploration



Interpreter

A mixed concrete/symbolic interpreter for LLVM bitcode

- Be concrete whenever possible

```
instruction *i = ki->inst;
switch ( i->getOpcode()) {
     case Instruction::Ret:
     …
    case Instruction::Br:
    // if both sides feasible, fork
   …
}
```

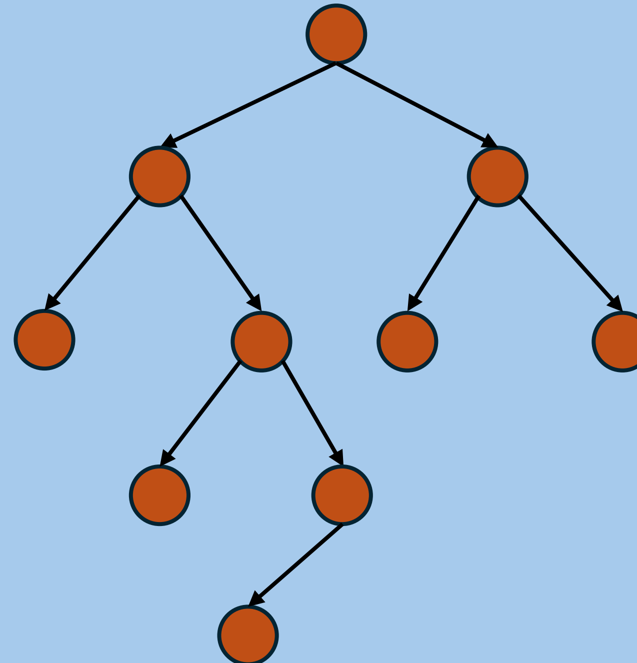
State Representation

- **Each path represented by an ExecutionState, with KLLE acting as an OS for ExecutionStates**

ExecutionState

- PC
- Stack
- Address space
- List of sym objects
- Path constraints
- Etc.

Tree of ESs



Two Key Challenges

Path exploration challenges

- Naïve exploration can easily get “stuck”
- Employing search heuristics
- Dynamically eliminating redundant paths
- Statically merging paths
- Skipping irrelevant code

Constraint solving challenges

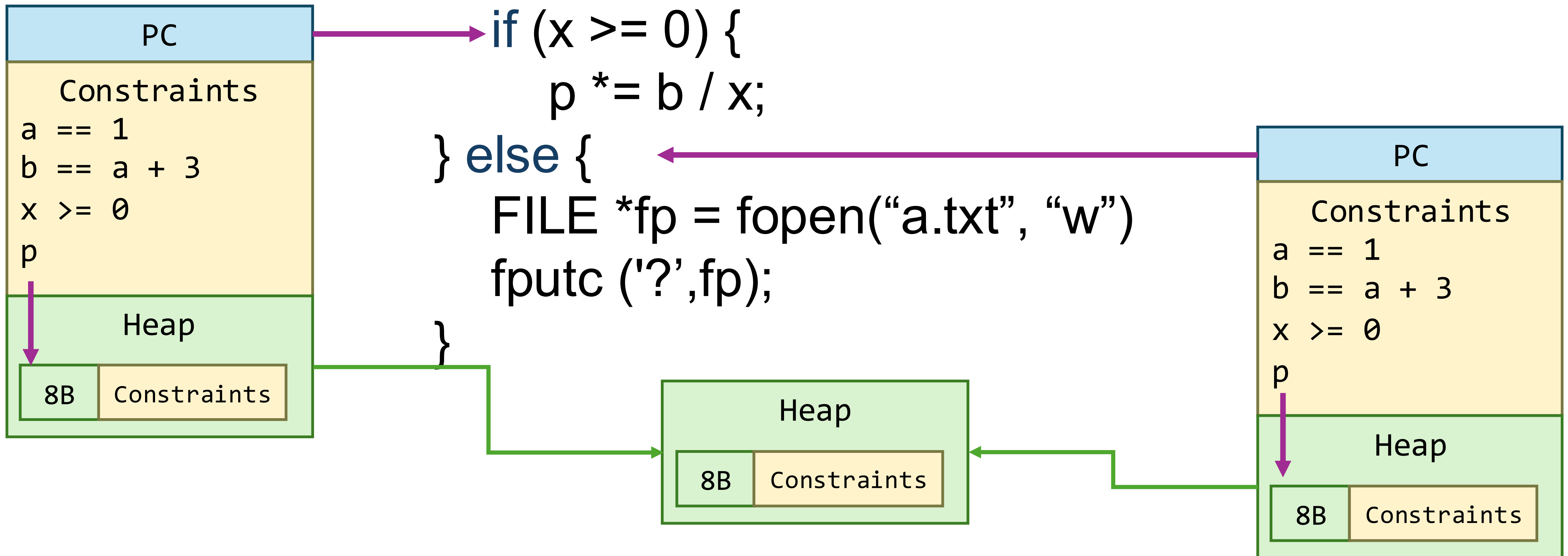
Addressing Path Exploration: States Search

Search Heuristics in KLEE

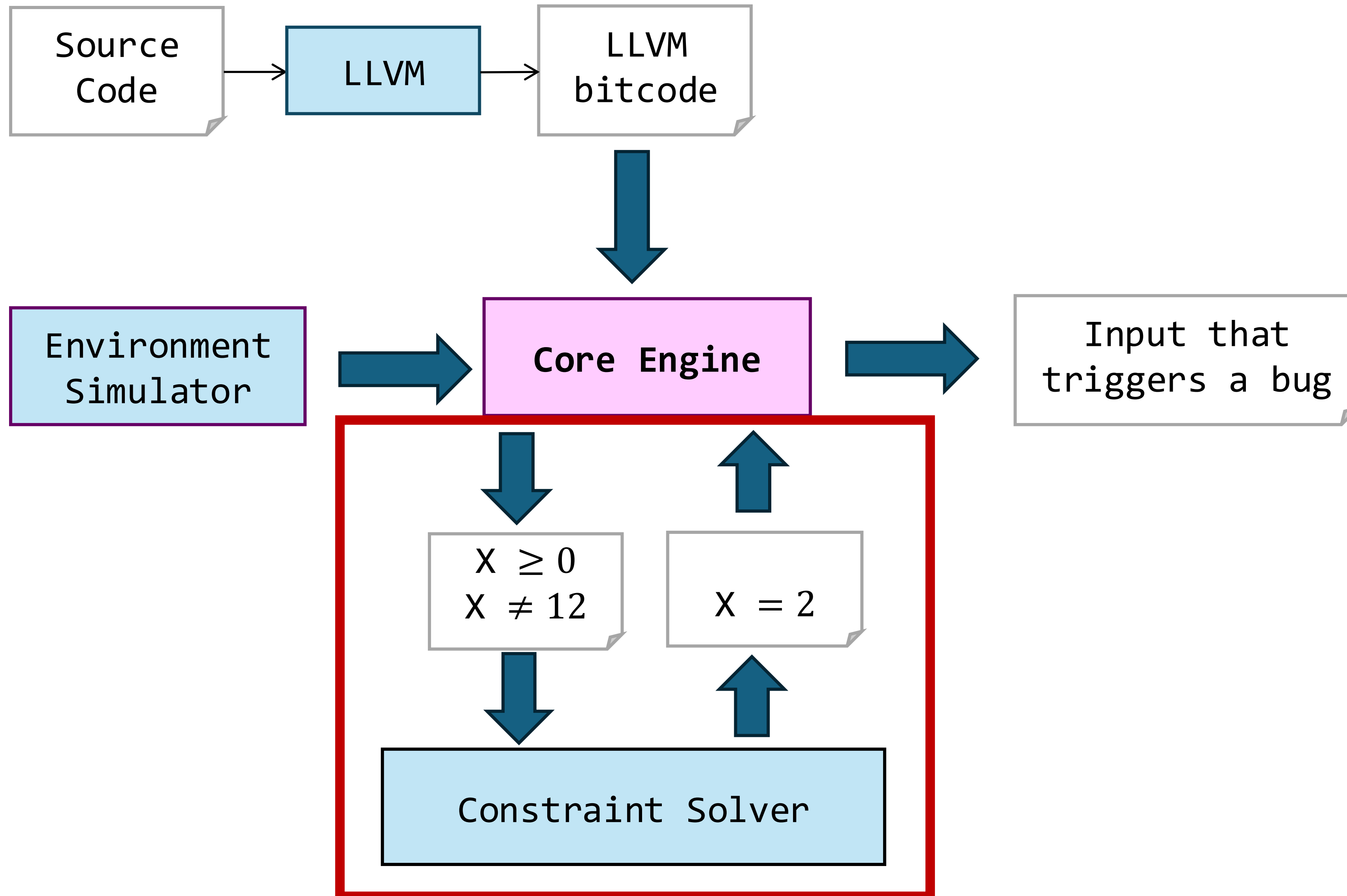
- BFS/DFS
- Coverage-optimized search
- Random-state search
- Random path search

Memory Sharing

```
int foo (int x) {
    int a = 1;
    int b = a + 3;
    int *p = malloc( sizeof(int) );
    if (x >= 0) {
        p *= b / x;
} else {
    FILE *fp = fopen("a.txt", "w")
    fputc ('?',fp);
}
```



KLEE Framework



Constrains Solving: Performance

Constraint solving challenges

- Inherently expensive
- Invoked at every branch

Key insight: exploit the characteristics of constraints generated by symex

Query Optimization: Solver : I get stressed out, you get timeout

- **Two Simple and effective optimizations**
  - Eliminating irrelevant constraints

Eliminating Irrelevant Constraints

- **In practice, each branch usually depends on a small number of variables**

```
…
…
  if (x < 10) {
…
}
```

~~w + z > 100~~

~~2 * w – 1 < 123~~

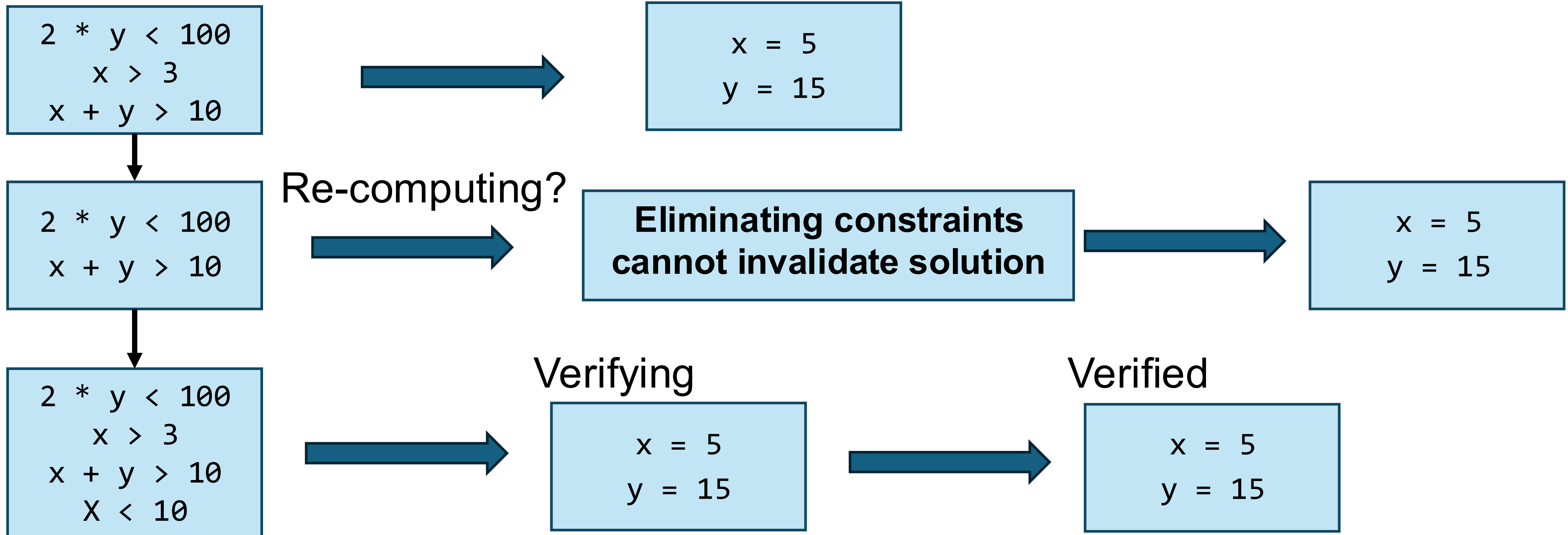
x + y > 10

~~z < 10~~

Question: x < 10?

Cache Solutions

- **Lots of similar constraint sets**



Speedup