CE 874 - Secure Software Systems

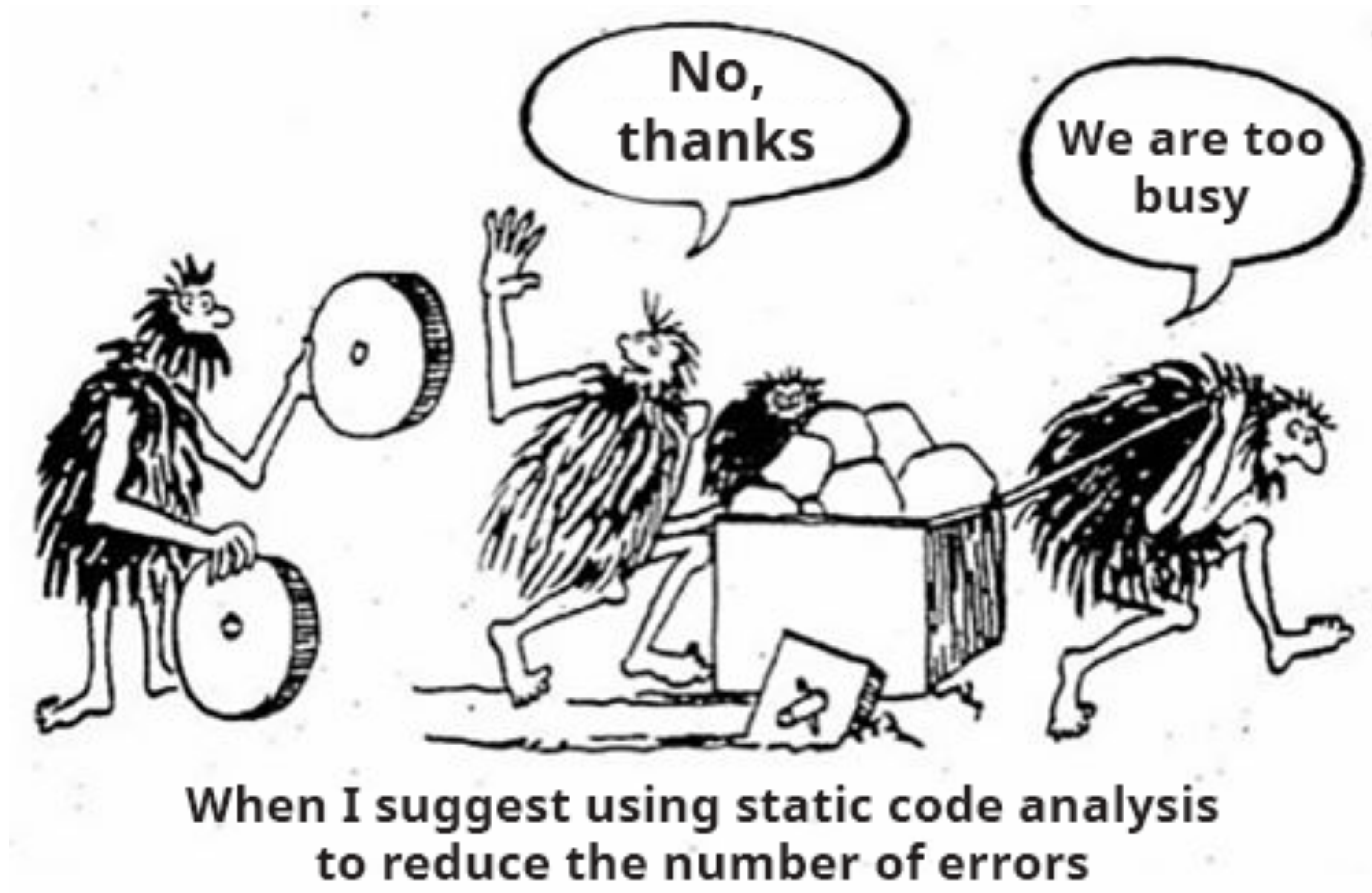
Program Analysis

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

Program Analysis

- How could we analyze a program (with source code) and look for problems?
- How accurate would our analysis be without executing the code?
- If we execute the code, what input values should we use to test/analyze the code?



https://www.viva64.com

What is Program Analysis?

- Body of work to discover useful facts about programs

- Broadly classified into three kinds:
  - Dynamic (execution-time)
  - Static (compile-time)
  - Hybrid (combines dynamic and static)

Dynamic Program Analysis

- Infer facts of program by monitoring its runs
- Examples:

Array bound checking *Purify*	Datarace detection *Eraser*
Memory leak detection *Valgrind*	Finding likely invariants *Daikon*

Static Analysis

- Infer facts of the program by inspecting its source (or binary) code

- Examples:

Suspicious error patterns *Lint, FindBugs, Coverity*	Memory leak detection *Facebook Infer*
Checking API usage rules *Microsoft SLAM*	Verifying invariants *ESC/Java*

Dynamic vs. Static Analysis

	Dynamic	Static
Cost		
Effectiveness		

A. Unsound (may miss errors)

B. Proportional to program's execution time

C. Proportional to program's size

D. Incomplete (may report false positives)

QUIZ: Dynamic vs. Static Analysis

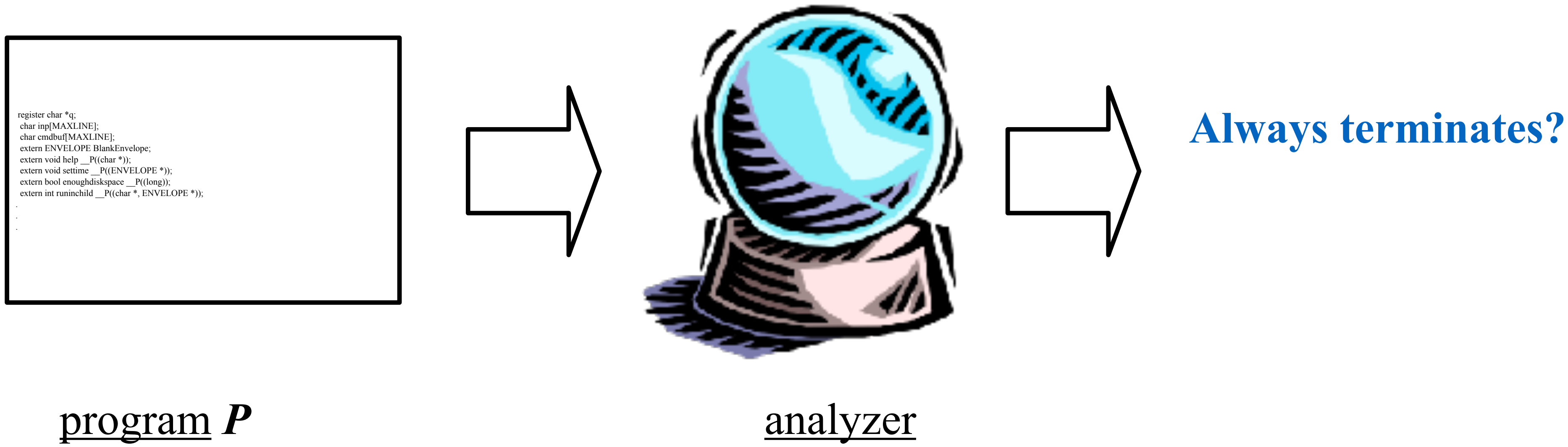
	Dynamic	Static
Cost	B. Proportional to program's execution time	C. Proportional to program's size
Effectiveness	A. Unsound (may miss errors)	D. Incomplete (may report false positives)

Static Analysis

Static analysis

- Analyze program's code without running it
  - In a sense, ask a computer to do code review
- Benefit: (much) higher coverage
    - Reason about many possible runs of the program
      - Sometimes all of them, providing a guarantee
    - Reason about incomplete programs (e.g., libraries)
- Drawbacks:
  - Can only analyze limited properties
  - May miss some errors, or have false alarms
  - Can be time- and resource-consuming

The Halting Problem



- Can we write an analyzer that can prove, for any program P and inputs to it, P will terminate?
    - Doing so is called the halting problem
    - Unfortunately, this is undecidable: any analyzer will fail to produce an answer for at least some programs and/or inputs

So is static analysis impossible?

- Perfect static analysis is not possible
- Useful static analysis is perfectly possible, despite
  - Nontermination - analyzer never terminates, or
  - False alarms - claimed errors are not really errors, or
  - Missed errors - no error reports ≠ error free
- Nonterminating analyses are confusing, so tools tend to exhibit only false alarms and/or missed errors

Reminder

- Soundness: No error found = no error exists
  - Alarms may be false errors
- Completeness: Any error found = real error
  - Silence does not guarantee no errors
- Basically any useful analysis
  - is neither sound nor complete (def. not both)
  - … usually leans one way or the other

The Art of Static Analysis

- Design goals:
  - Precision: Carefully model program, minimize false positives/negatives
  - Scalability: Successfully analyze large programs
  - Understandability: Error reports should be actionable
- Observation: Code style is important
  - Aim to be precise for “good” programs
    - OK to forbid yucky code in the name of safety
    - Code that is more understandable to the analysis is more understandable to humans

Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions

Dawson Engler, Benjamin Chelf, Andy Chou, Seth Hallem, OSDI 2005

Motivation

- Developers of systems software have “rules” to check for correctness or performance. (Do X, don’t do X, do X before Y…)
- Code that does not obey these “rules” will run slow, crash the system, launch the missiles…
- Consequently, we need a systematic way of finding as many of these bugs as we can, preferably for as little cost as possible.

What’s the Problem?

- Current solutions all have trade-offs.
- Formal Specifications-rigorous, mathematical approach
  - Finds obscure bugs, but is hard to do, expensive, and don’t always mirror the actual written code.
- Testing-systematic approach to test the actual code
  - Will detect bugs, but testing a large system could require exponential/ combinatorial number of test cases. It also doesn’t isolate where the bug is, just that a bug exists.
- Manual Inspection-peer review of the code
  - Peer has knowledge of whole system and semantics, but doesn’t have the diligence of a computer.

What's the Problem?

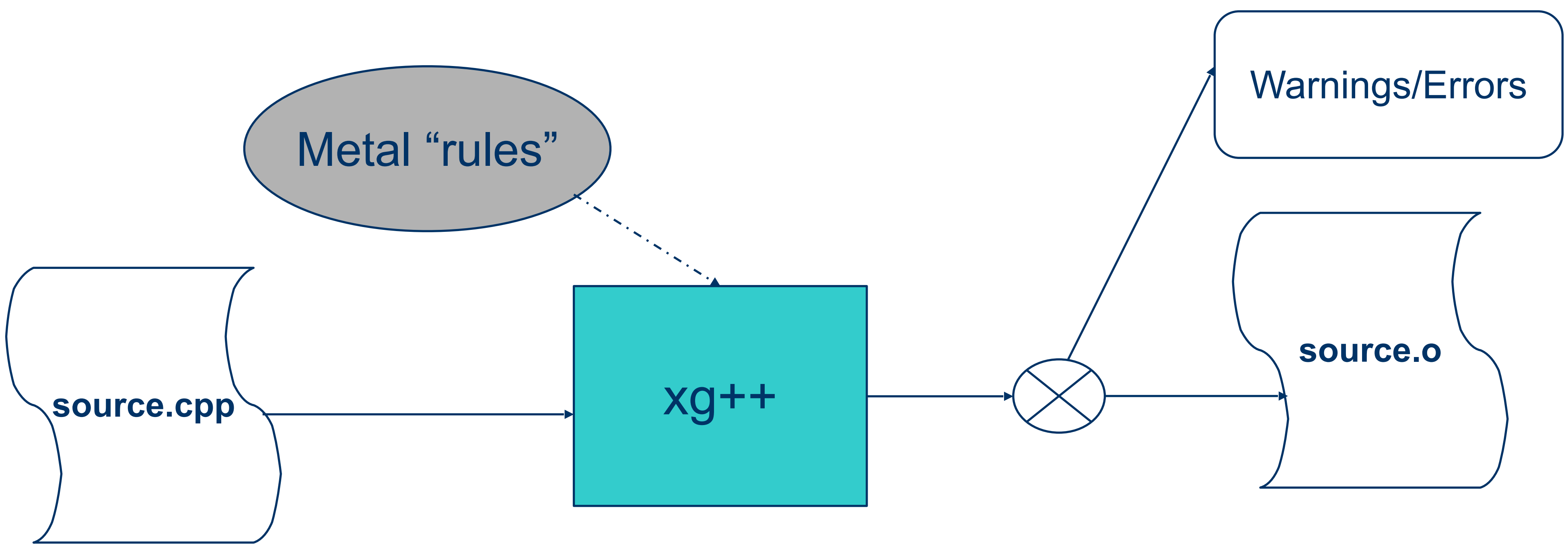
- None of the current methods seem to give us what we're looking for.
- Can the compiler check the code?
  - It would be nice to put the code in the compiler and have it check all of the "rules."
  - Unfortunately, those "rules" are based on semantics of the system that the compiler doesn't understand. (Lock and Unlock are valid to the compiler, but how and when they should be used isn't.)
- Need some technique that merges the domain knowledge of the developer with the analysis of a compiler.

What's the Solution?

- Meta-level compilation (MC) combines the domain knowledge of developers with analysis capabilities of a compiler.
- Allows programmers to write short, simple, system-specific checkers that take into account unique semantics of a system.
- Checkers are then added to a compiler to check during compile-time.

What's the Solution?

- The author's [Engler] MC system uses a high-level, state-machine language called Metal.
- Metal extensions written by programmers are linked to a compiler (xg++) that analyzes the code as it is being compiled.
  - Intra and Interprocedural analysis.

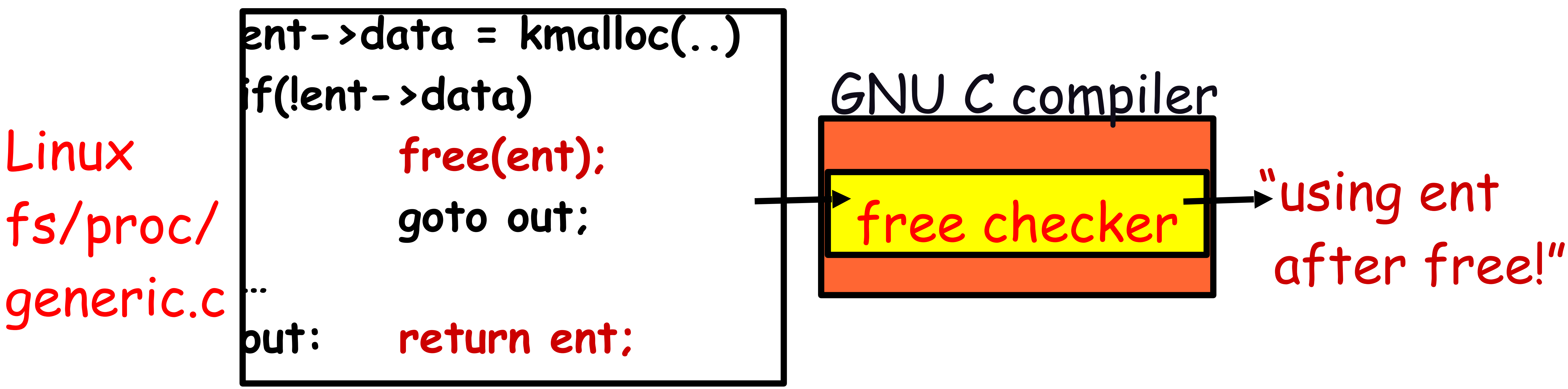


How does it work?

- The language is a high-level, state-machine language.
- Two parts of the language—pattern part and state-transition part.
  - Pattern language—finds “interesting” parts of code based on the extension the programmer writes.
  - State-transition—Based on the discovered pattern, current state, either move to a new state or raise an error.
- Tests are written and then added to the xg++ compiler. Xg++ includes a base library that includes some common, useful functions and types.

Metacompilation (MC)

- Implementation:
  - Extensions dynamically linked into GNU gcc compiler
  - Applied down all paths in input program source



  - Scalable: handles millions of lines of code
  - Precise: says exactly what error was
  - Immediate: finds bugs without having to execute path
  - Effective: 1500+ errors in Linux source code

Bugs to Detect

Some examples

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code
- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

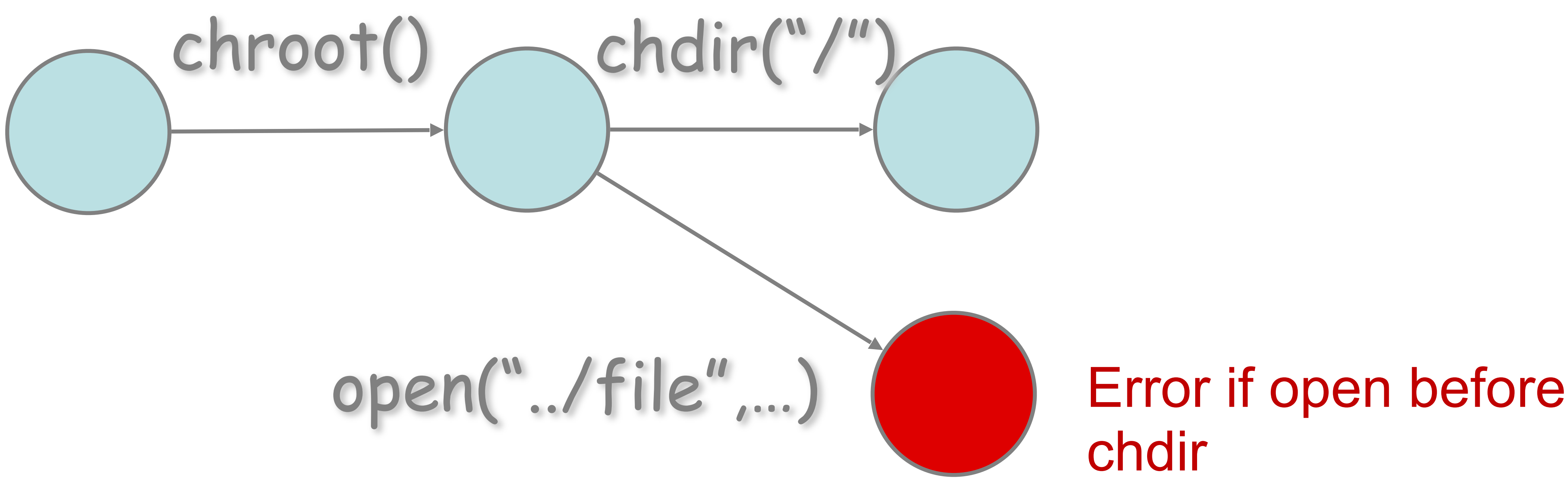
Slide credit: Andy Chou

Example: Check for missing optional args

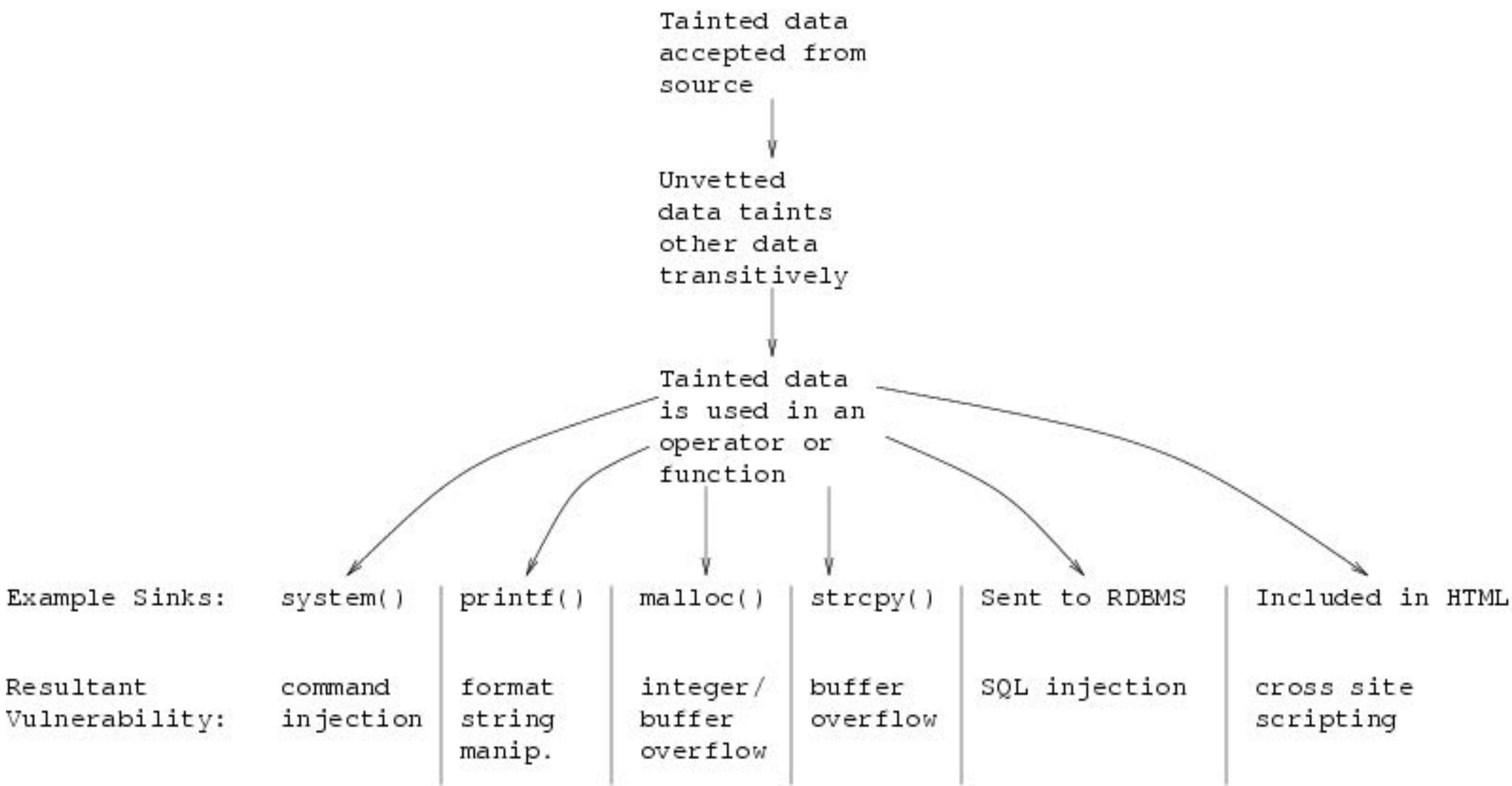
- Prototype for open() syscall:
  - int open(const char *path, int oflag, /* mode_t mode */...);
- Typical mistake:
  - fd = open(“file”, O_CREAT);
- Result: file has random permissions
- Check: Look for oflags == O_CREAT without mode argument

Example: Chroot protocol checker

- Goal: confine process to a “jail” on the filesystem
  - chroot() changes filesystem root for a process
- Problem
  - chroot() itself does not change current working directory



Tainting checkers

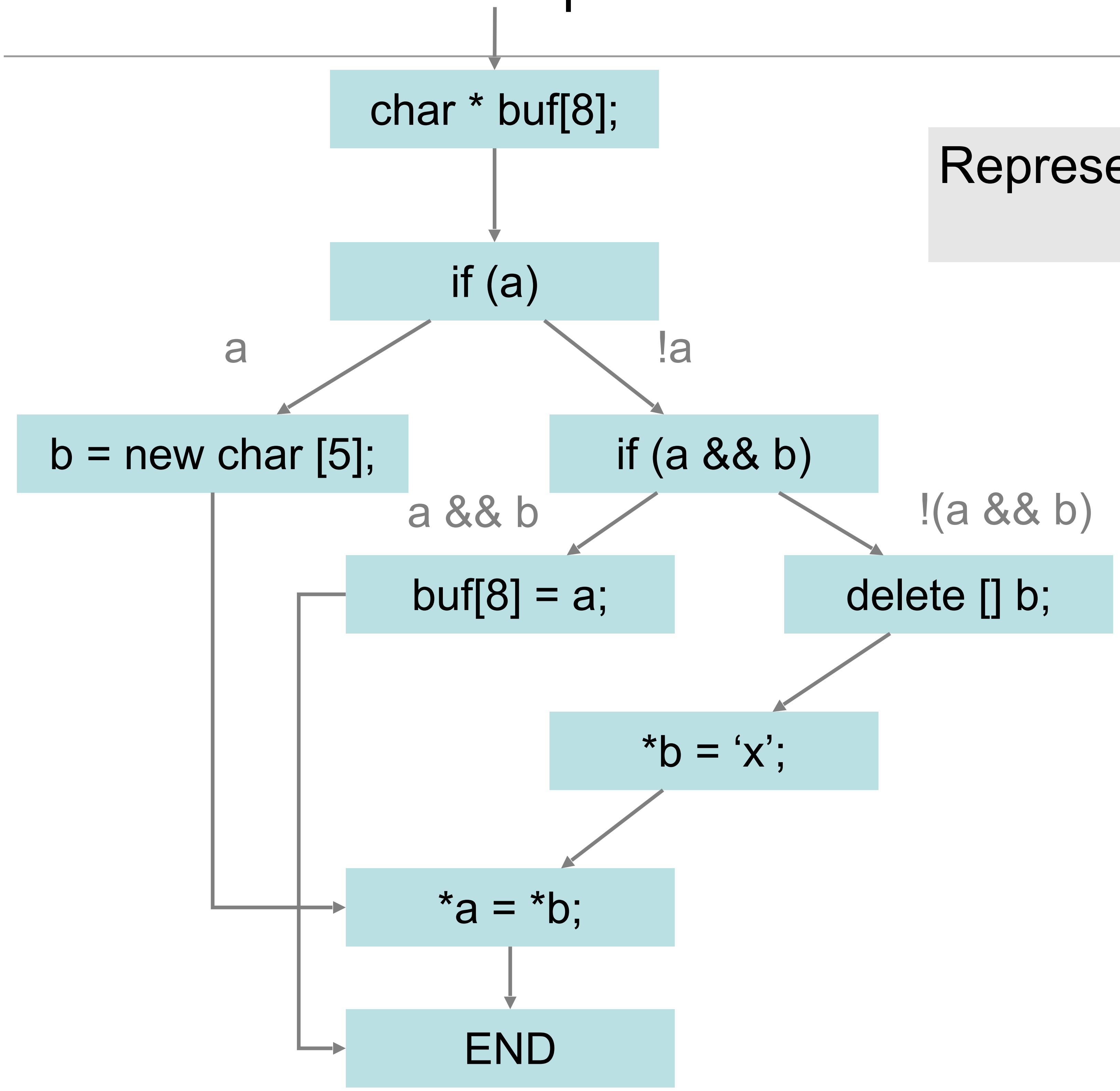


 [Mitchell’15]

Finding Local Bugs

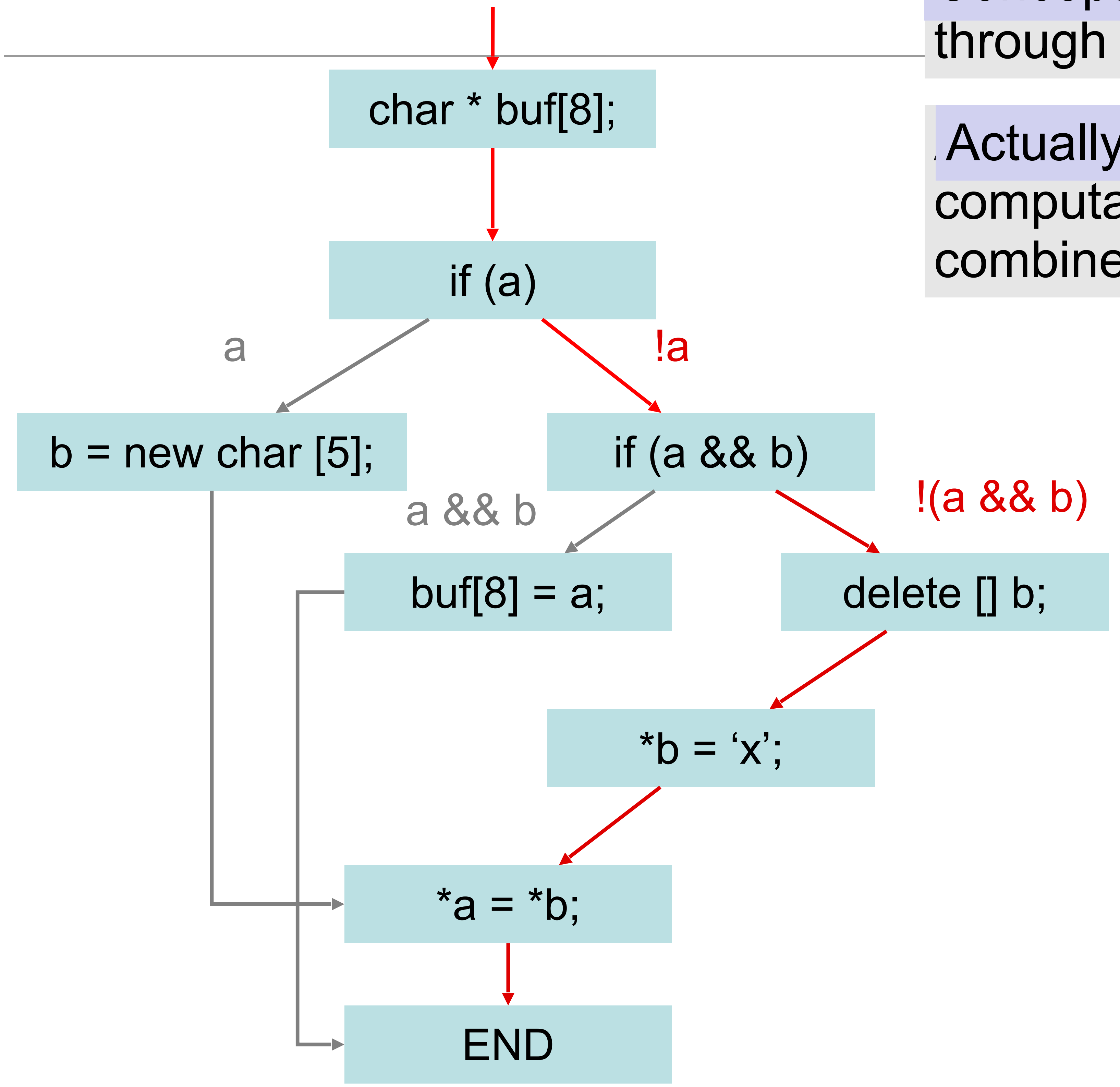
```
#define SIZE 8
void set_a_b(char * a, char * b) {
   char * buf[SIZE];
   if (a) {
      b = new char[5];
   } else {
      if (a && b) {
         buf[SIZE] = a;
         return;
      } else {
         delete [] b;
      }
      *b = 'x';
   }
   *a = *b;
}
```

Control Flow Graph



Represent logical structure of code in graph form

Path Traversal

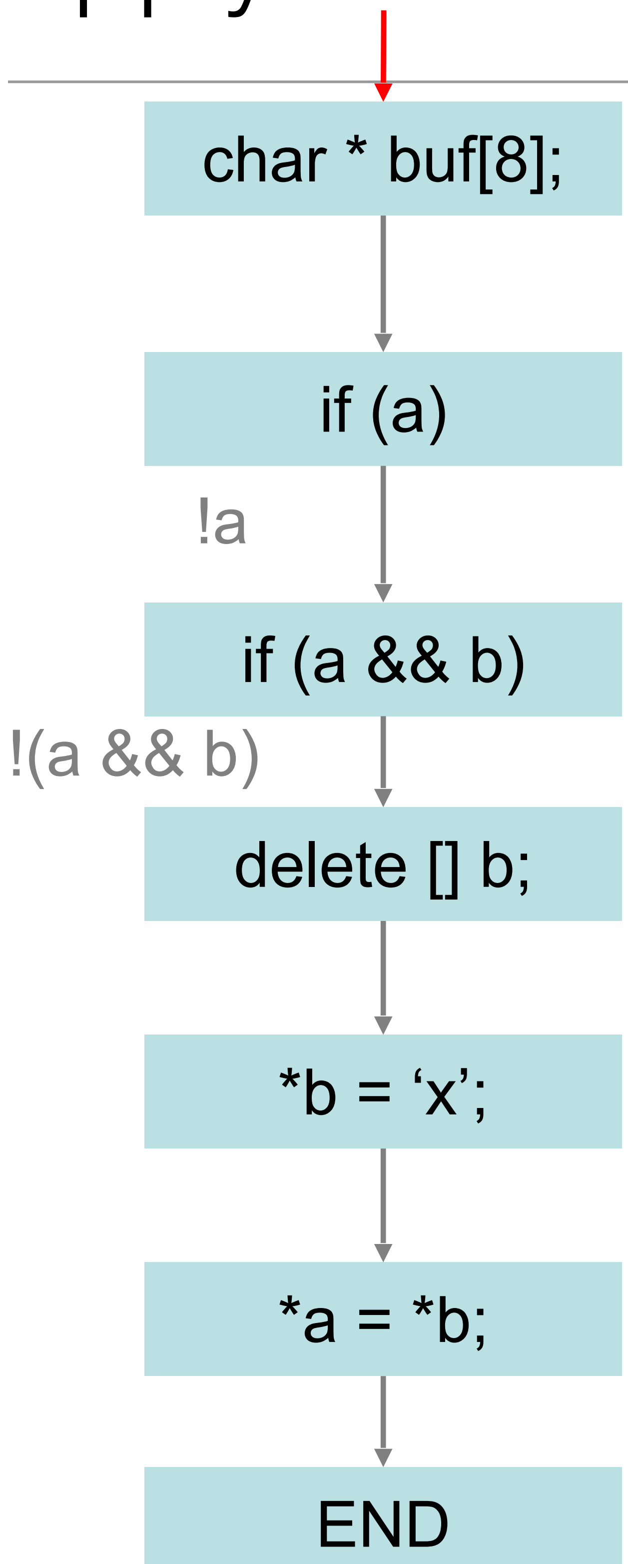


Conceptually Analyze each path through control graph separately

Actually Perform some checking computation once per node; combine paths at merge nodes

Apply Checking

Null pointers | Use after free | Array overrun

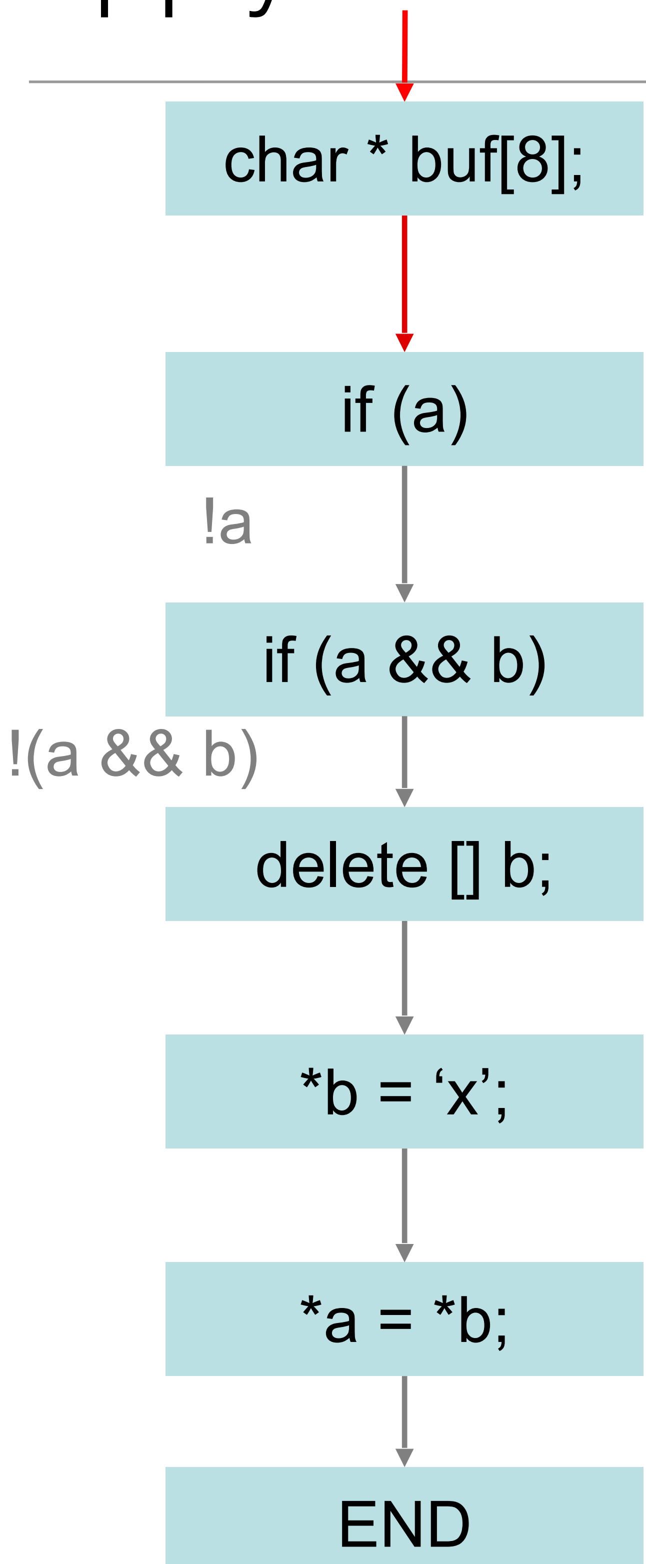


See how three checkers are run for this path

Checker
- Defined by a state diagram, with state transitions and error states

Run Checker
- Assign initial state to each program var
- State at program point depends on state at previous point, program actions
- Emit error if error state reached

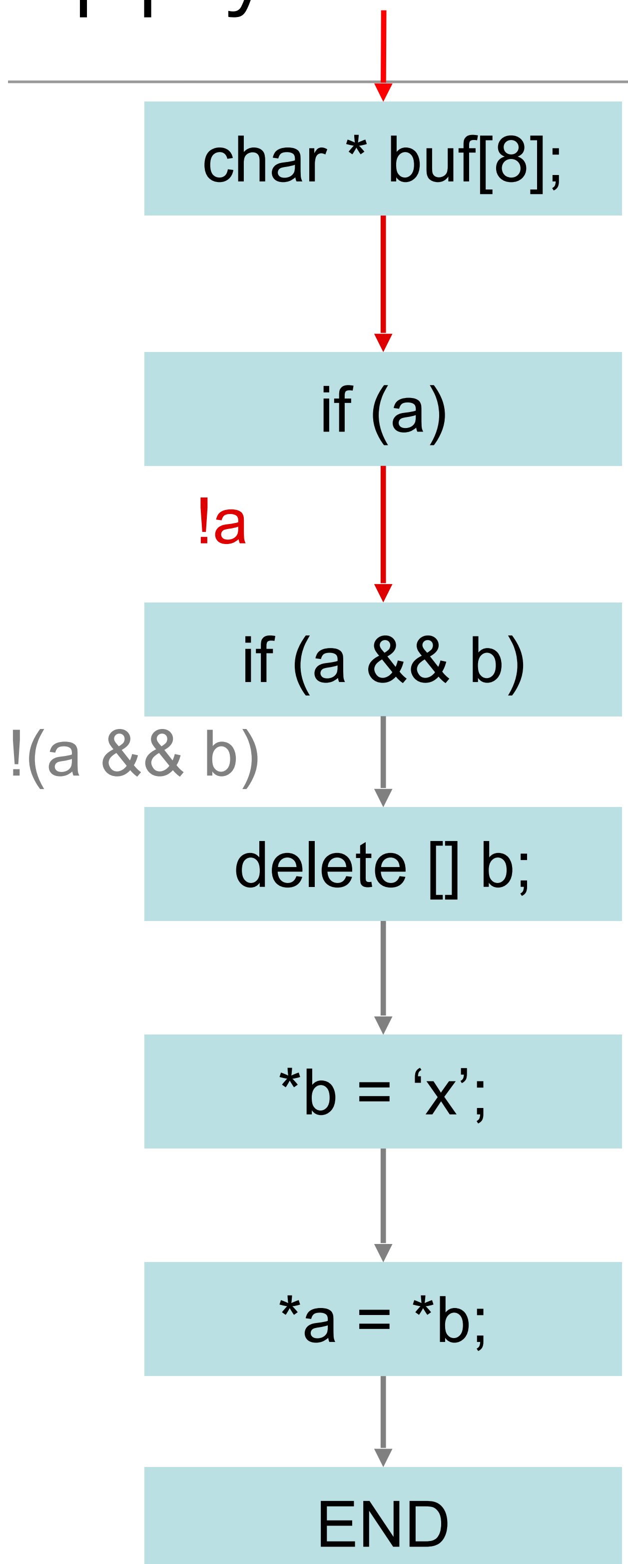
Apply Checking



Null pointers	Use after free	Array overrun
		“buf is 8 bytes”

Apply Checking

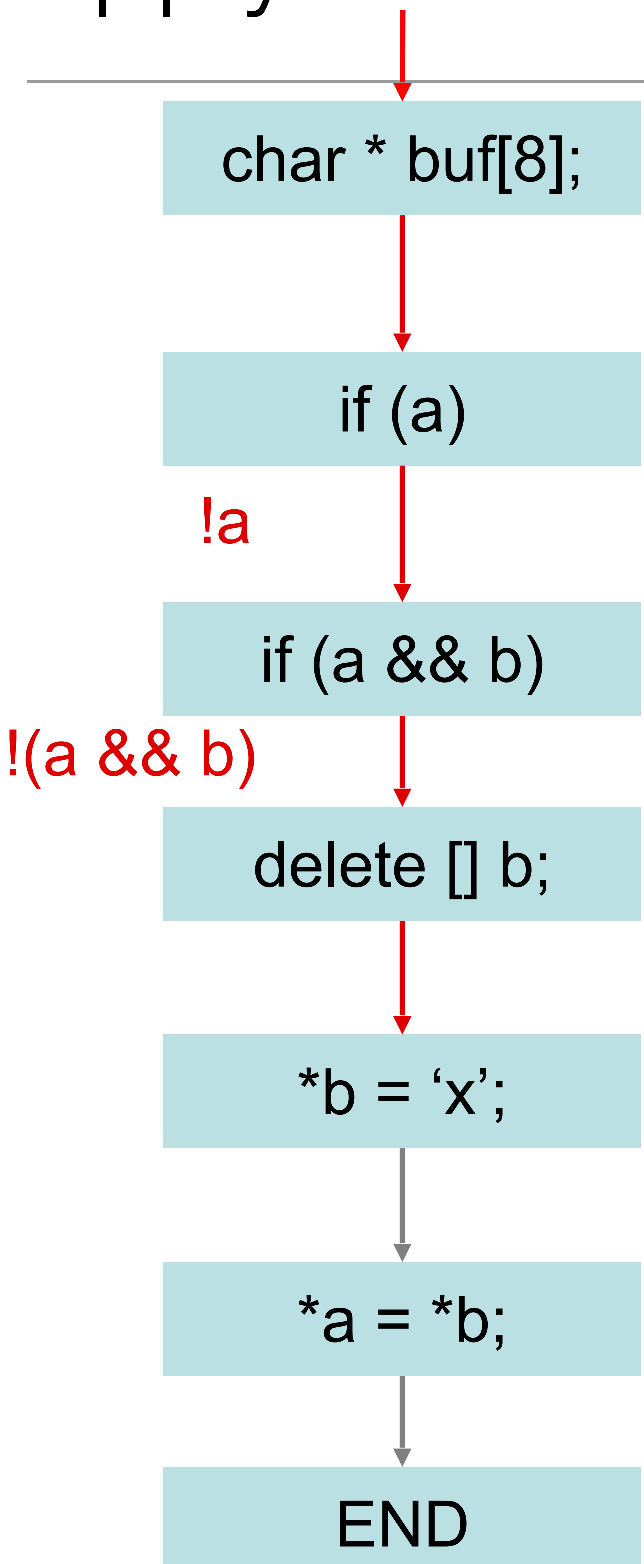
Null pointers	Use after free	Array overrun
		“buf is 8 bytes”
“a is null”		



Apply Checking

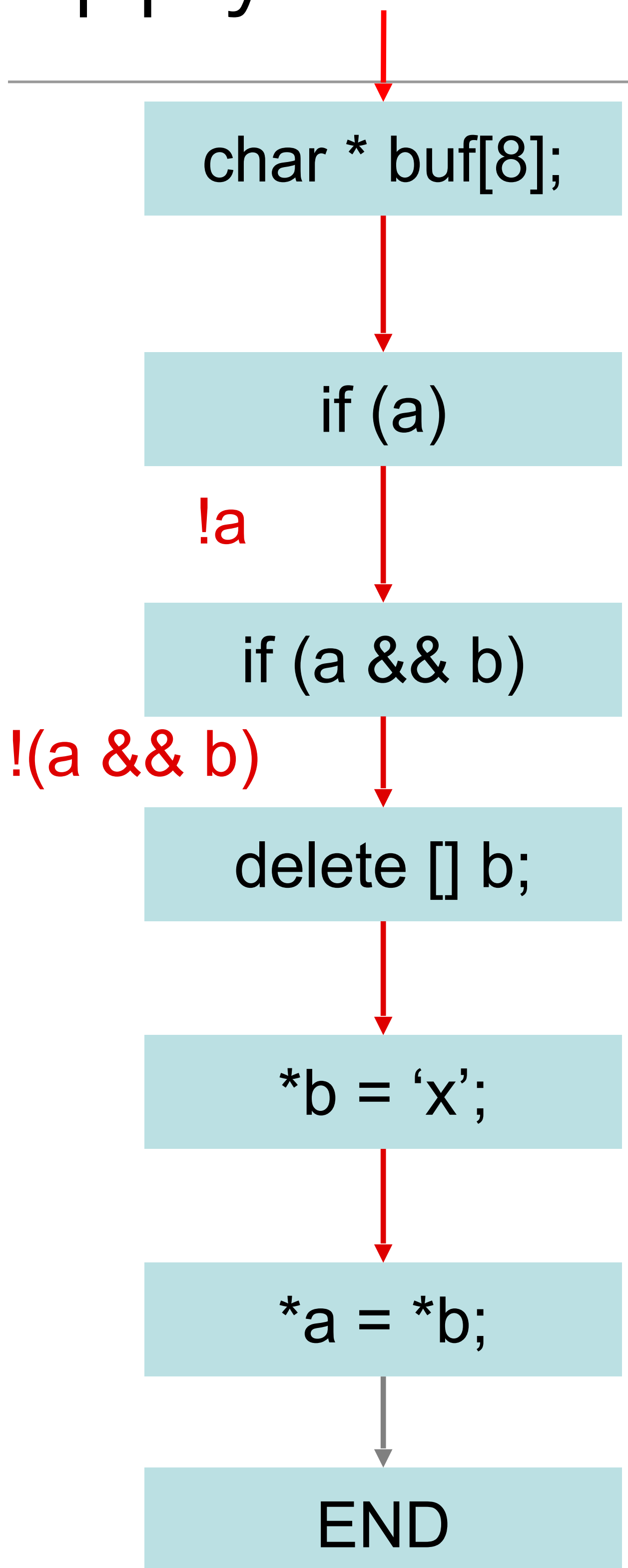
	Null pointers	Use after free	Array overrun
char * buf[8];			"buf is 8 bytes"
if (a)			
!a	"a is null"		
if (a && b)	Already knew a was null		
!(a && b)			
delete [] b;			
*b = 'x';			
*a = *b;			
END			

Apply Checking



Null pointers	Use after free	Array overrun
		“buf is 8 bytes”
“a is null”		
	“b is deleted”	

Apply Checking



Null pointers

“a is null”

Use after free

“b is deleted”

“b dereferenced!”

Array overrun

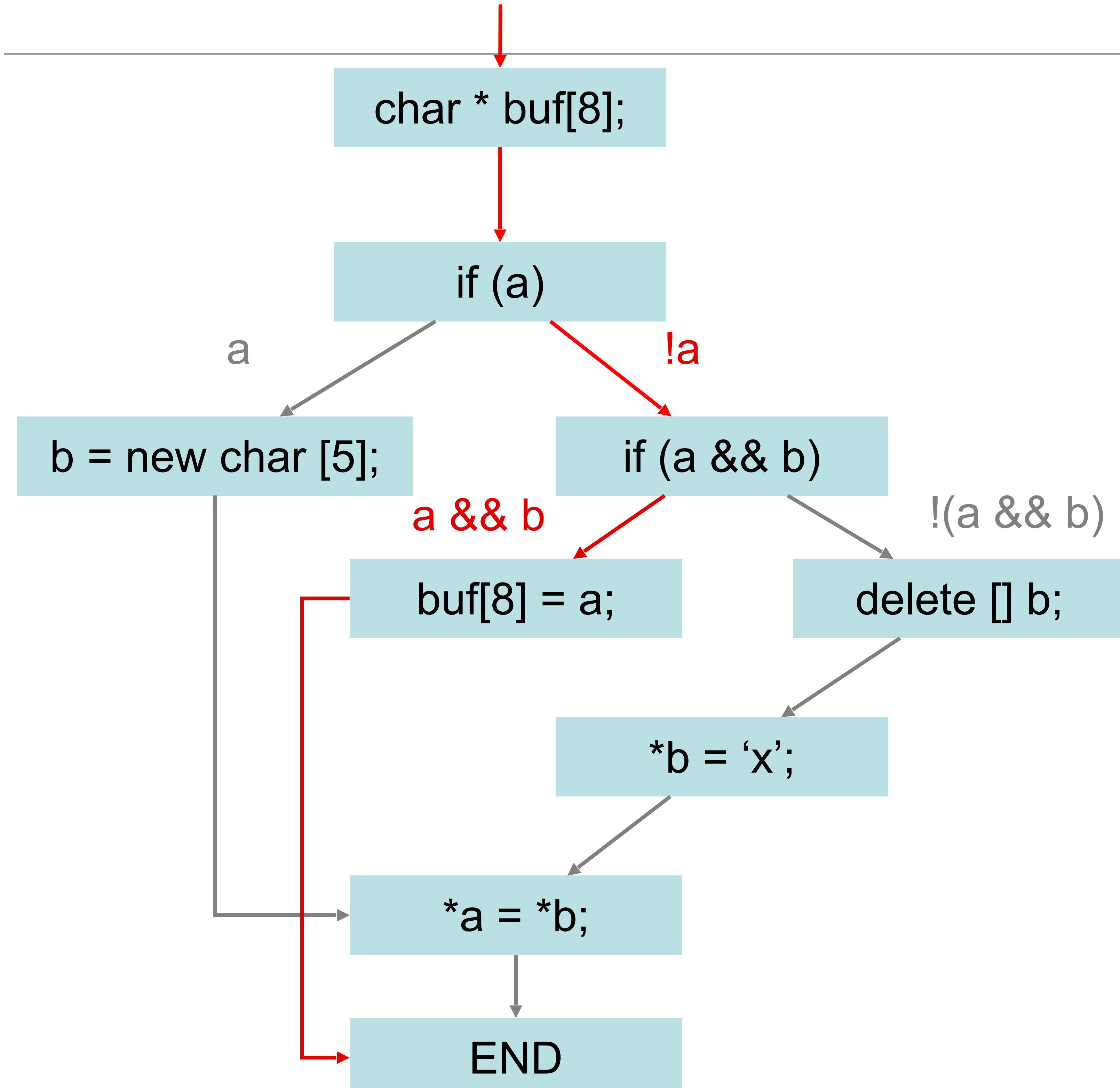
“buf is 8 bytes”

False Positives

- What is a bug? Something the user will fix.

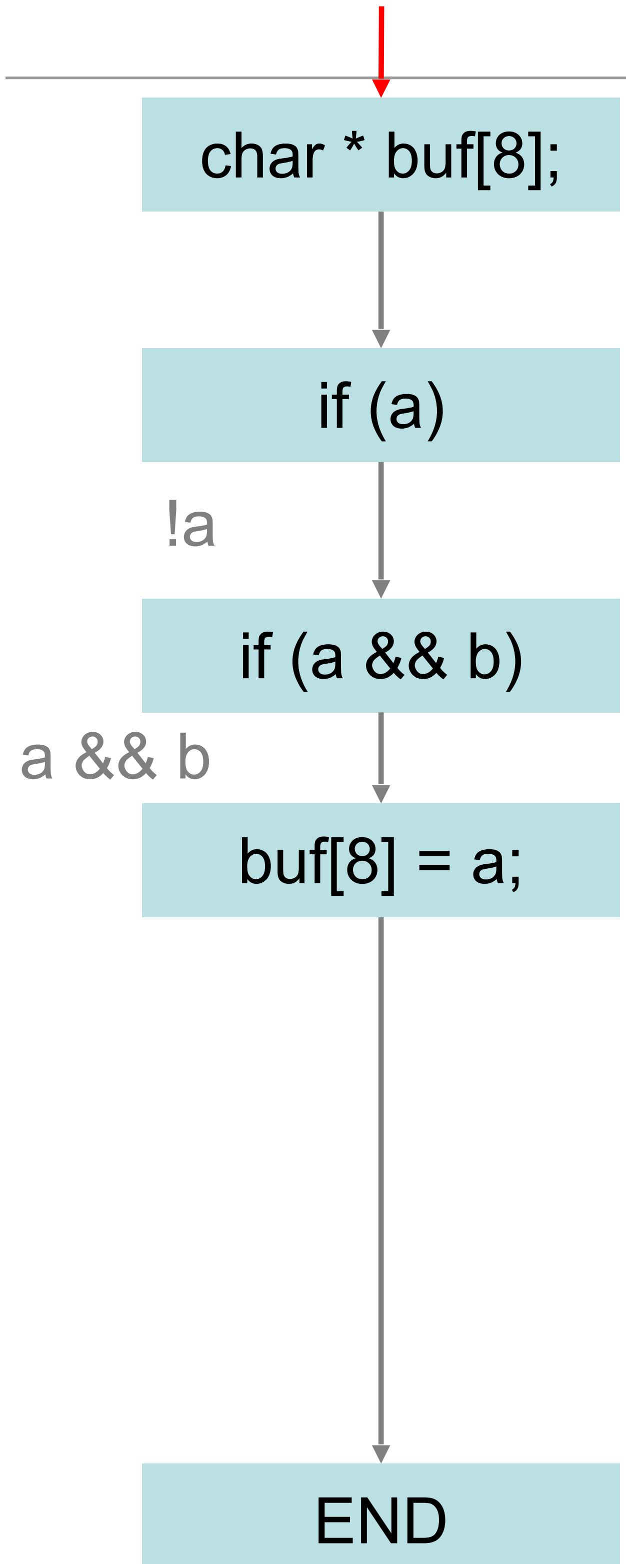
- Many sources of false positives
  - False paths
  - Execution environment assumptions
  - Killpaths
  - Conditional compilation
  - “third party code”
  - Analysis imprecision
  - …

A False Path

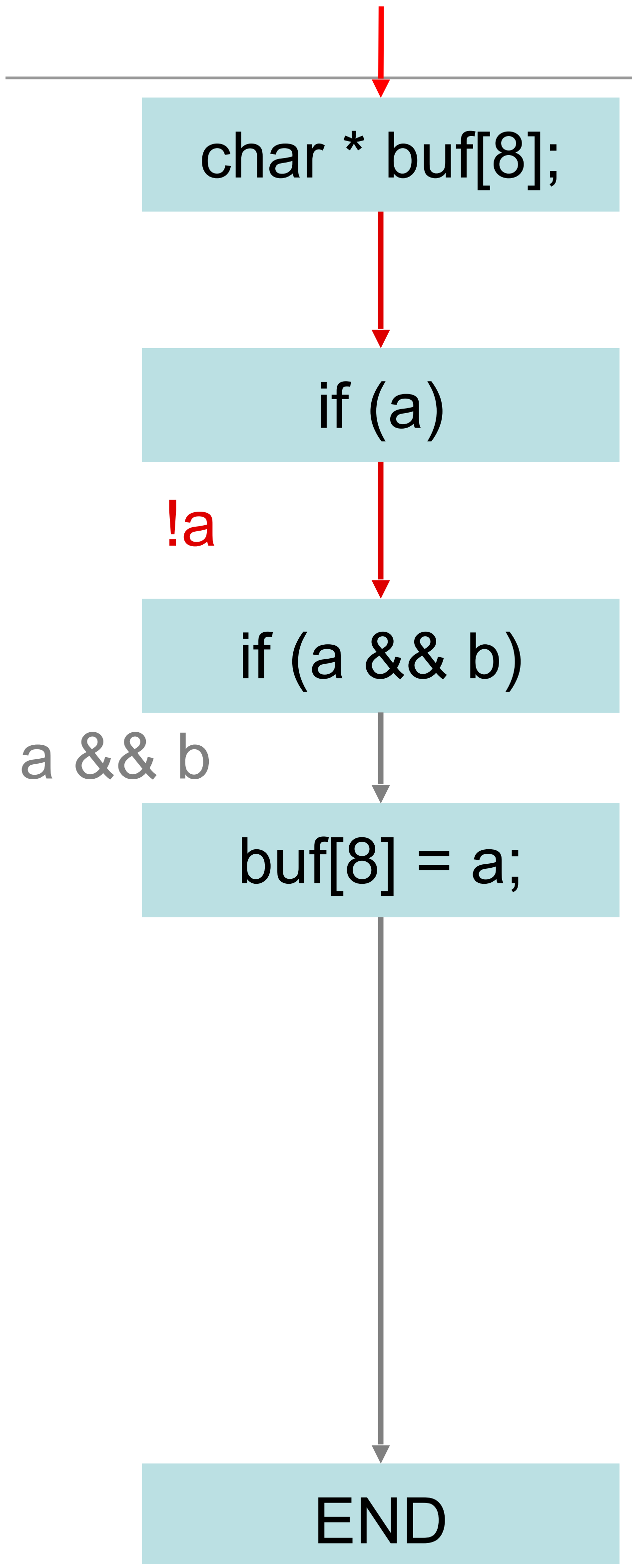


False Path Pruning

Integer Range | Disequality | Branch

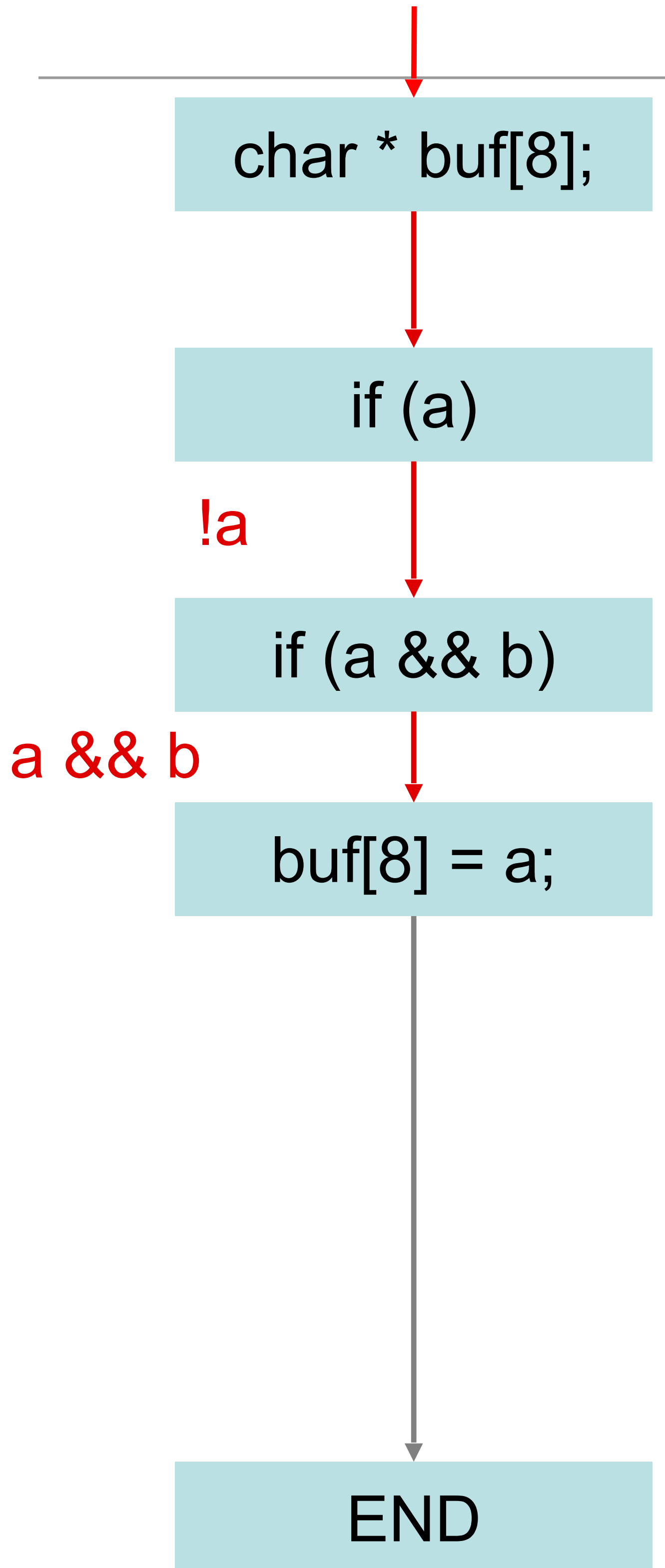


False Path Pruning



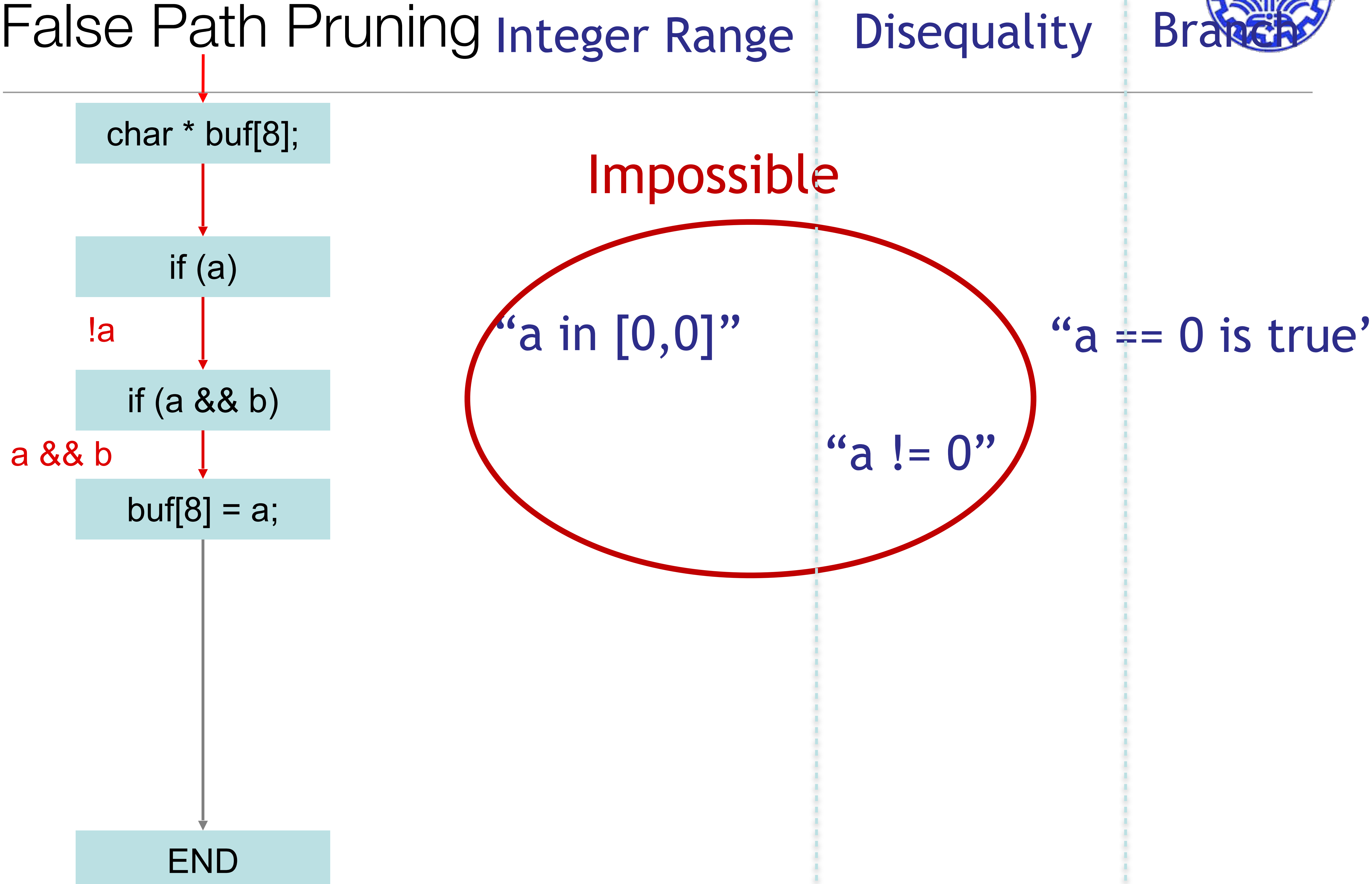
Integer Range	Disequality	Branch
"a in [0,0]"		"a == 0 is true"

False Path Pruning



Integer Range	Disequality	Branch
“a in [0,0]”		“a == 0 is true”
	“a != 0”	

False Path Pruning



Goal: find as many serious bugs as possible

- Problem: what are the rules?!?!
  - 100-1000s of rules in 100-1000s of subsystems.
  - To check, must answer: Must a() follow b()? Can foo() fail? Does bar(p) free p? Does lock l protect x?
  - Manually finding rules is hard. So don't. Instead infer what code believes, cross check for contradiction
- Intuition: how to find errors without knowing truth?
  - Contradiction. To find lies: cross-examine. Any contradiction is an error.
  - Deviance. To infer correct behavior: if 1 person does X, might be right or a coincidence. If 1000s do X and 1 does Y, probably an error.
  - Crucial: we know contradiction is an error without knowing the correct belief!

Cross-checking program belief systems

- MUST beliefs:
  - Inferred from acts that imply beliefs code *must* have.

```
x = *p / z;  // MUST belief: p not null
             // MUST: z != 0
unlock(l);   // MUST: l acquired
x++;         // MUST: x not protected by l
```

  - Check using internal consistency: infer beliefs at different locations, then cross-check for contradiction
- MAY beliefs: could be coincidental
  - Inferred from acts that imply beliefs code *may* have

```
A();   A();   A();   A();
…      …      …      …
B();   B();   B();   B();   // MAY: A() and B()
                            // must be paired
```

```
B();  // MUST: B() need not
      // be preceded by A()
```

  - Check as MUST beliefs; rank errors by belief confidence.

Environment Assumptions

- Should the return value of malloc() be checked?

```
int *p = malloc(sizeof(int));
*p = 42;
```

OS Kernel: Crash machine.	File server: Pause filesystem.	Web application: 200ms downtime
Spreadsheet: Lose unsaved changes.	Game: Annoy user.	IP Phone: Annoy user.
Library: ?	Medical device: malloc?!	

Statistical Analysis

- Assume the code is usually right

3/4 deref

```
int *p = malloc(sizeof(int));
*p = 42;

int *p = malloc(sizeof(int));
*p = 42;

int *p = malloc(sizeof(int));
*p = 42;

int *p = malloc(sizeof(int));
if(p)  *p = 42;
```

1/4 deref

```
int *p = malloc(sizeof(int));
if(p) *p = 42;

int *p = malloc(sizeof(int));
if(p) *p = 42;

int *p = malloc(sizeof(int));
if(p) *p = 42;

int *p = malloc(sizeof(int));
*p = 42;
```

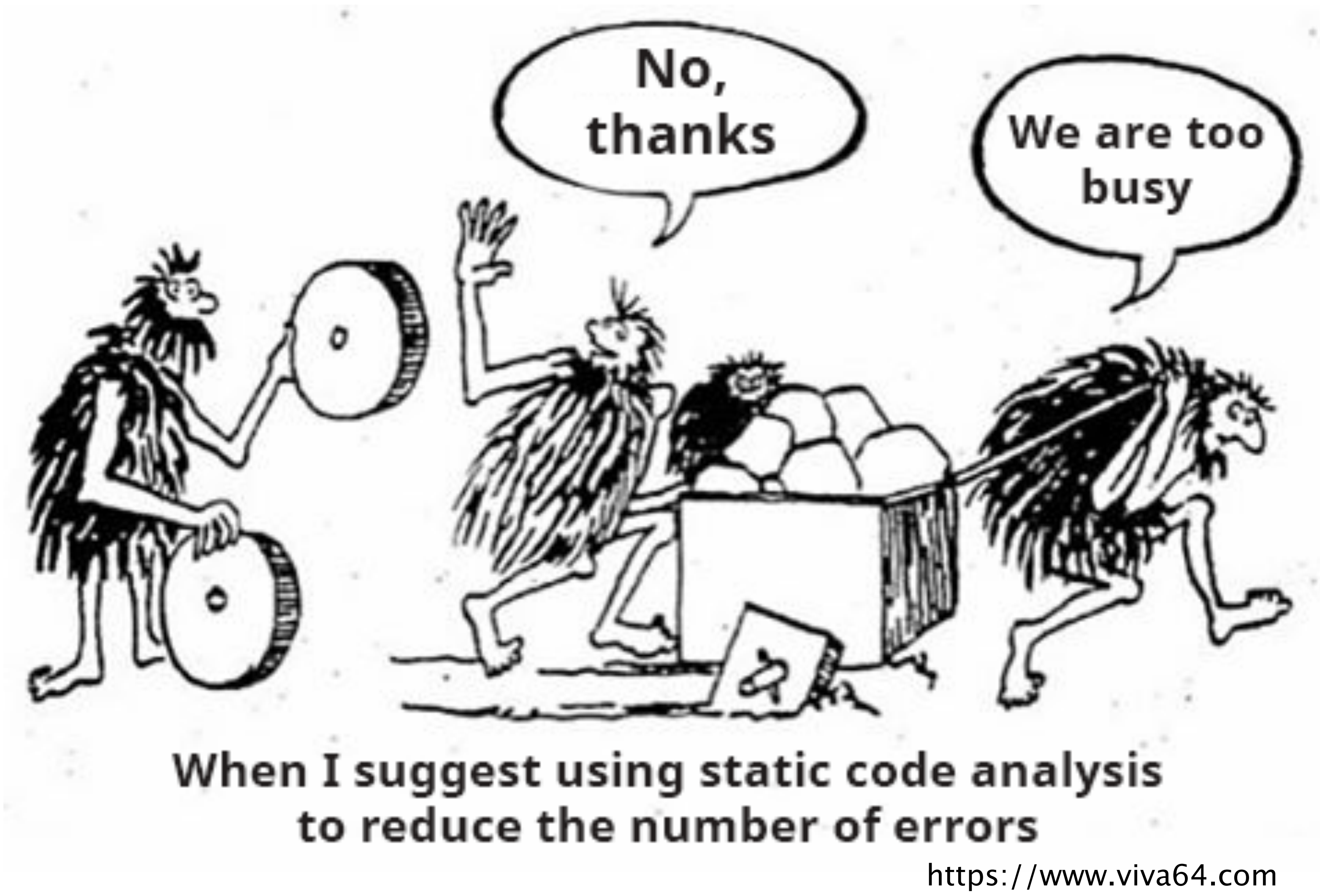
Results for BSD and Linux

- All bugs released to implementers; most serious fixed

Violation	Linux Bug	Linux Fixed	BSD Bug	BSD Fixed
Gain control of system	18	15	3	3
Corrupt memory	43	17	2	2
Read arbitrary memory	19	14	7	7
Denial of service	17	5	0	0
Minor	28	1	0	0
Total	125	52	12	12

Program Analysis

- How could we analyze a program (with source code) and look for problems?
- How accurate would our analysis be without executing the code?
- If we execute the code, what input values should we use to test/analyze the code?



Symbolic Execution

Symbolic Execution --- History

- 1976: A system to generate test data and symbolically execute programs (Lori Clarke)
- 1976: Symbolic execution and program testing (James King)
- 2005-present: practical symbolic execution
  - Using SMT solvers
  - Heuristics to control exponential explosion
  - Heap modeling and reasoning about pointers
  - Environment modeling
  - Dealing with solver limitations

[Chowdhury'15]

Motivation

- Writing and maintaining tests is tedious and error-prone
- Idea: Automated Test Generation
  - Generate regression test suite
  - Execute all reachable statements

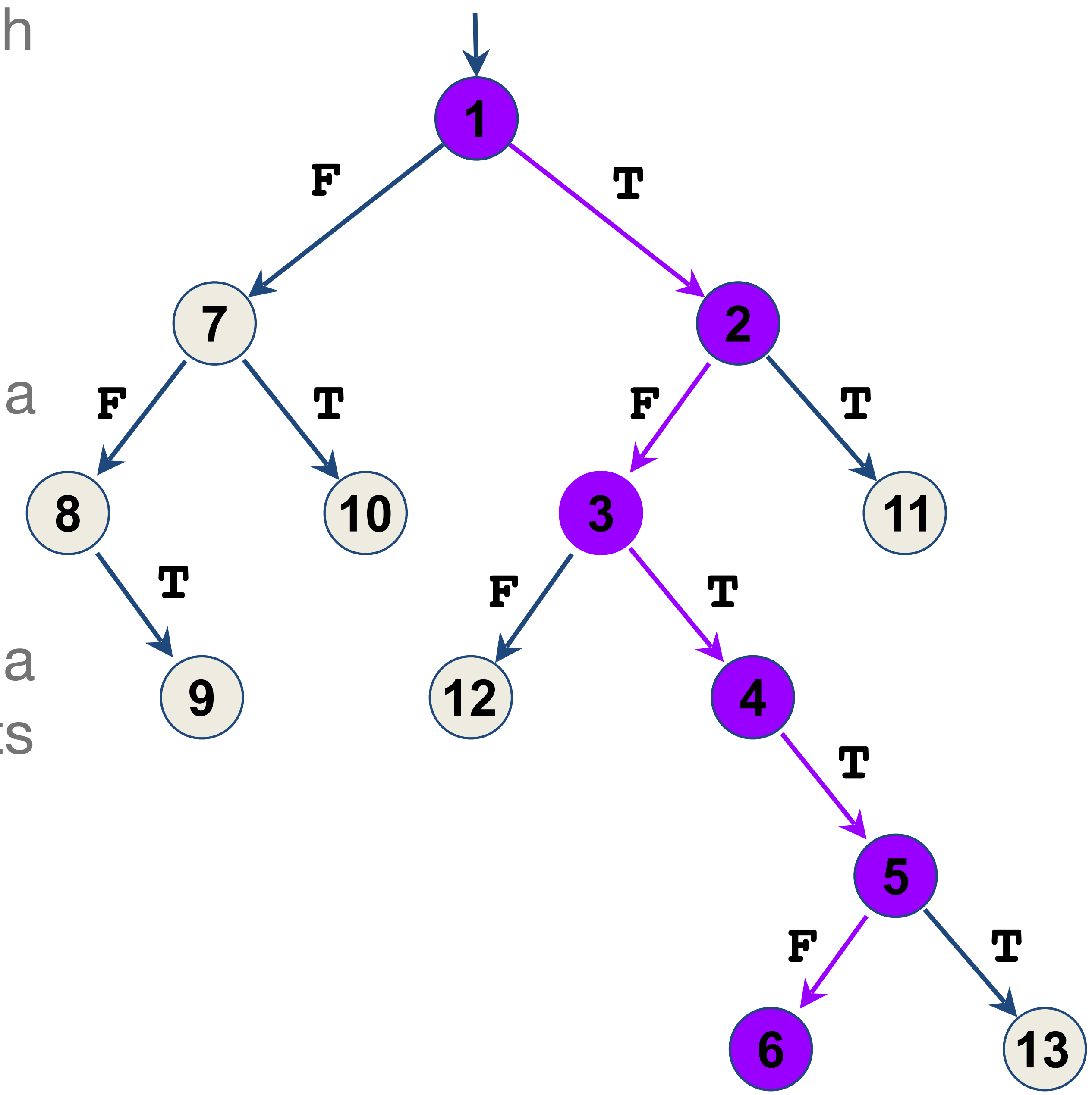
Approach

- Dynamic Symbolic Execution
  - Stores program state concretely and symbolically
  - Solves constraints to guide execution at branch points
  - Explores all execution paths of the unit tested

- Example of Hybrid Analysis
  - Collaboratively combines dynamic and static analysis

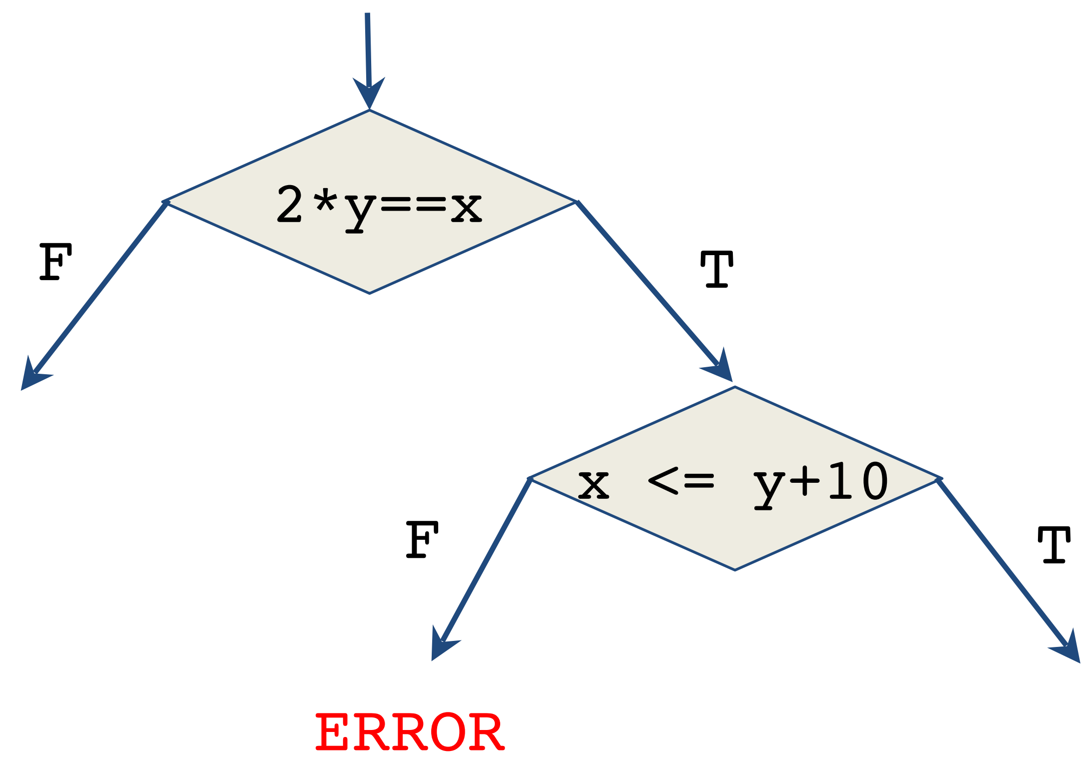
Execution Paths of a Program

- Program can be seen as binary tree with possibly infinite depth
  - Called Computation Tree
- Each node represents the execution of a conditional statement
- Each edge represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs



Example of Computation Tree

```
void test_me(int x, int y) {
    if (2*y == x) {
        if (x <= y+10)
            print("OK");
        else {
            print("something bad");
            ERROR;
        }
    } else
        print("OK");
}
```



Existing Approach I

- Random Testing:
  - Generate random inputs
  - Execute the program on those (concrete) inputs
- Problem:
  - Probability of reaching error could be astronomically small

```
void test_me(int x) {
    if (x == 94389) {
        ERROR;
    }
}
```

Probability of ERROR:

$1/2^{32} \approx 0.000000023\%$

Existing Approach II

- Symbolic Execution
  - Use symbolic values for inputs
  - Execute program symbolically on symbolic input values
  - Collect symbolic path constraints
  - Use theorem prover to check if a branch can be taken
- Problem:
  - Does not scale for large programs

```
void test_me(int x) {
    if (x*3 == 15) {
        if (x % 5 == 0)
            print("OK");
        else {
            print("something
bad");
            ERROR;
        }
    } else
        print("OK");
}
```

Existing Approach II

- Symbolic Execution
  - Use symbolic values for inputs
  - Execute program symbolically on symbolic input values
  - Collect symbolic path constraints
  - Use theorem prover to check if a branch can be taken
- Problem:
  - Does not scale for large programs

```
void test_me(int x) {
    // c = product of two
    // large primes
    if (pow(2,x) % c == 17) {
        print(“something bad”);
        ERROR;
    } else
        print(“OK”);
}
```

Symbolic execution will say both branches are reachable: False Positive

Combined Approach

- Dynamic Symbolic Execution (DSE)
  - Start with random input values
  - Keep track of both concrete values and symbolic constraints
  - Use concrete values to simplify symbolic constraints
  - Incomplete theorem-prover

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	x = x_0	
y = 7	y = y_0	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

concrete state	symbolic state	path condition
x = 22	$x = x_0$	
y = 7	$y = y_0$	
z = 14	$z = 2*y_0$	

Concrete Execution | Symbolic Execution

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 22		$x = x_0$	$2*y_0 \; != \; x_0$
y = 7		$y = y_0$	
z = 14		$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
$x = 22$	$x = x_0$		$2*y_0 \;!= x_0$
$y = 7$	$y = y_0$		
$z = 14$	$z = 2*y_0$		

Solve: $2*y_0 == x_0$

Solution: $x_0 = 2,\ y_0 = 1$

An Illustrative Example

```
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y)
{
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 2		$x = x_0$	
y = 1		$y = y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 2	$x = x_0$	
y = 1	$y = y_0$	
z = 2	$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
	concrete state	symbolic state	path condition
	x = 2	$x = x_0$	$2*y_0 == x_0$
	y = 1	$y = y_0$	
	z = 2	$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	**symbolic state**	**path condition**
$x = 2$	$x = x_0$	$2*y_0 == x_0$
$y = 1$	$y = y_0$	
$z = 2$	$z = 2*y_0$	$x_0 <= y_0+10$

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 2	$x = x_0$	$2*y_0 == x_0$
y = 1	$y = y_0$	
z = 2	$z = 2*y_0$	$x_0 <= y_0+10$

Solve: $(2*y_0 == x_0)$ and $(x_0 > y_0+10)$

Solution: $x_0 = 30$, $y_0 = 15$

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 30		$x = x_0$	
y = 15		$y = y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	path condition
	x = 30		$x = x_0$	
	y = 15		$y = y_0$	
	z = 30		$z = 2*y_0$	

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 30	$x = x_0$		$2*y_0 == x_0$
y = 15	$y = y_0$		
z = 30	$z = 2*y_0$		

An Illustrative Example

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Program Error

Concrete Execution: concrete state	Symbolic Execution: symbolic state	path condition
x = 30	$x = x_0$	$2*y_0 == x_0$
y = 15	$y = y_0$	$x_0 > y_0+10$
z = 30	$z = 2*y_0$	

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 22	$x = x_0$		
y = 7	$y = y_0$		

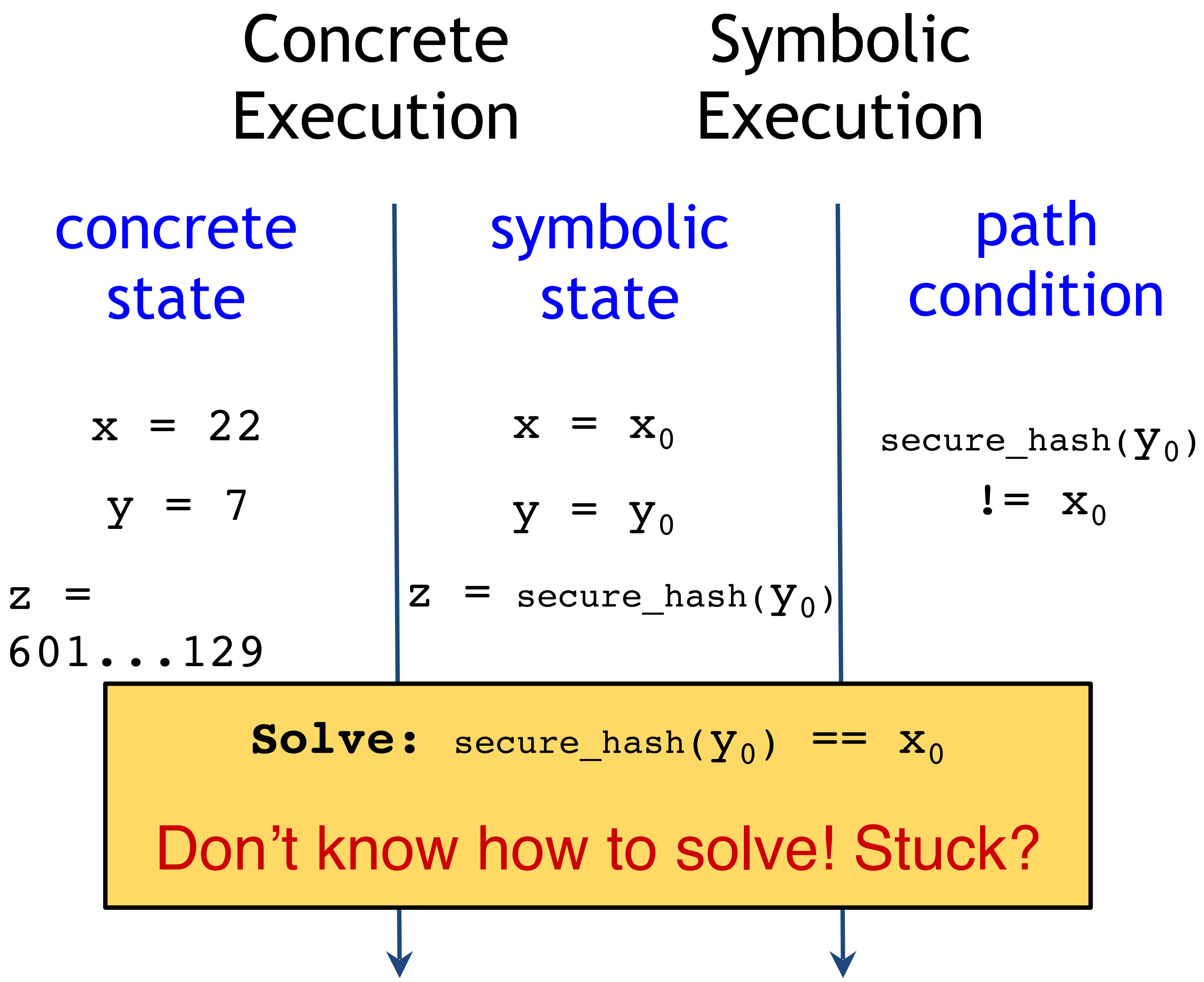
A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 22		$x = x_0$	
y = 7		$y = y_0$	
z = 601...129		$z = \text{secure\_hash}(y_0)$	

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22 y = 7 z = 601...129	$x = x_0$ $y = y_0$ $z = secure\_hash(y_0)$	$secure\_hash(y_0) \mathrel{!=} x_0$

Solve: $secure\_hash(y_0) == x_0$

Don't know how to solve! Stuck?

Not stuck! Use concrete state: replace y_0 by 7

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$secure\_hash(y_0)$!= x_0
y = 7	$y = y_0$	
z = 601...129	$z = secure\_hash(y_0)$	

Solve: $601...129 == x_0$

Solution: $x_0 = 601...129$, $y_0 = 7$

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

	Concrete Execution		Symbolic Execution	
	concrete state	symbolic state		path condition
	x = 601...129 y = 7	$x = x_0$ $y = y_0$		

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 601...129	y = 7	$x = x_0$	
		$y = y_0$	
z = 601...129		$z = secure\_hash(y_0)$	

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 601...129	$x = x_0$	$\text{secure\_hash}(y_0) == x_0$
y = 7	$y = y_0$	
z = 601...129	$z = \text{secure\_hash}(y_0)$	

A More Complex Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

Program Error

Concrete Execution	Symbolic Execution	
concrete state	**symbolic state**	**path condition**
x = 601...129	$x = x_0$	$\text{secure\_hash}(y_0) == x_0$
y = 7	$y = y_0$	$x_0 > y_0+10$
z = 601...129	$z = \text{secure\_hash}(y_0)$	

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 22		$x = x_0$	
y = 7		$y = y_0$	

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$x_0 \mathrel{!=} y_0$
y = 7	$y = y_0$	

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$x_0 \;!=\; y_0$
y = 7	$y = y_0$	$\text{secure\_hash}(x_0) \;!=\; \text{secure\_hash}(y_0)$

Solve: $x_0 \;!=\; y_0$ and $\text{secure\_hash}(x_0) == \text{secure\_hash}(y_0)$

Use concrete state: replace y_0 by 7.

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$x_0 \,!\!= y_0$
y = 7	$y = y_0$	$\text{secure\_hash}(x_0) \,!\!= \text{secure\_hash}(y_0)$

Solve: $x_0 \,!\!= 7$ and $\text{secure\_hash}(x_0) == 601...129$

Use concrete state: replace x_0 by 22.

A Third Example

```
int foo(int v) {
   return secure_hash(v);
}

void test_me(int x, int y)
{
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

False negative!

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$x_0 \ != y_0$
y = 7	$y = y_0$	secure_hash(x_0) != secure_hash(y_0)

Solve: 22 != 7 and 438...861 == 601...129

Unsatisfiable!

Another Example: Testing Data Structures

- Random Test Driver:
  - random value for x
  - random memory graph reachable from p
- Probability of reaching ERROR is extremely low

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	x = x_0	
p = NULL	p = p_0	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution
concrete state	symbolic state	path condition
x = 236 p = NULL	$x = x_0$ $p = p_0$	$x_0 > 0$

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 236 p = NULL		$x = x_0$ $p = p_0$	$x_0 > 0$ p_0 == NULL

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	$x_0 > 0$
p = NULL	$p = p_0$	p_0 == NULL

Solve: $x_0 > 0$ and p_0 != NULL

Solution: $x_0 = 236$, p_0 → [634 | NULL]

Data-Structure Example

```c
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	
	$p = p_0$	
p → [634 | NULL]	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	$x_0 > 0$
p → [634 | NULL]	$p = p_0$	
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	$x_0 > 0$
p → | 634 | NULL |	$p = p_0$	p_0 != NULL
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	$x_0 > 0$
p → [634 | NULL]	$p = p_0$	p_0 != NULL
	$p\text{->}data = v_0$	$2*x_0+1$!= v_0
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
  if (x > 0)
    if (p != NULL)
      if (foo(x) == p->data)
        if (p->next == p)
          ERROR;
  return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	**symbolic state**	**path condition**
x = 236	x = x_0	$x_0 > 0$
p → [634 | NULL]	p = p_0	p_0 != NULL
	p->data = v_0	$2*x_0+1$!= v_0
	p->next = n_0	

Solve: $x_0 > 0$ and p_0 != NULL and $2*x_0+1==v_0$

Solution: $x_0 = 1$, p_0 → [3 | NULL]

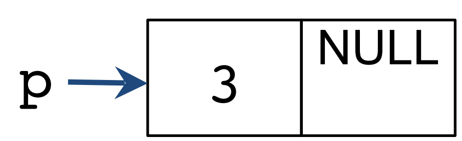
Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition

concrete state:

x = 1



symbolic state:

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition:

Data-Structure Example

```c
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	$x = x_0$	$x_0 > 0$
p → [3 | NULL]	$p = p_0$	
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	$x = x_0$	$x_0 > 0$
p → [3 | NULL]	$p = p_0$	p_0 != NULL
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 1$	$x = x_0$	$x_0 > 0$
p → [3 | NULL]	$p = p_0$	p_0 != NULL
	p->data $= v_0$	$2*x_0+1 == v_0$
	p->next $= n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	**symbolic state**	**path condition**
$x = 1$	$x = x_0$	$x_0 > 0$
p → [3 | NULL]	$p = p_0$	$p_0 \,!= NULL$
	$p\text{->}data = v_0$	$2*x_0+1 == v_0$
	$p\text{->}next = n_0$	$n_0 \,!= p_0$

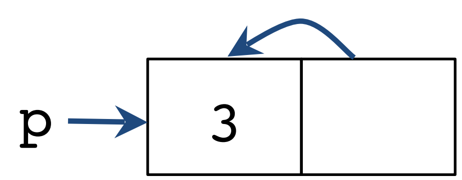
Solve: $x_0 > 0$ and $p_0 \,!= NULL$ and $2*x_0+1==v_0$ and $n_0 == p_0$

Solution: $x_0 = 1$, p_0 → [3 |]

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	x = x_0 p = p_0 p->data = v_0 p->next = n_0	

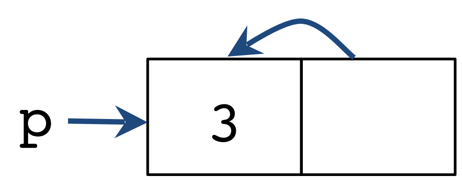


Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution | **Symbolic Execution**

concrete state	symbolic state	path condition
x = 1	$x = x_0$ $p = p_0$ $p\text{->}data = v_0$ $p\text{->}next = n_0$	$x_0 > 0$



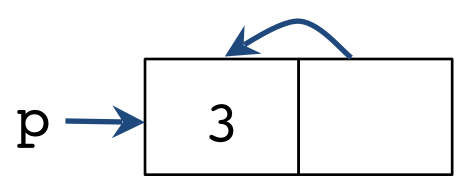
Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution

concrete state

x = 1



Symbolic Execution

symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

$x_0 > 0$

$p_0 \;!= \mathrm{NULL}$

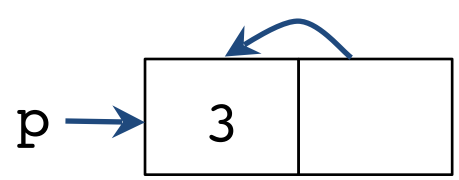
Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution | **Symbolic Execution**

concrete state

x = 1



symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

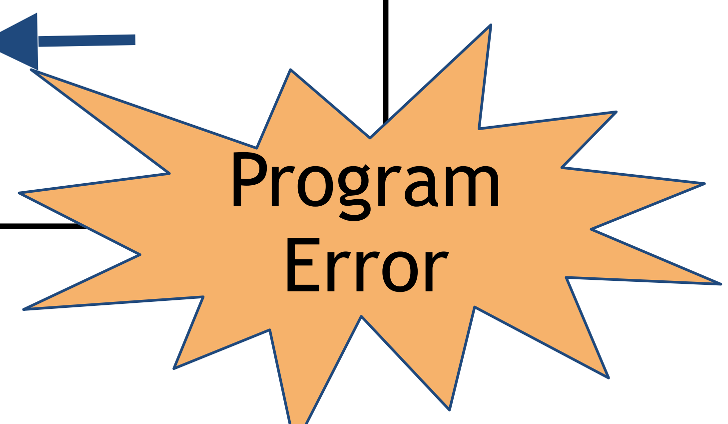
$x_0 > 0$

$p_0 \;!= NULL$

$2*x_0+1 == v_0$

Data-Structure Example

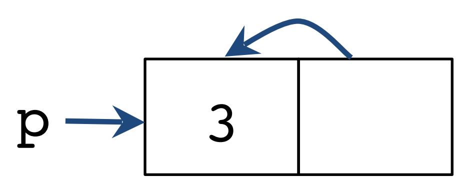
```
typedef struct cell {
    int data;
    struct cell *next;
} cell;


int foo(int v) { return 2*v + 1; }


int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```



Concrete Execution

concrete state

$x = 1$



Symbolic Execution

symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

$x_0 > 0$

$p_0\ \text{!=}\ NULL$

$2 * x_0 + 1 == v_0$

$n_0\ \text{!=}\ p_0$

Approach in a Nutshell

- Generate concrete inputs, each taking different program path
- On each input, execute program both concretely and symbolically
- Both cooperate with each other:
  - Concrete execution guides symbolic execution
    - Enables it to overcome incompleteness of theorem prover
  - Symbolic execution guides generation of concrete inputs
    - Increases program code coverage

Realistic Implementations

- KLEE: LLVM (C family of languages)
- PEX: .NET Framework
- jCUTE: Java
- Jalangi: Javascript
- SAGE and S2E: binaries (x86, ARM, ...)

How does Symbolic Execution Find bugs?

- It is possible to extend symbolic execution to help us catch bugs
- How: Dedicated checkers
  - Divide by zero example --- y = x / z where x and z are symbolic variables and assume current PC (i.e. path constraint) is f
  - Even though we only fork in branches we will now fork in the division operator
  - One branch in which z = 0 and another where z !=0
  - We will get two paths with the following constraints:
  - z = 0 && f, z != 0 && f
  - Solving the constraint z = 0 && f will give us concrete input values that will trigger the divide by zero error.

How does Symbolic Execution Find bugs?

- It is possible to extend symbolic execution to help us catch b
- How: Dedicated checkers
  - Divide by zero example --- y = x / z where x … s and assume current PC (i.e. path constr…
  - Even though we only fork in bran… division operator
  - One branch in which z… 0
  - We will get two … nstraints:
  - z = 0 &&…
  - Sol… will give us concrete input values that will … error.

Write a dedicated checker for each kind of bug (e.g., buffer overflow, integer overflow, integer underflow)

Classic Symbolic Execution --- Practical Issues

- Loops and recursions --- infinite execution tree
- Path explosion --- exponentially many paths
- Heap modeling --- symbolic data structures and pointers
- SMT solver limitations --- dealing with complex path constraints
- Environment modeling --- dealing with native / system/library calls/file operations/network events

 [Chowdhury'15]

Acknowledgments/References (1/2)

- [Naik’18] IS 700: Software Analysis and Testing, Mayur Naik, Upenn Fall 2018.
- [Levin’18] ENEE457/CMSC498E Computer Systems Security, Dana Dachman-Soled, UMD, Fall 2017
- [Jana’17] COMS W4995: Secure Software Development: Theory and Practice, Sumana Jana, Columbia Univ, Spring 2017.
- [Aldrich’11] 17-654: Analysis of Software Artifacts, Jonathan Aldrich, CMU, Spring 2011.
- [Thornton’05] CS5204 Operating Systems course presentation by Matthew Thornton, Fall 2005.
- [Engler’02] Finding bugs with system-specific static analysis, Dawson Engler, PASTE 2002.
- [Mitchell’15] CS155 Computer and Network Security, John Mitchell, Stanford, Spring 2017.

Acknowledgments/References (2/2)

- [Naik'18] IS 700: Software Analysis and Testing, Mayur Naik, Upenn Fall 2018.
- [Chowdhury'15] Information Security, CS 526, Omar Chowdhury, University of Iowa, 2015
- [Leibowitz'13] Presented by Yoni Leibowitz, EECS 395/495: Programming Languages and Analysis for Security , Northwestern University, 2013
- [Ramos'15] Under-Constrained Symbolic Execution: Correctness Checking for Real Code, David A. Ramos and Dawson Engler, Slidesm, Usenix Security 2015
- [Engler'08] A couple billion lines of code later: static checking in the real world, Dawson Engler, Slides from Usenix Security 2008.