CE 874 - Secure Software Systems

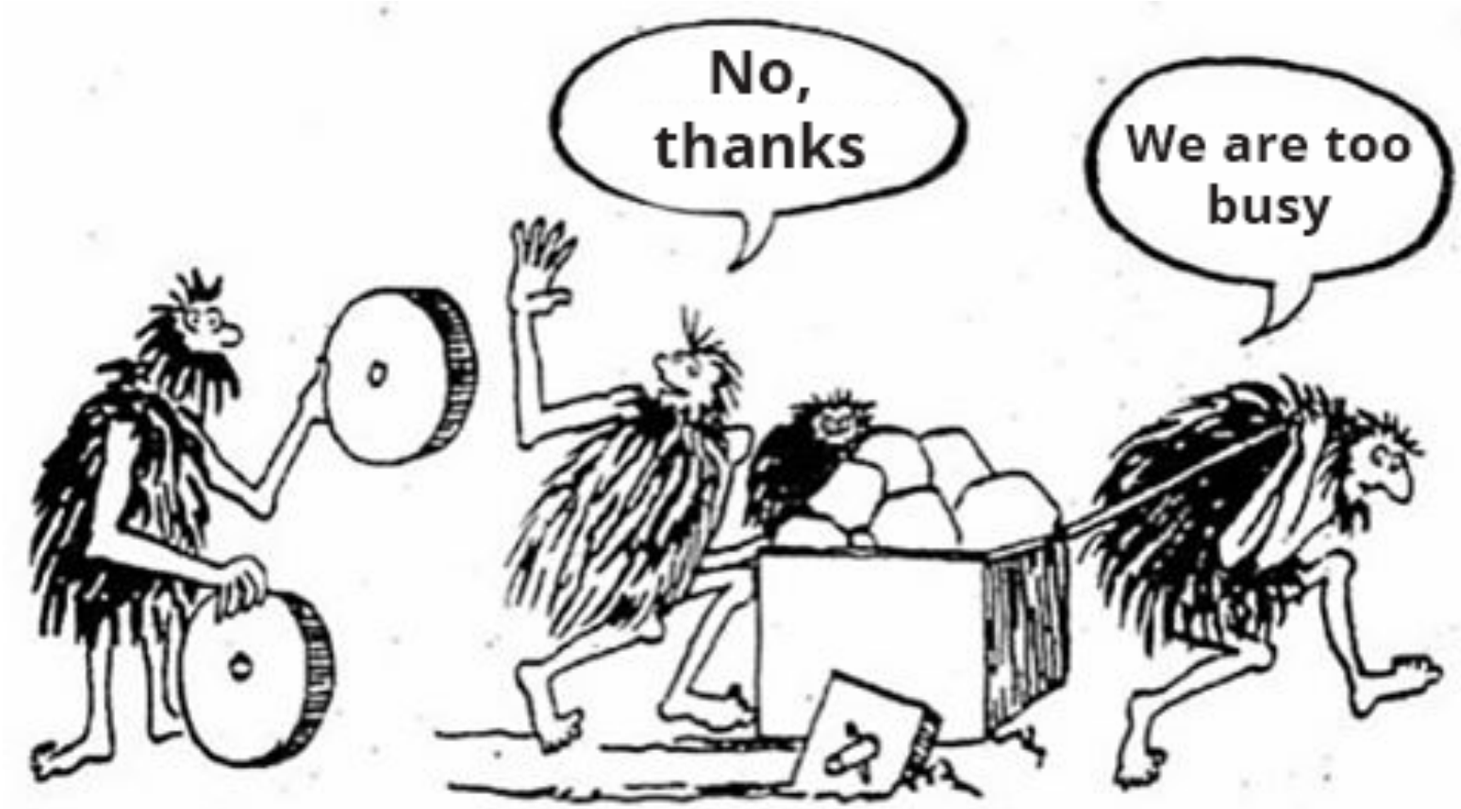
Program Analysis

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?
- What if we don't have the source 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*

Program Invariants

- An invariant at the end of the program is (z == c) for some constant c. What is c?

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) - 7;
                        z = ?
}
```

QUIZ: Program Invariants

- An invariant at the end of the program is (z == c) for some constant c. What is c?

Disaster averted!

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) - 7;

  if (z != 42)
      disaster();
}
```

z = 42

Discovering Invariants By Dynamic Analysis

- (z == 42) might be an invariant
- (z == 30) is definitely not an invariant

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) - 7;

  if (z != 42)
      disaster();
}
```

z = 42

Discovering Invariants By Static Analysis

is definitely

- (z == 42) ~~might be~~ an invariant

- (z == 30) is definitely not an invariant

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) - 7;
                                  z = 42
  if (z != 42)
      disaster();
}
```

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

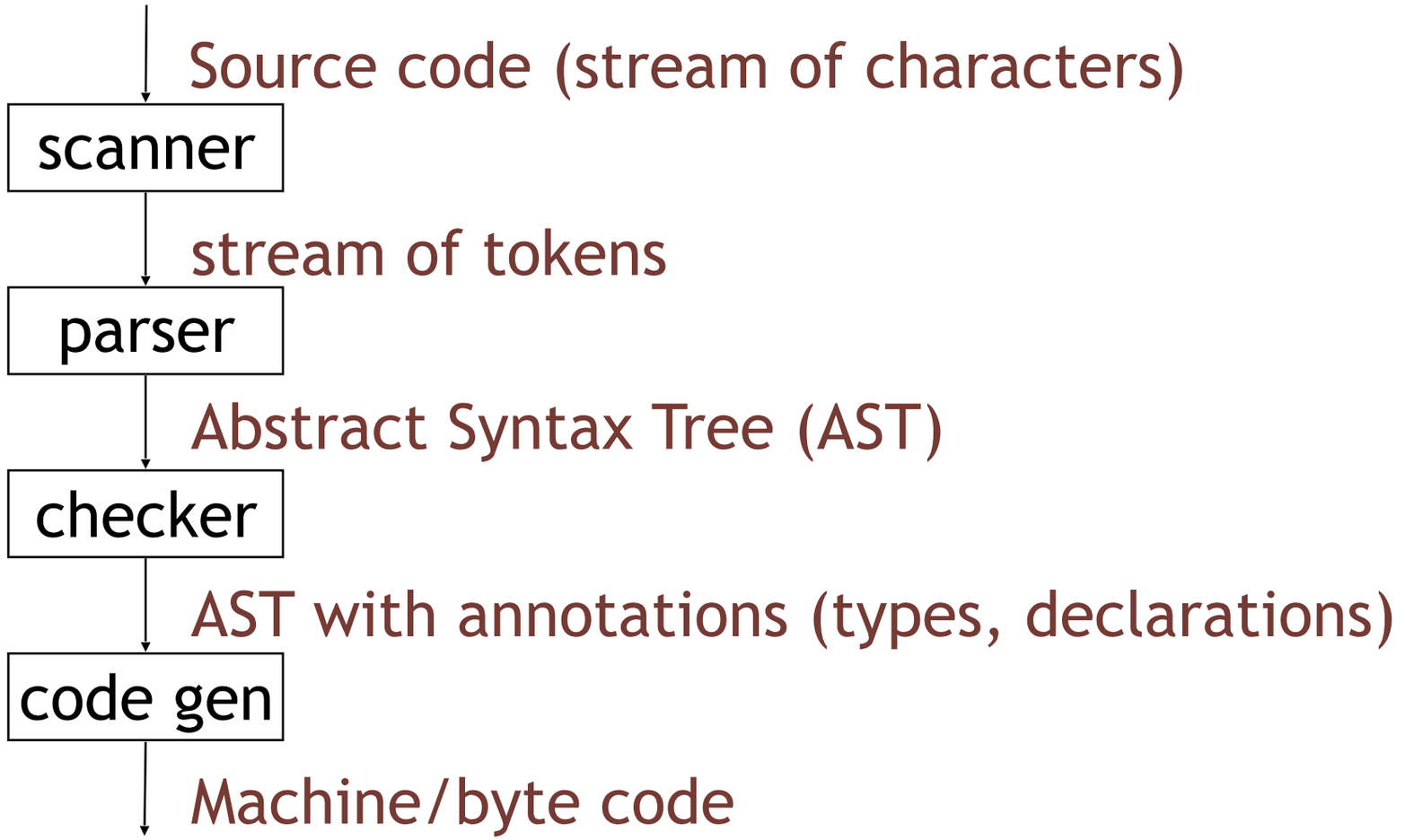
- Soundness: No error found = no error exists
  - Alarms may be false errors
- Completeness: Any error found = real error
  - Silence does not guarantee no errors
- Basically any useful analysis
  - is neither sound nor complete (def. not both)
  - … usually leans one way or the other

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

A very quick and short review of compliers!

The Structure of a Compiler



Syntactic Analysis

- Input: sequence of tokens from scanner
- Output: abstract syntax tree
- Actually,
  - parser first builds a parse tree
  - AST is then built by translating the parse tree

Example

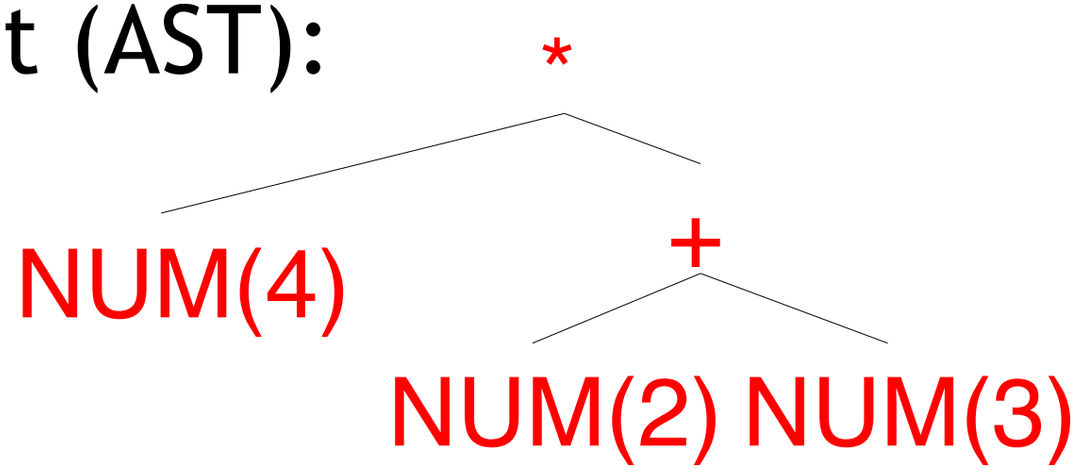
- Source Code

  4*(2+3)

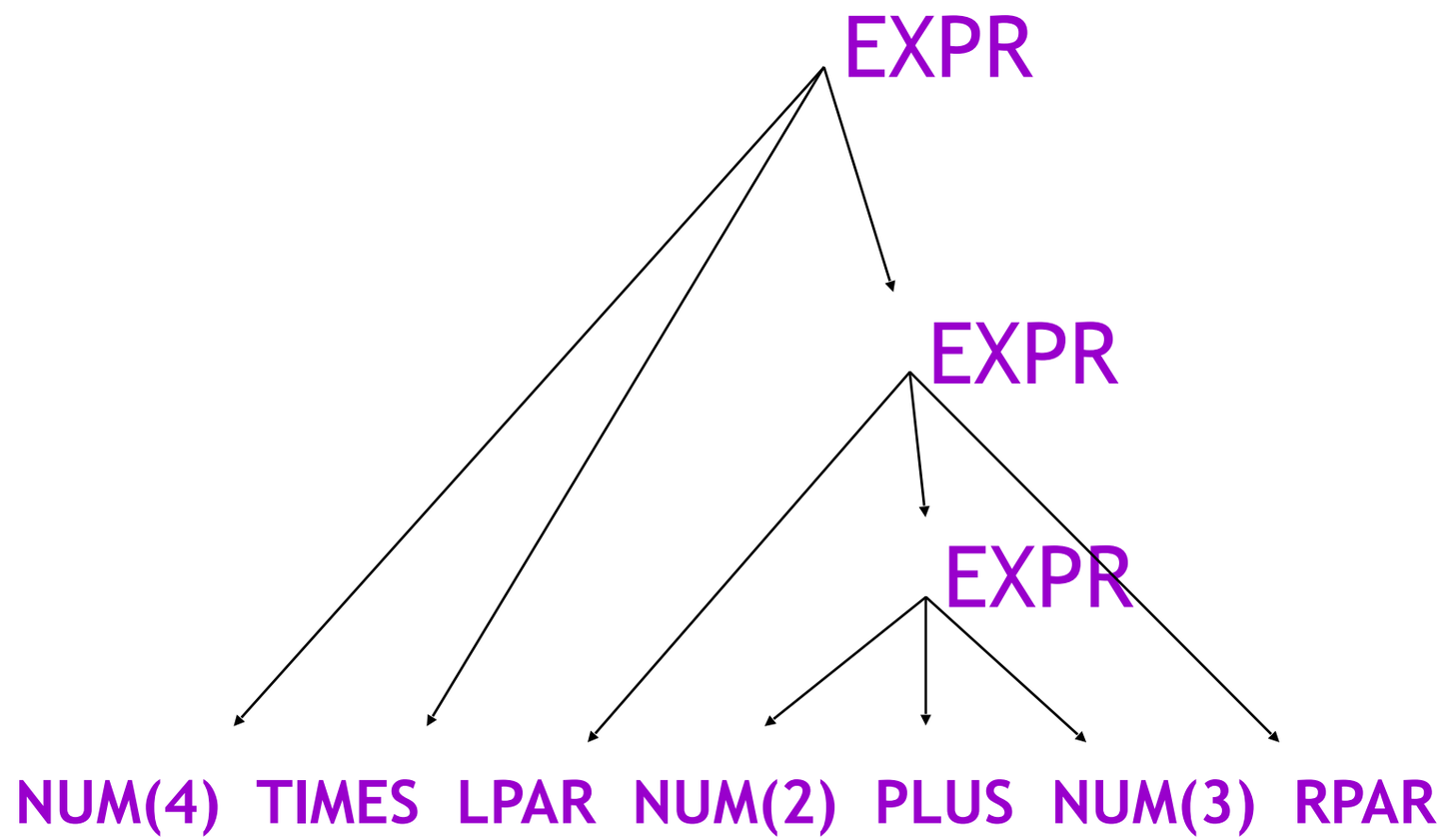
- Parser input

  NUM(4) TIMES LPAR NUM(2) PLUS NUM(3) RPAR

- Parser output (AST):



Parse tree for the example: 4*(2+3)



leaves are tokens

Another example

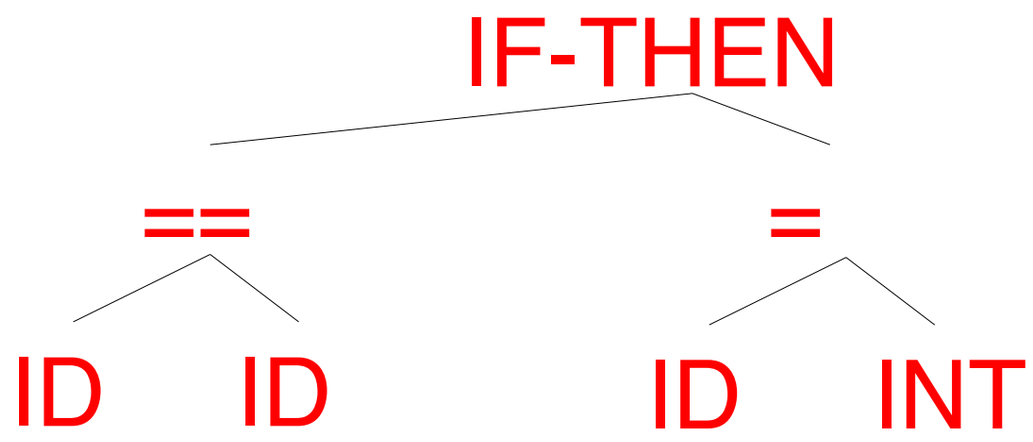
- Source Code

  if (x == y) { a=1; }

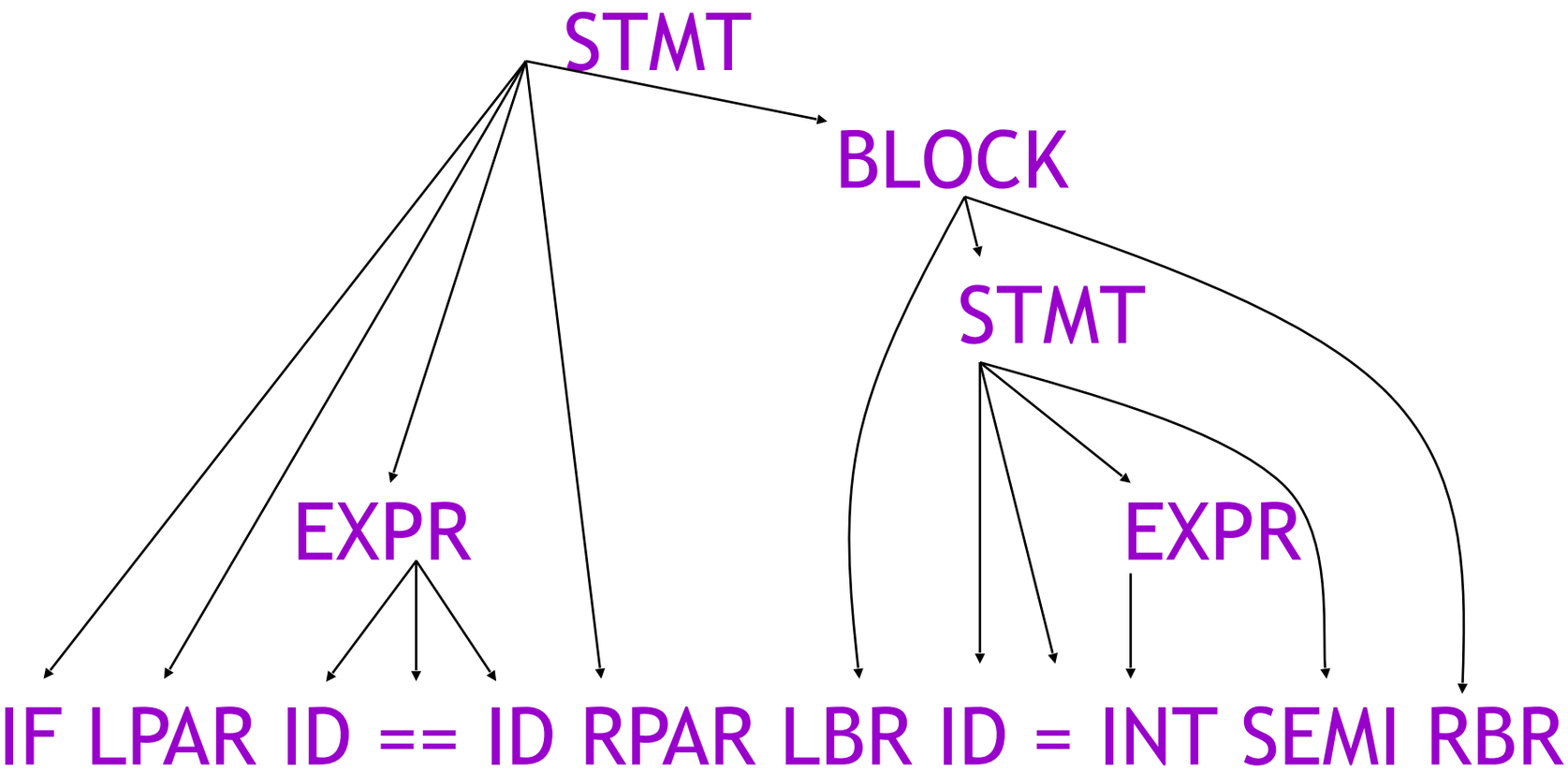
- Parser input

  IF LPAR ID EQ ID RPAR LBR ID AS INT SEMI RBR

- Parser output (AST):



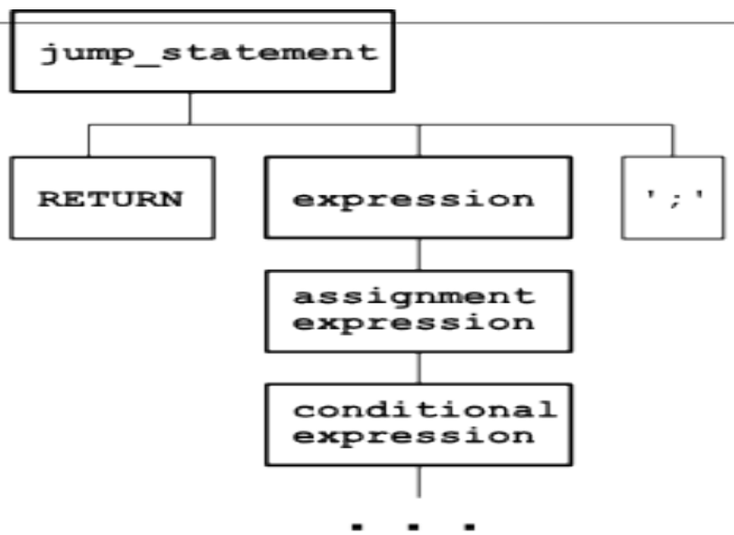
Parse tree for example: if (x==y) {a=1;}



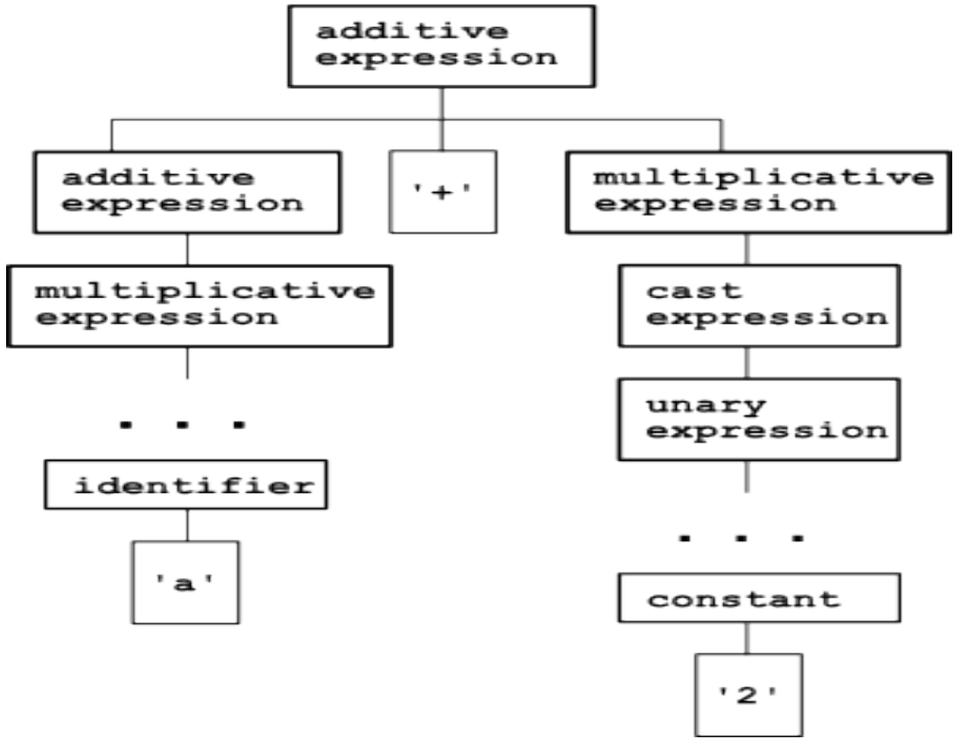
leaves are tokens

Parse Tree

- Representation of grammars in a tree-like form.
- Is a one-to-one mapping from the grammar to a tree-form.



C Statement: return a + 2



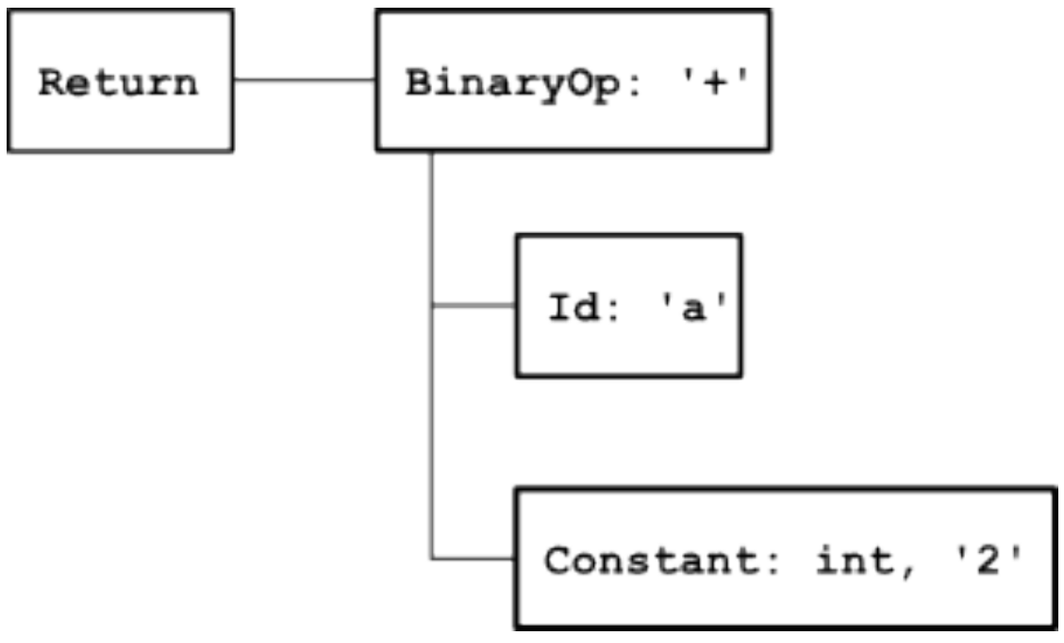
a very formal representation that strictly shows how the parser understands the statement return a + 2;

Abstract Syntax Tree (AST)

- Simplified syntactic representations of the source code, and they're most often expressed by the data structures of the language used for implementation
- Without showing the whole syntactic clutter, represents the parsed string in a structured way, discarding all information that may be important for parsing the string, but isn't needed for analyzing it.

AST

C Statement: return a + 2

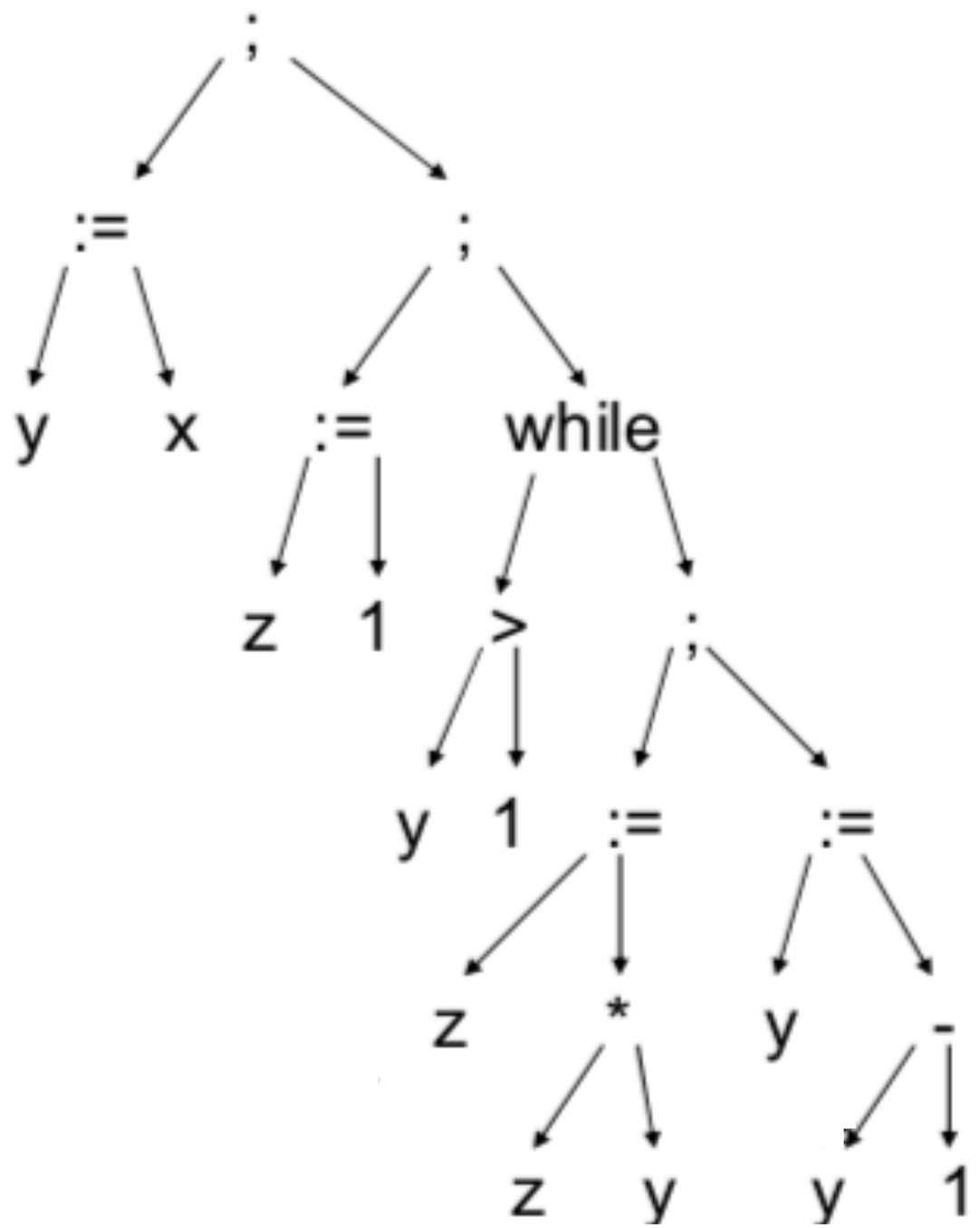


AST Example

- What would be the AST for:
- y := x;
  z := 1;
  while y>1 do

  z := z * y;

  y :=y–1



Matching AST against Bug Patterns

- AST Walker Analysis
  - Walk the AST, looking for nodes of a particular type
  - Check the immediate neighborhood of the node for a bug pattern
  - Warn if the node matches the pattern
- Semantic grep
  - Like grep, looking for simple patterns
  - Unlike grep, consider not just names, but semantic structure of AST

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?

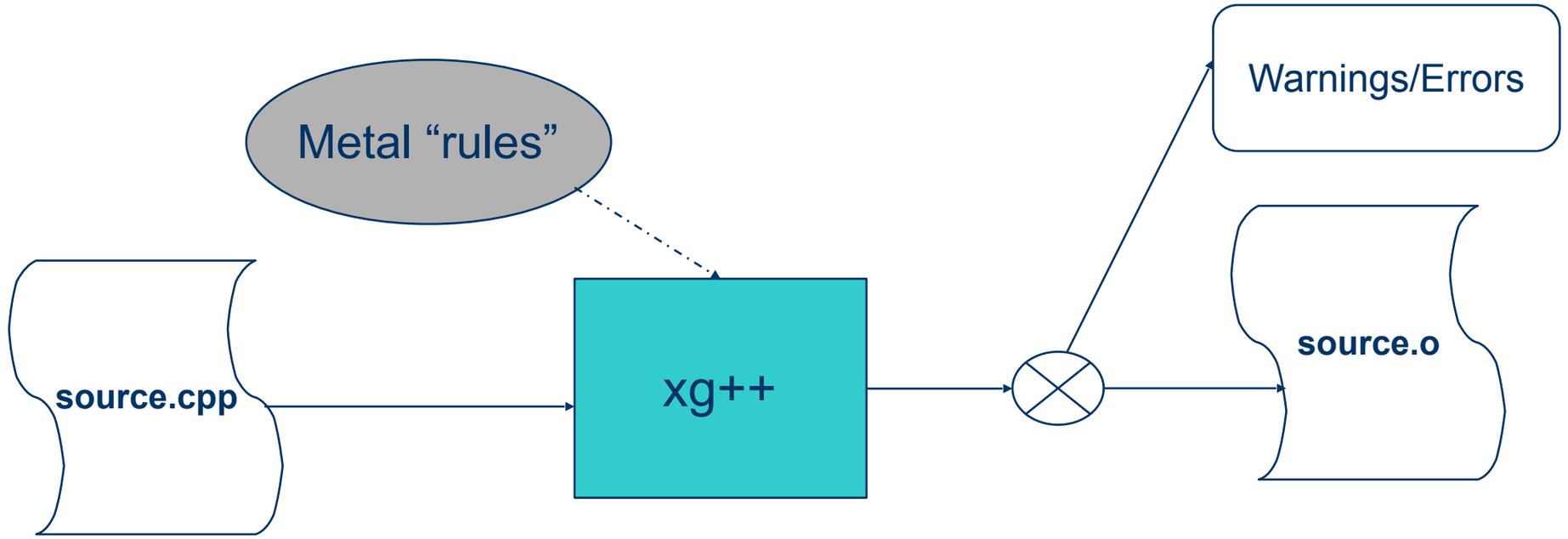
- None of the current methods seem to give us what we're looking for.
- Can the compiler check the code?
  - It would be nice to put the code in the compiler and have it check all of the "rules."
  - Unfortunately, those "rules" are based on semantics of the system that the compiler doesn't understand. (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.

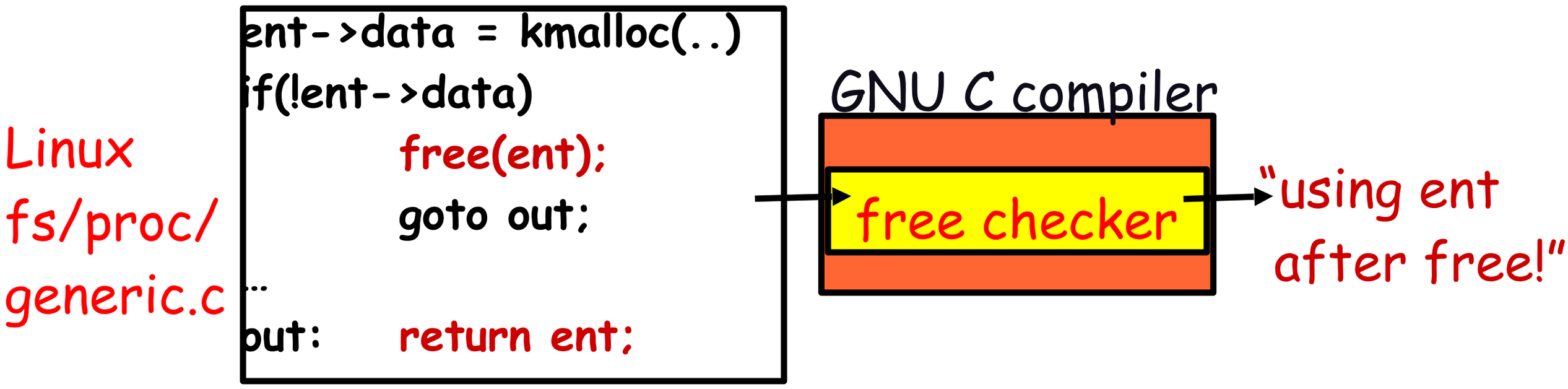


How does it work?

- 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



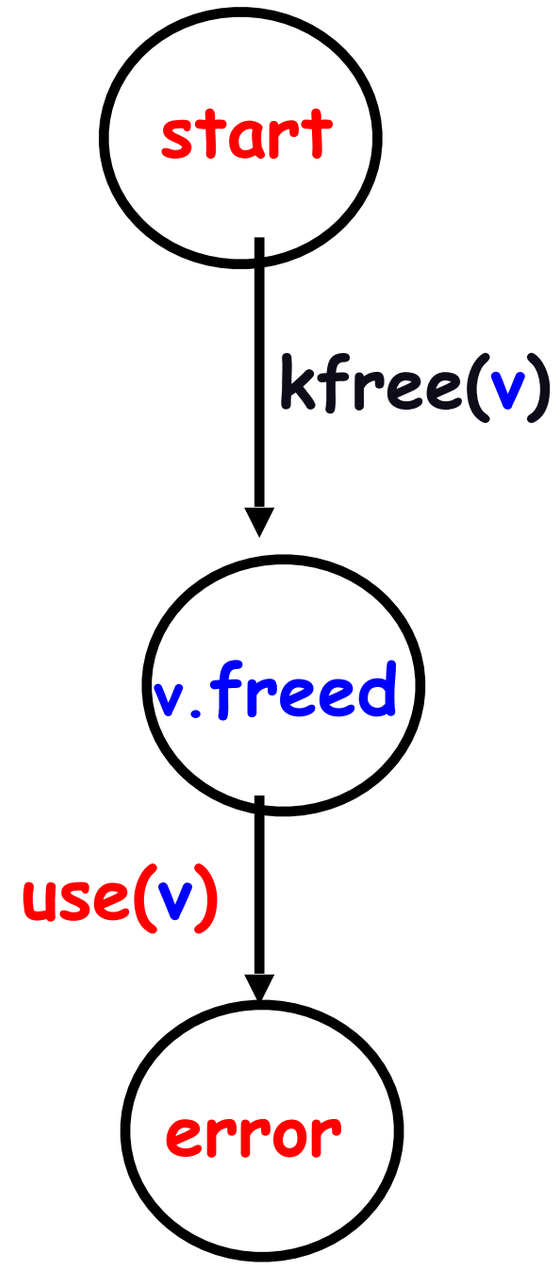
  - Scalable: handles millions of lines of code
  - Precise: says exactly what error was
  - Immediate: finds bugs without having to execute path
  - Effective: 1500+ errors in Linux source code

A bit more detail

```
sm free_checker {
 state decl any_pointer v;
 decl any_pointer x;

 start: { kfree(v); } ==> v.freed;
 v.freed:
   { v != x } || { v == x }
                ==> { /* do nothing */ }
  | { v } ==> { err("Use after free!"); }
   ;
}
```

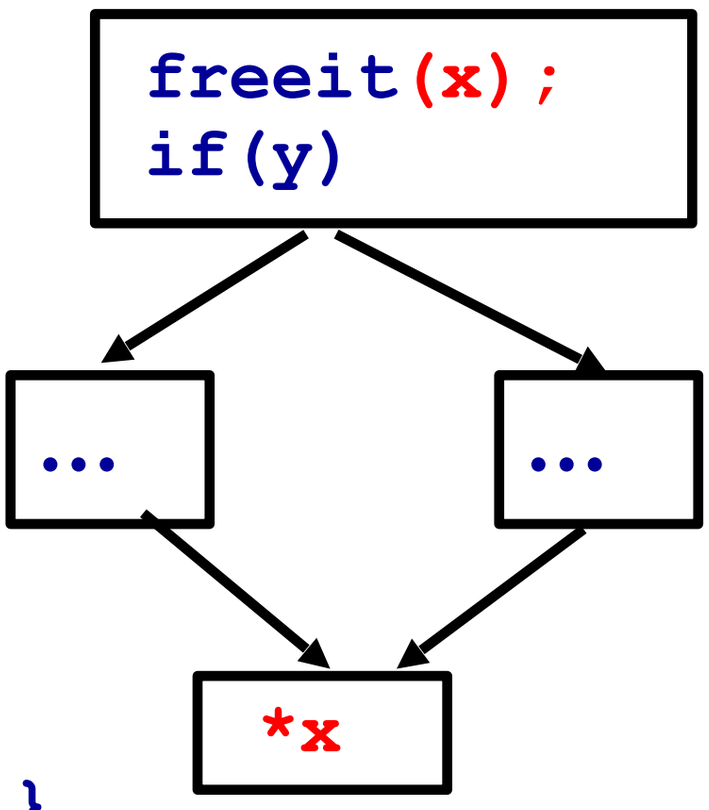
```
/* 2.4.1: fs/proc/generic.c */
ent->data = kmalloc(…)
if(!ent->data) {
      kfree(ent);
      goto out;
…
out: return ent;
```



A quick analysis example

```
freeit(int *z) {
  kfree(z);
}
```

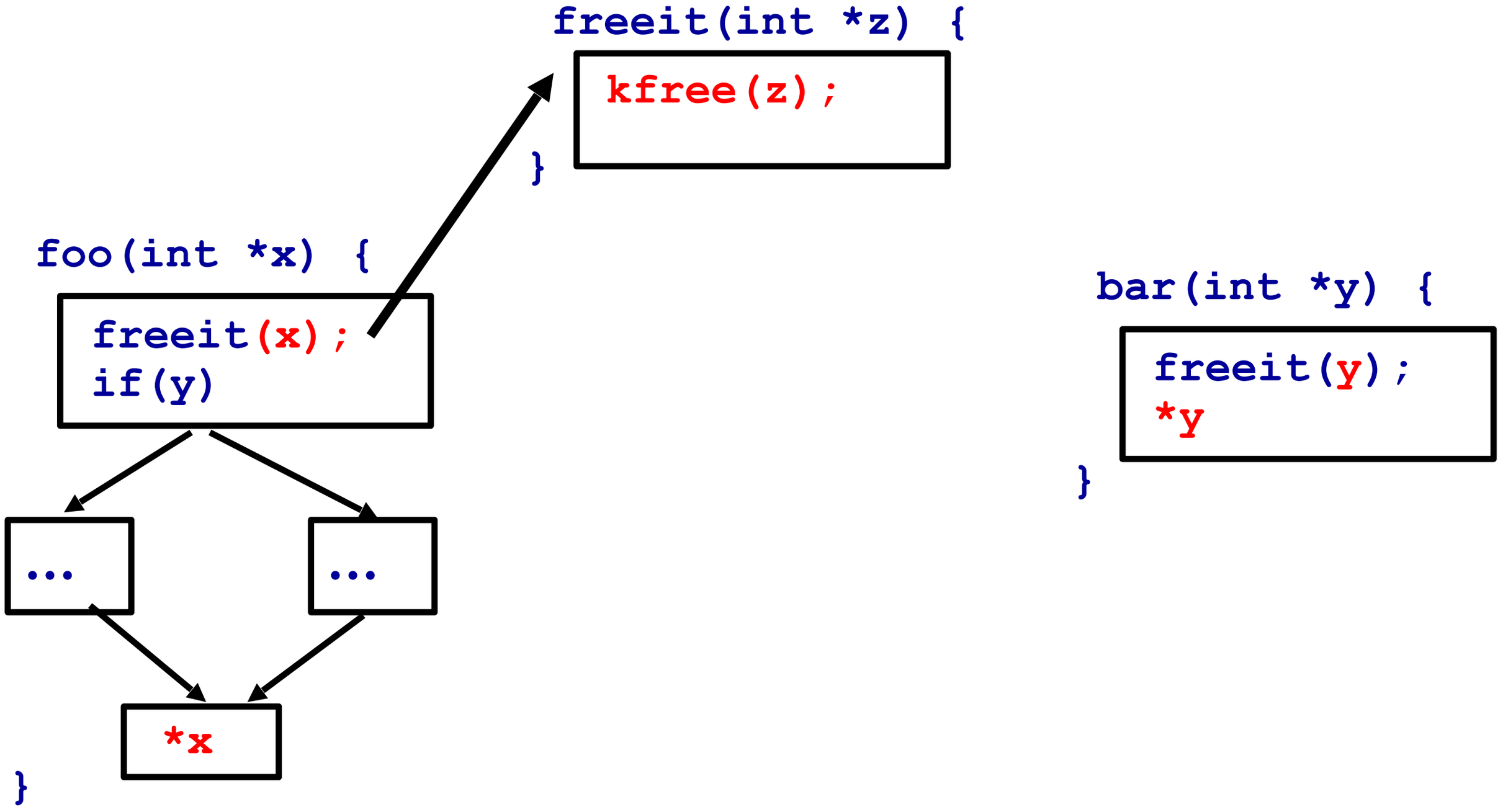
```
foo(int *x) {
  freeit(x);
  if(y)
```



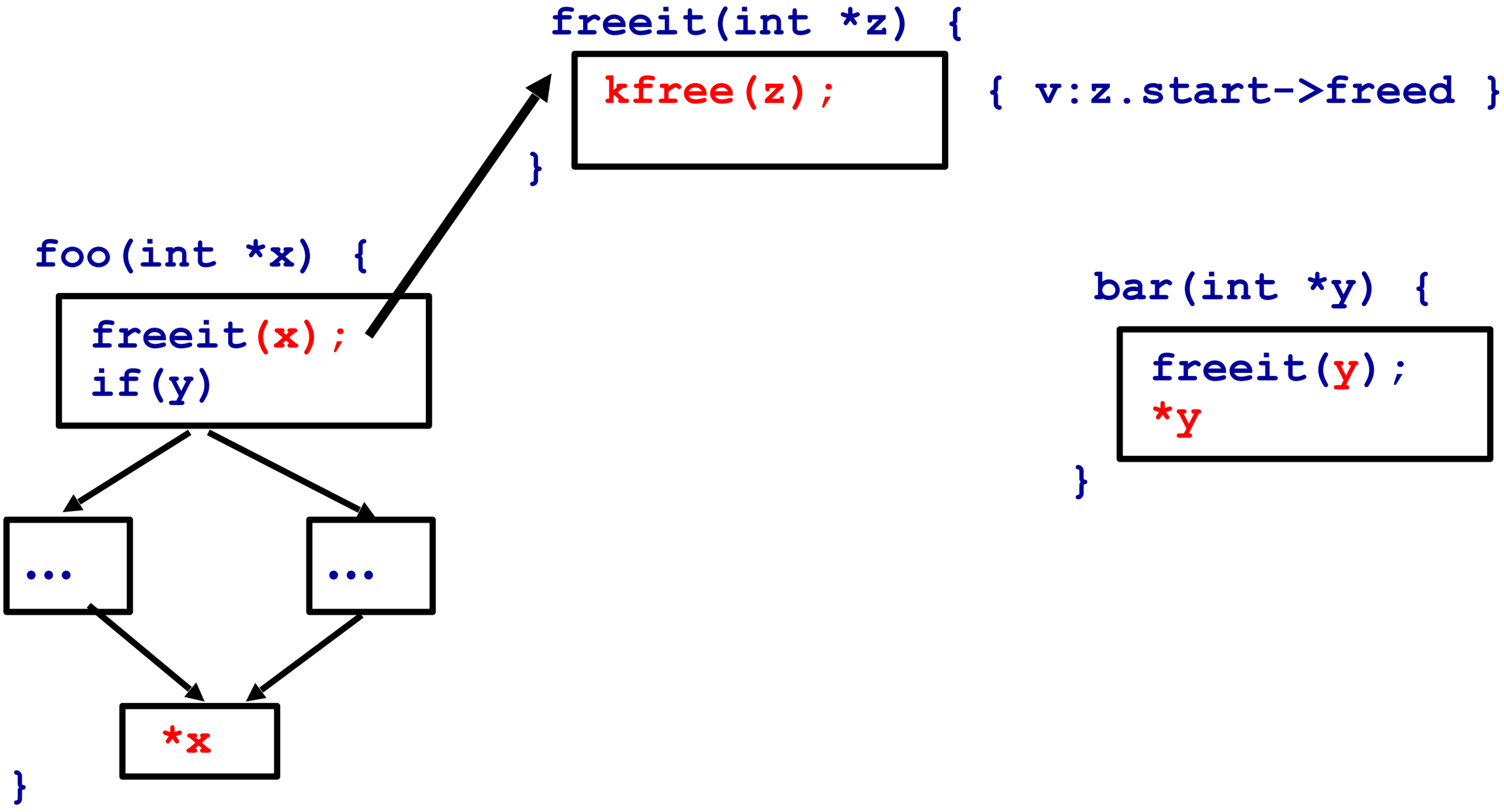

```
}
```

```
bar(int *y) {
  freeit(y);
  *y
}
```

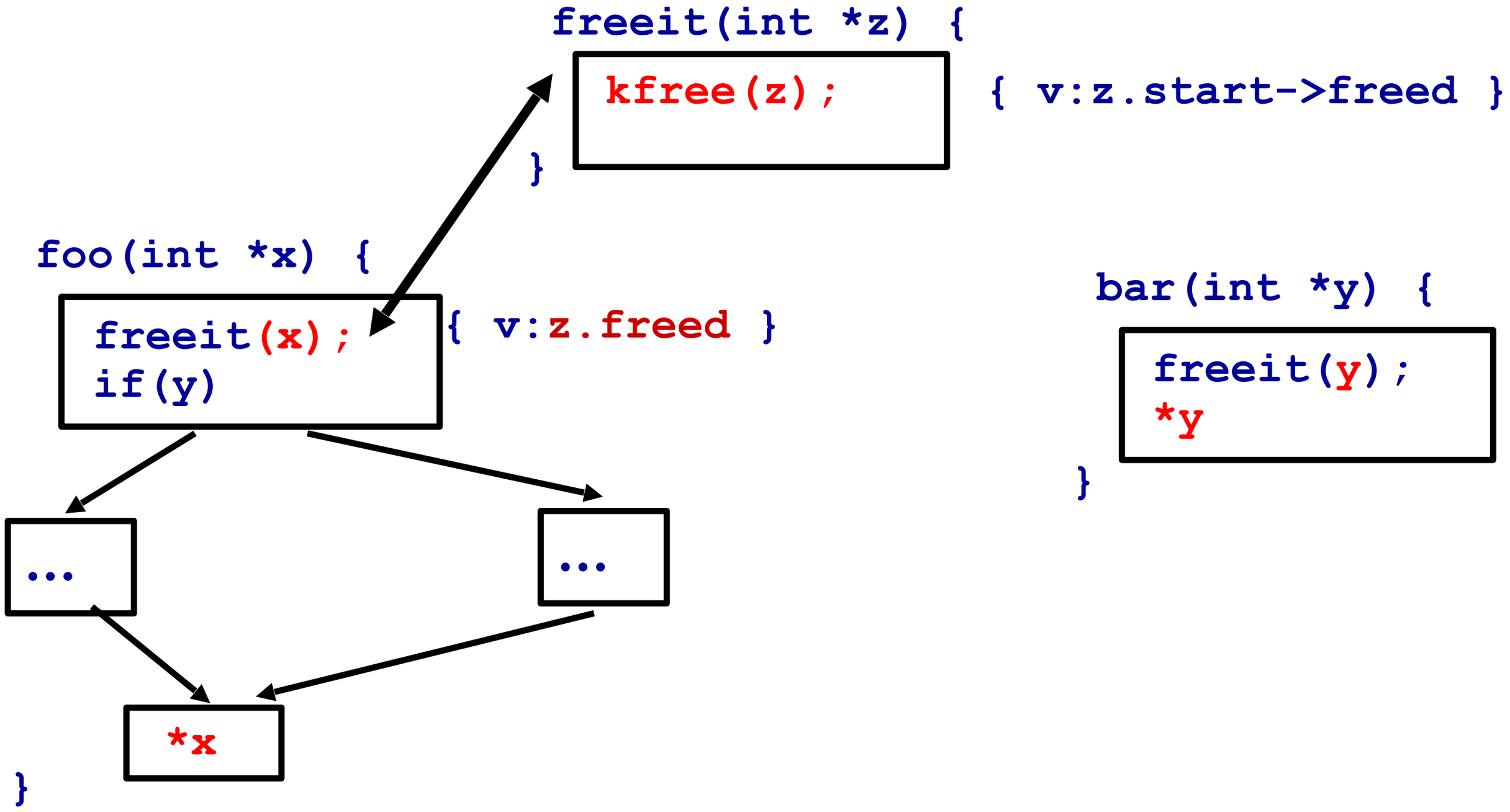
A quick analysis example



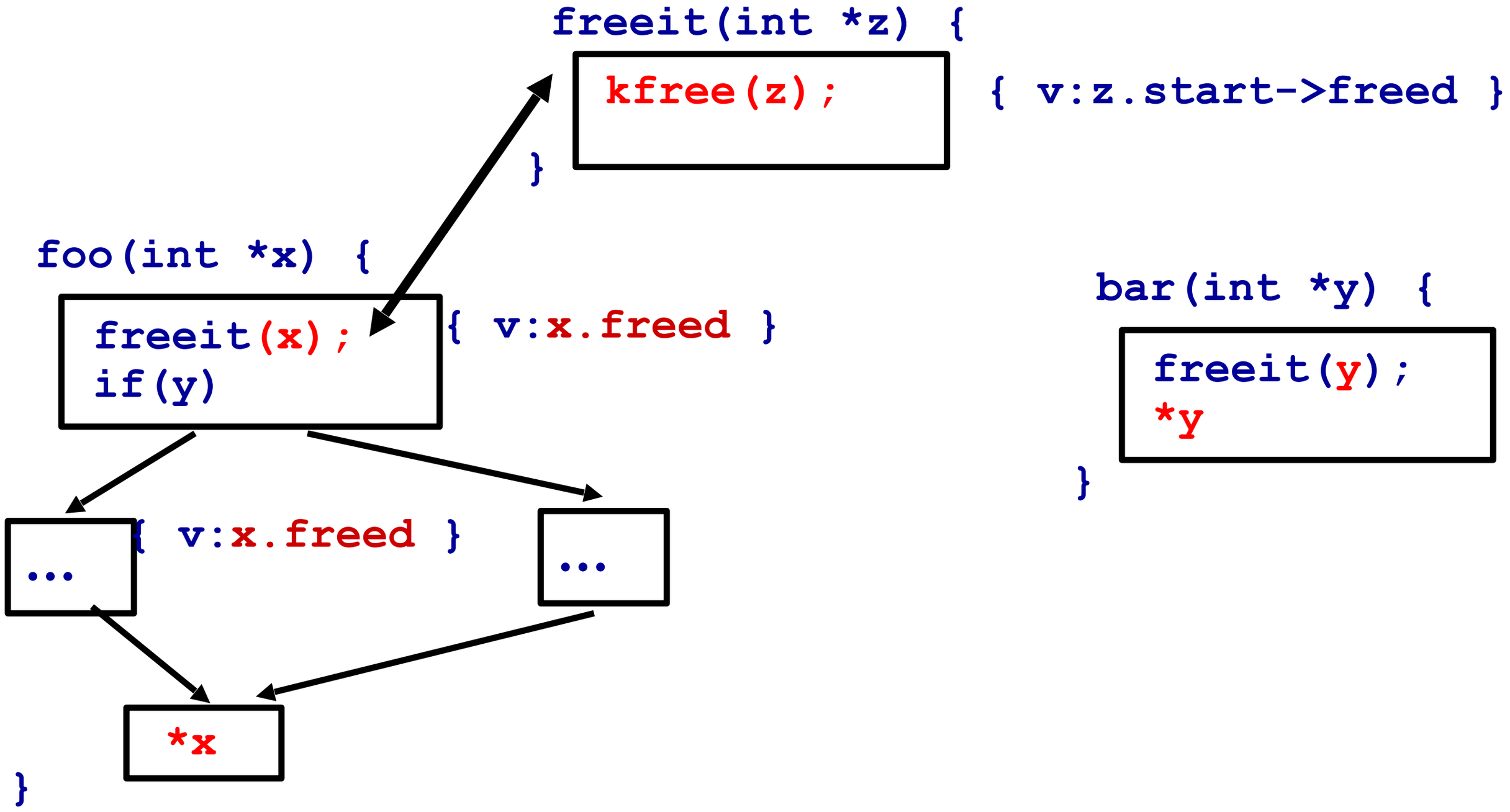
A quick analysis example



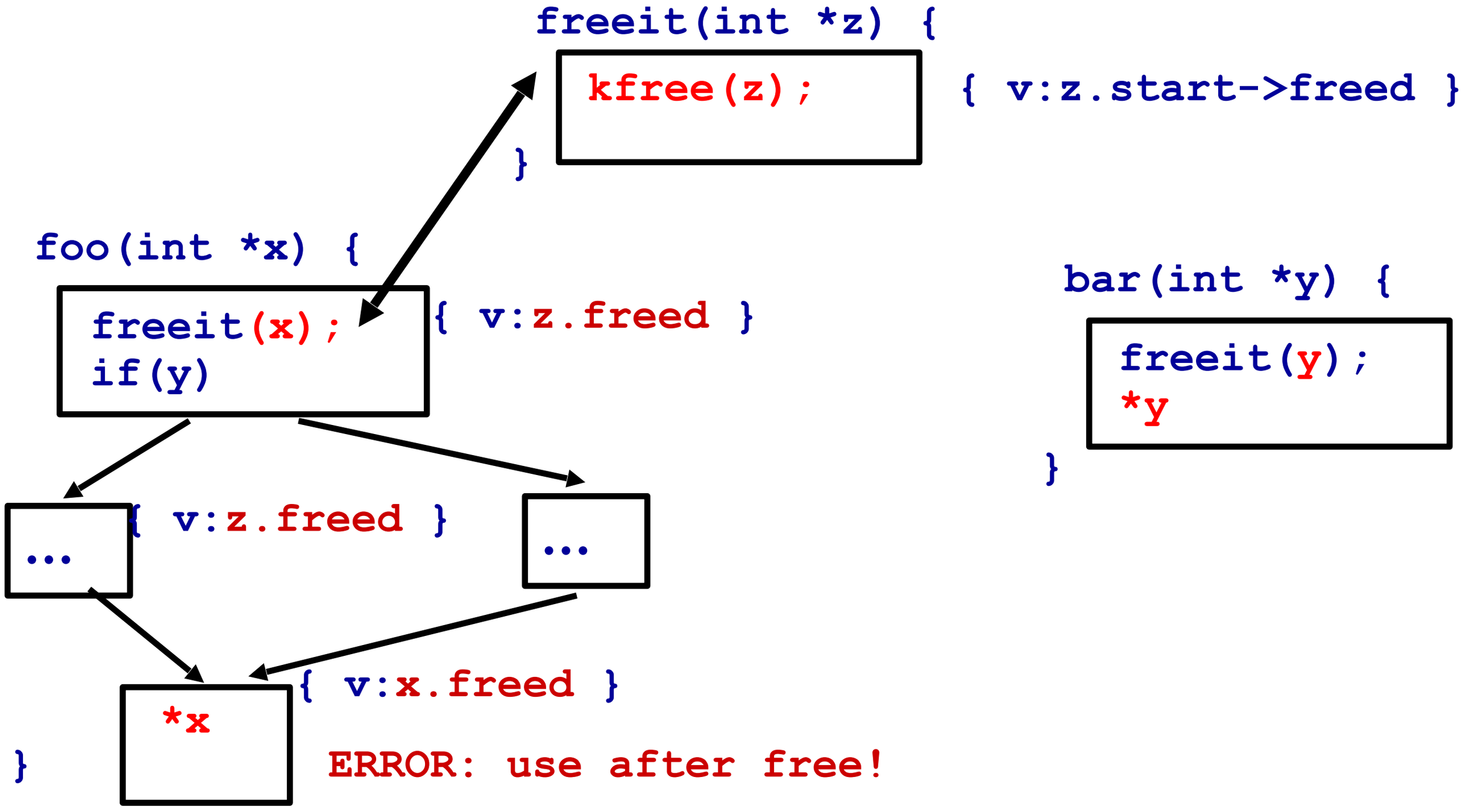
A quick analysis example



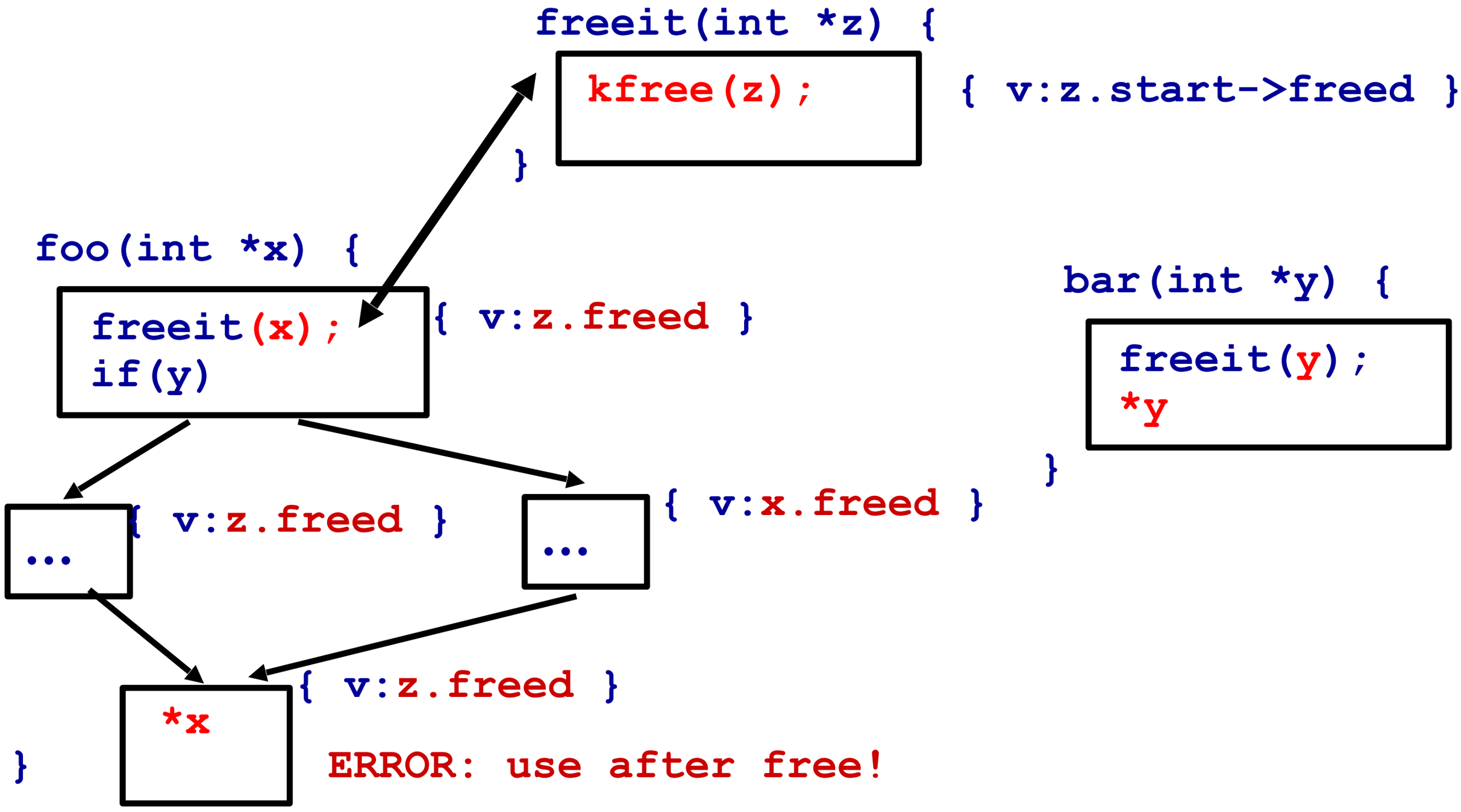
A quick analysis example



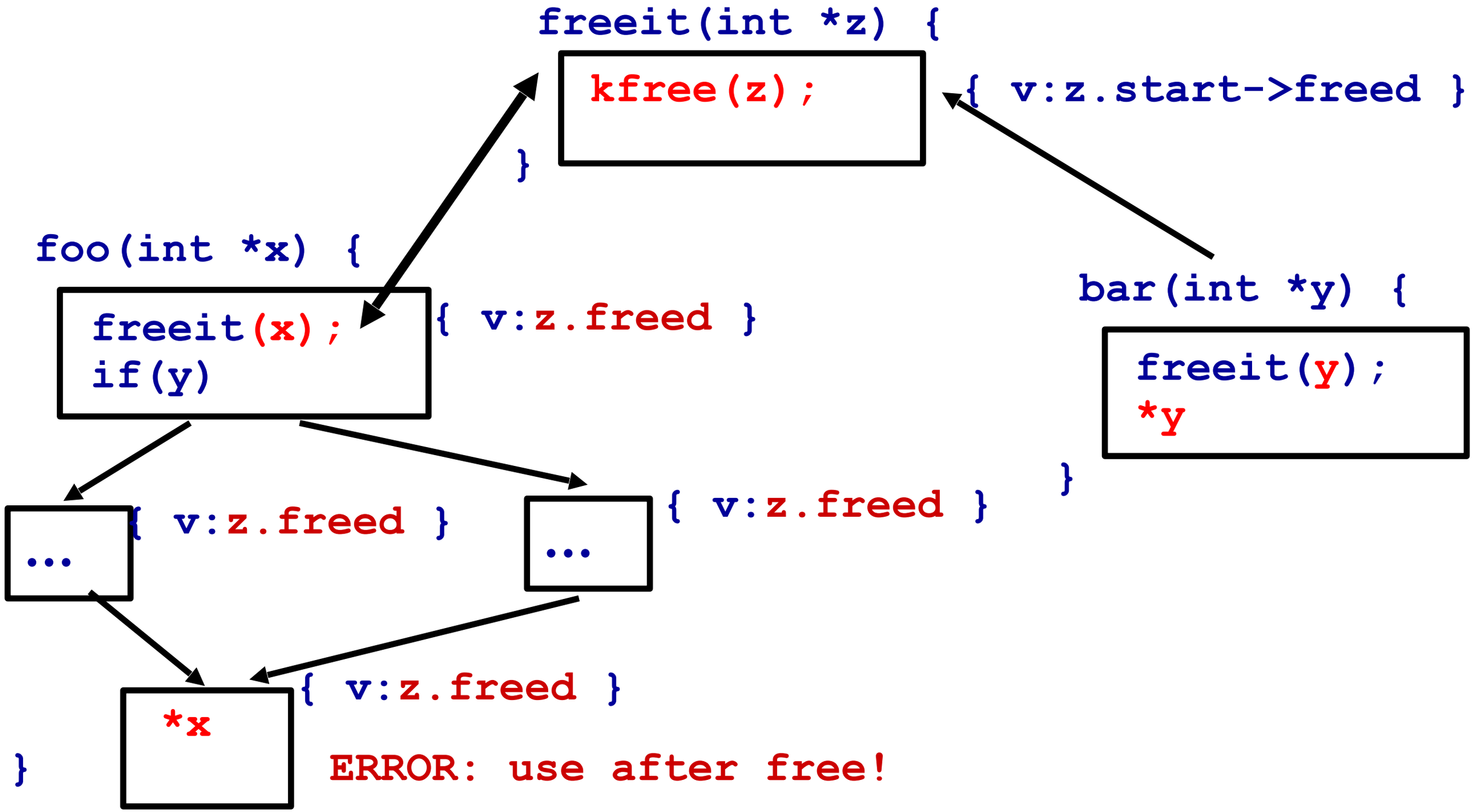
A quick analysis example



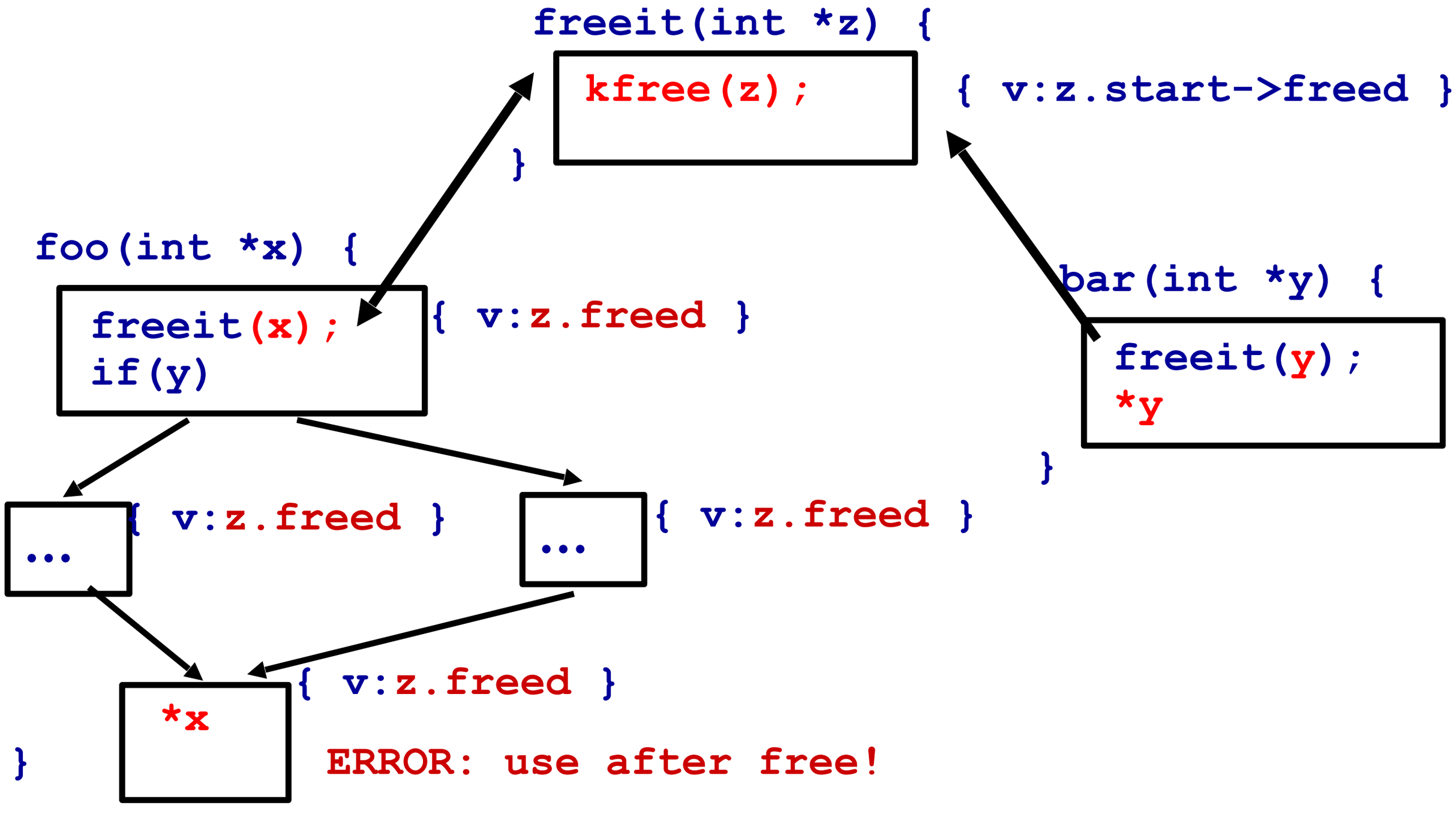
A quick analysis example



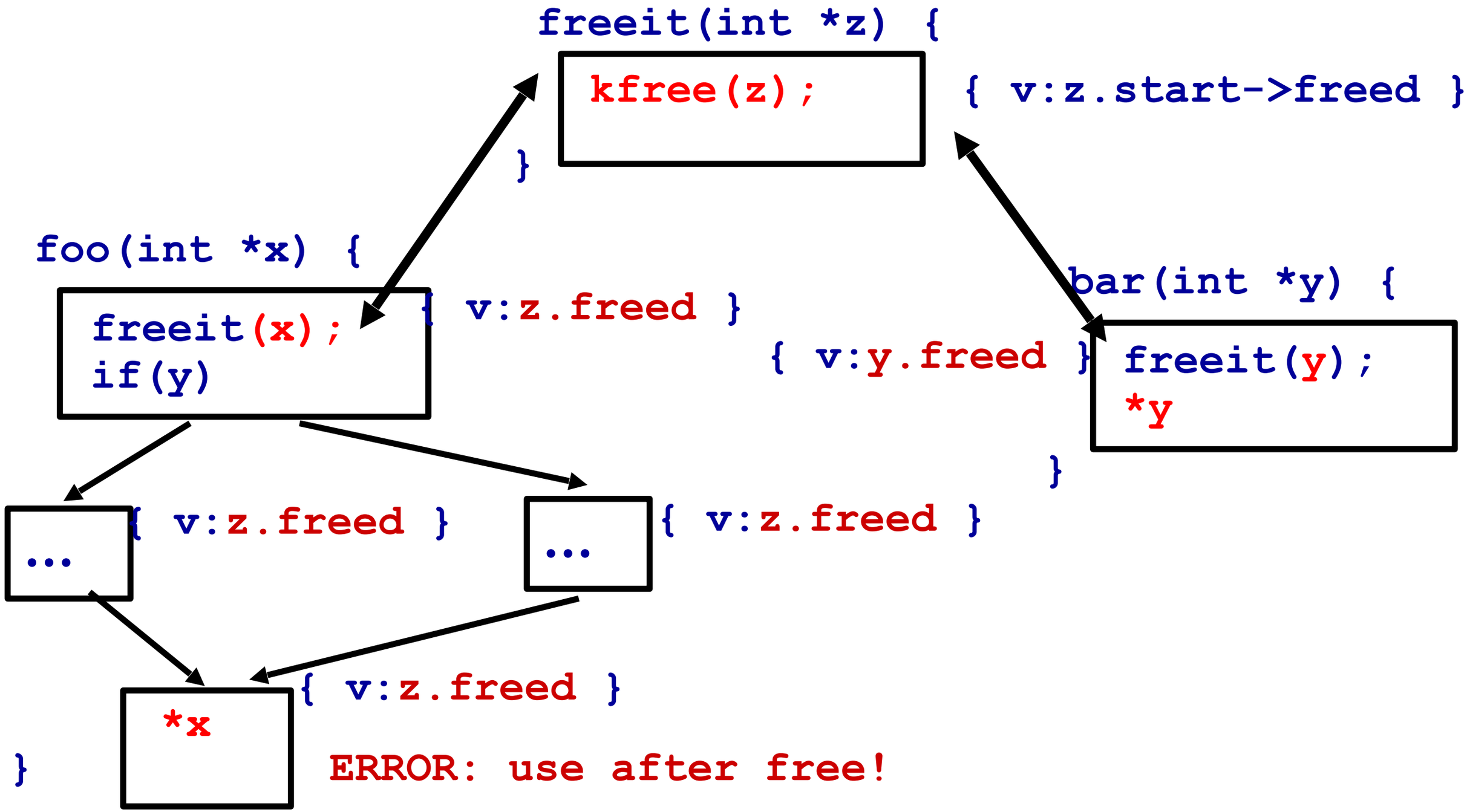
A quick analysis example



A quick analysis example



A quick analysis example



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 */...);

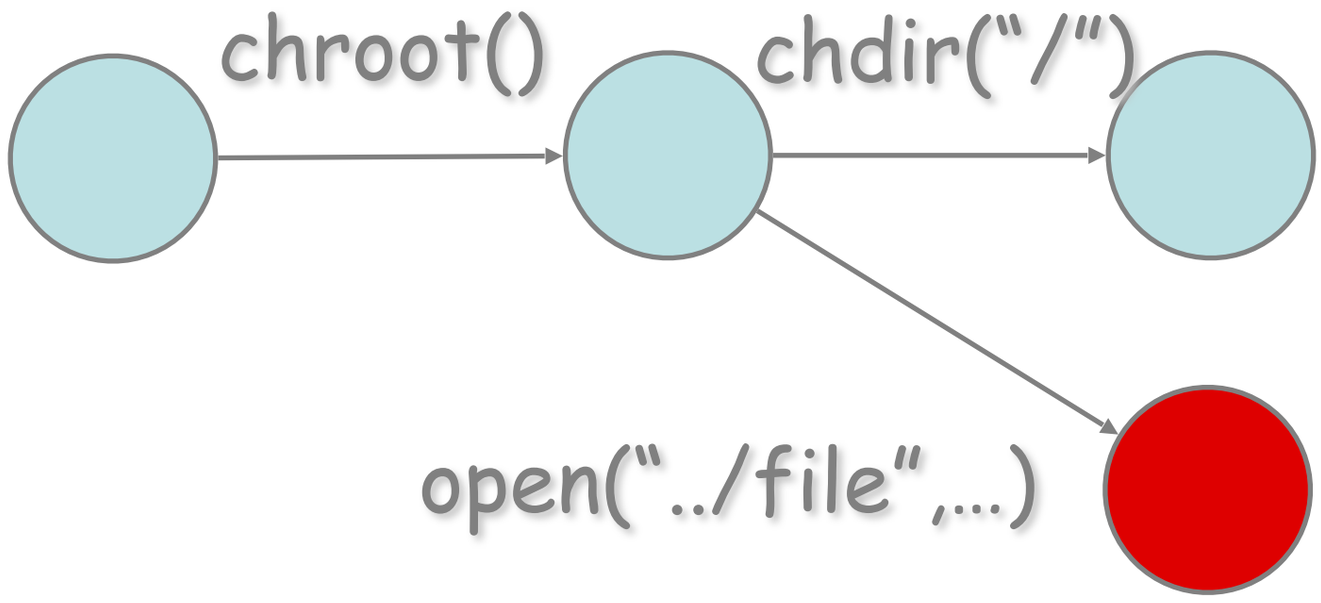
- Typical mistake:
  - fd = open(“file”, O_CREAT);

- Result: file has random permissions

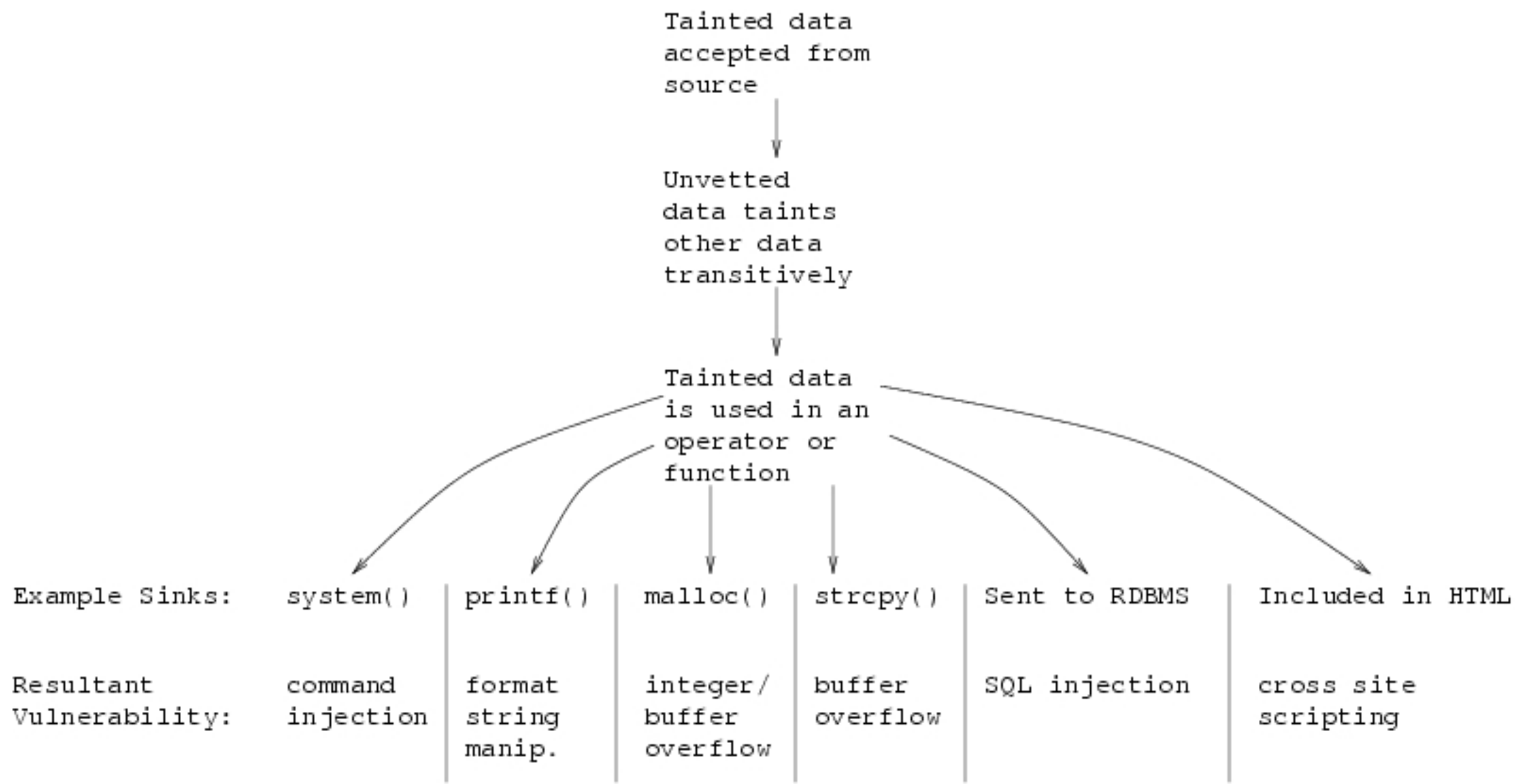
- Check: Look for oflags == O_CREAT without mode argument

Example: Chroot protocol checker

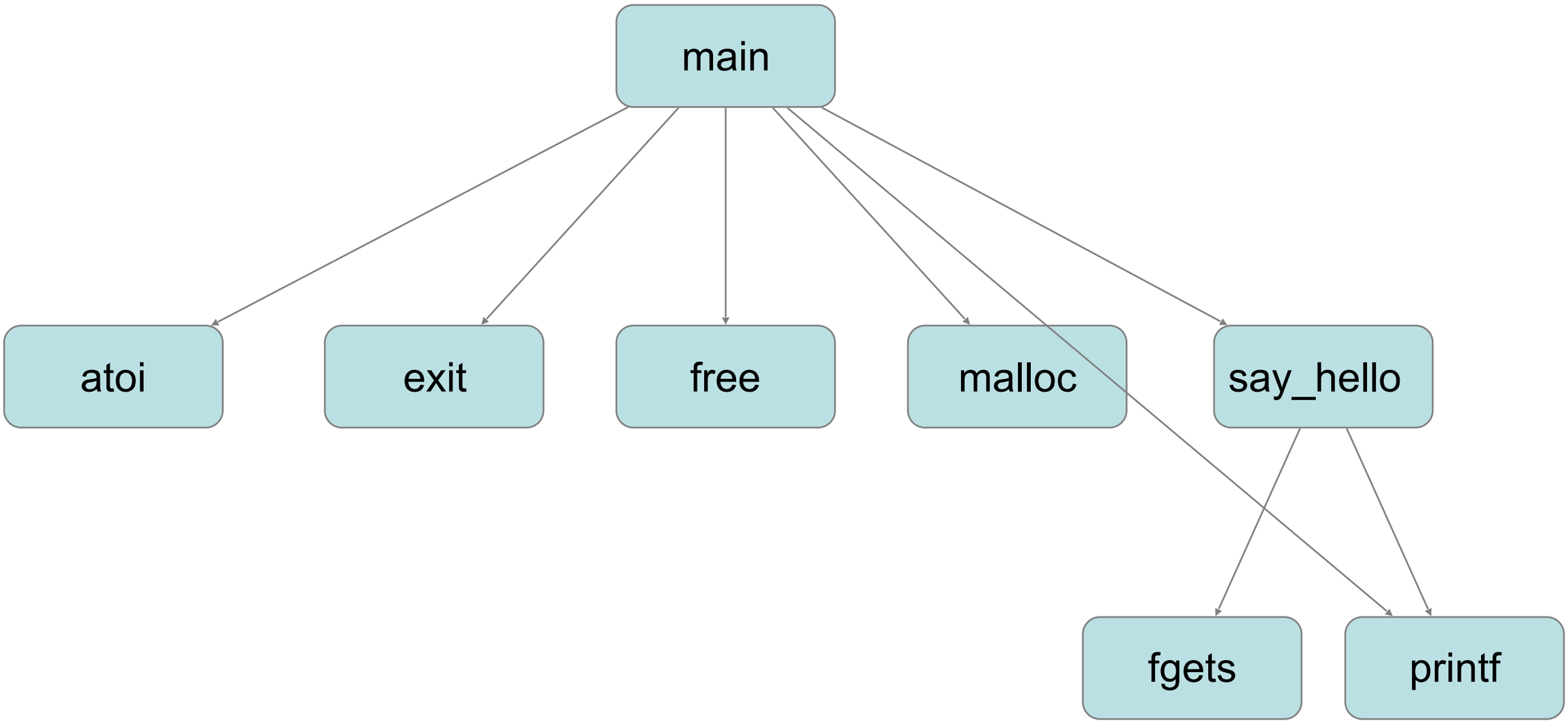
- Goal: confine process to a “jail” on the filesystem
  - chroot() changes filesystem root for a process
- Problem
  - chroot() itself does not change current working directory



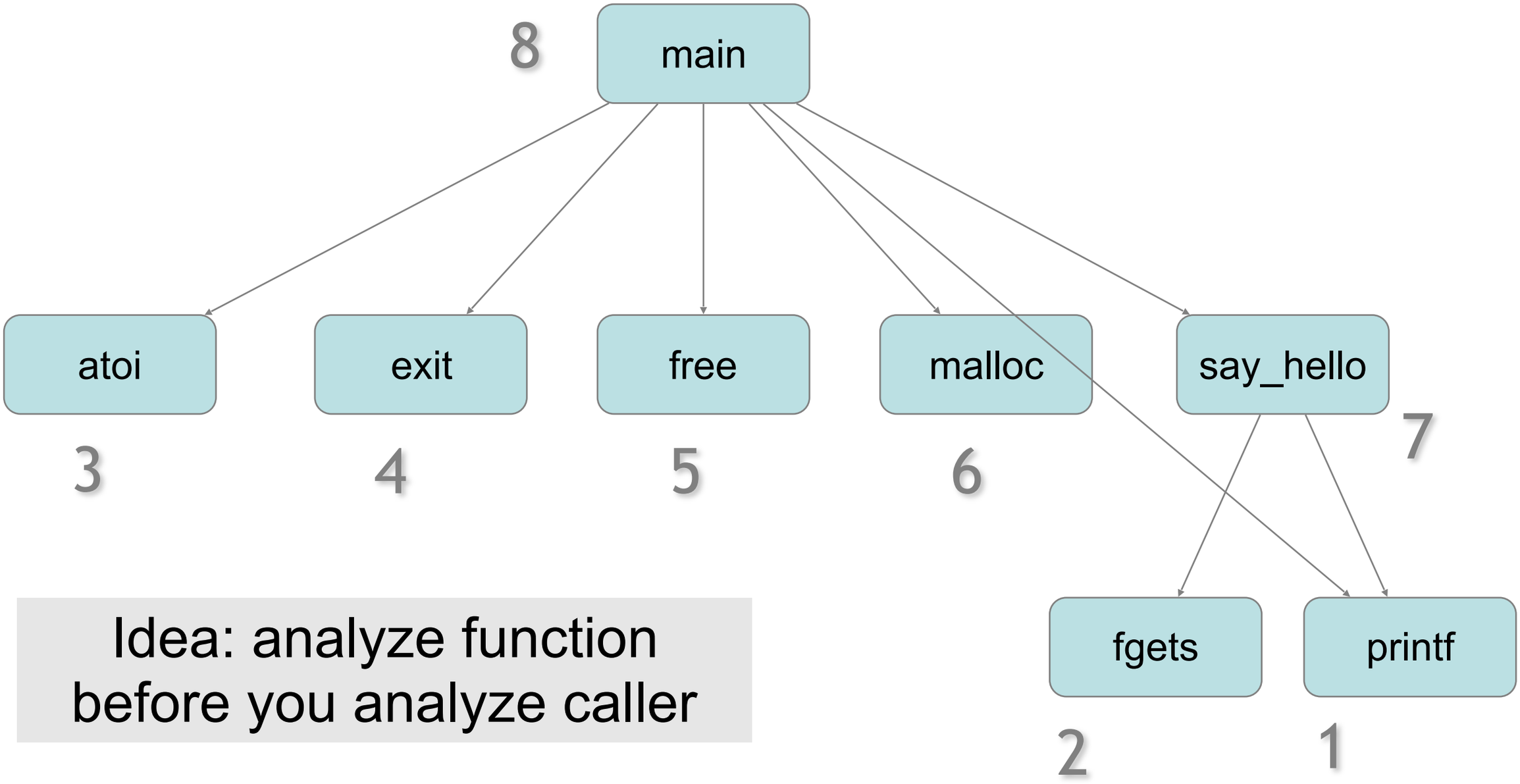
Tainting checkers



Callgraph

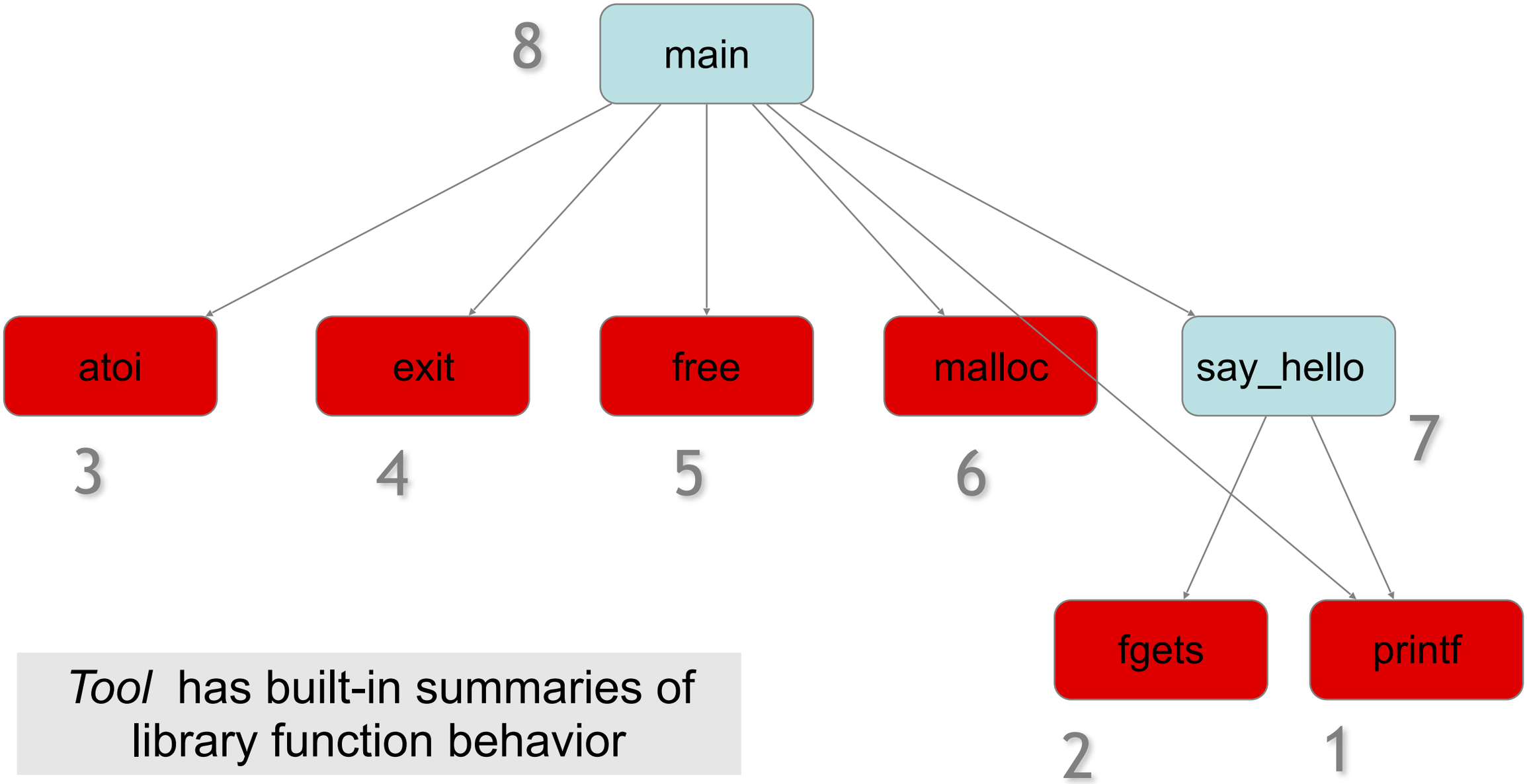


Reverse Topological Sort



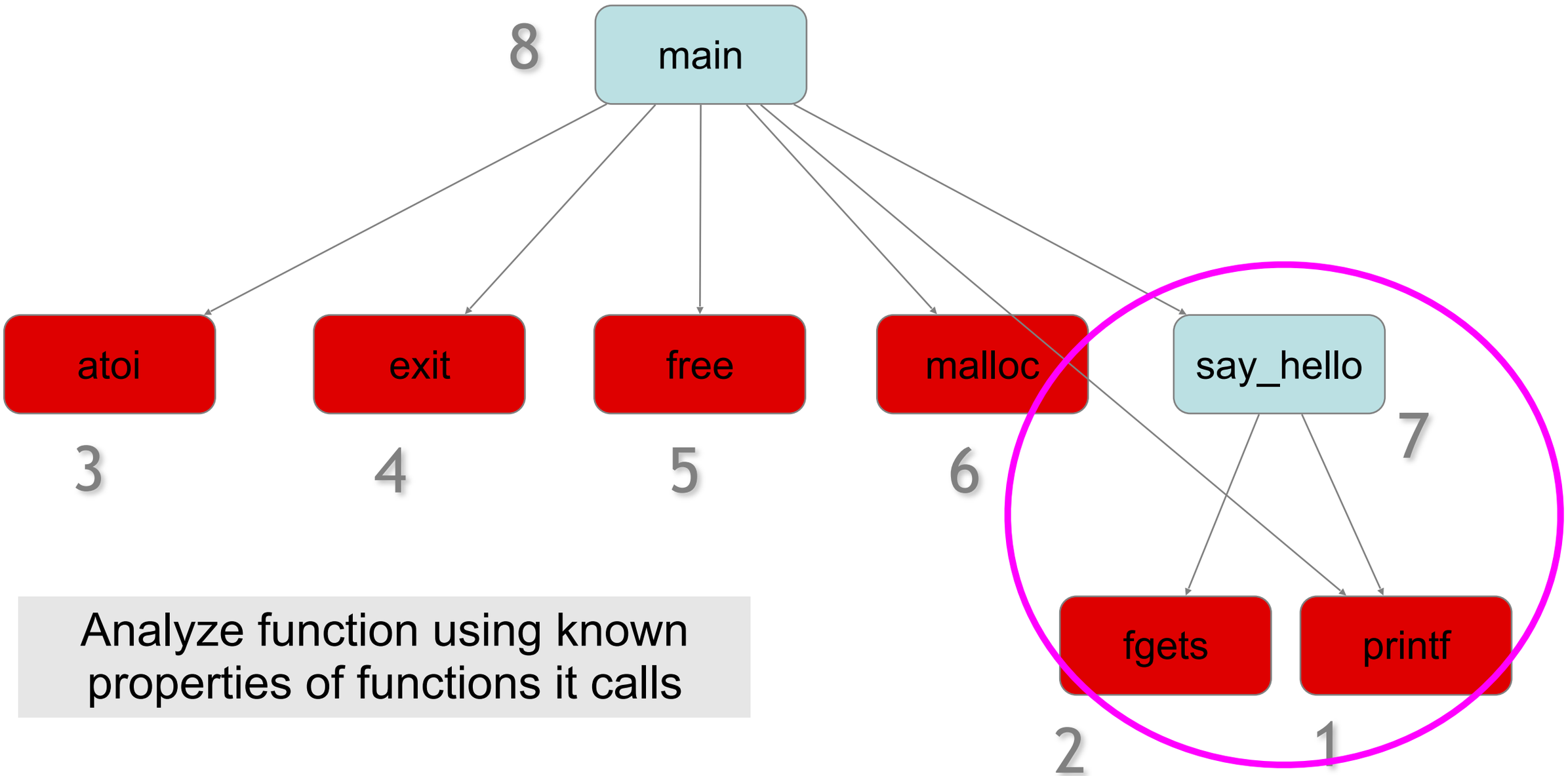
Idea: analyze function before you analyze caller

Apply Library Models



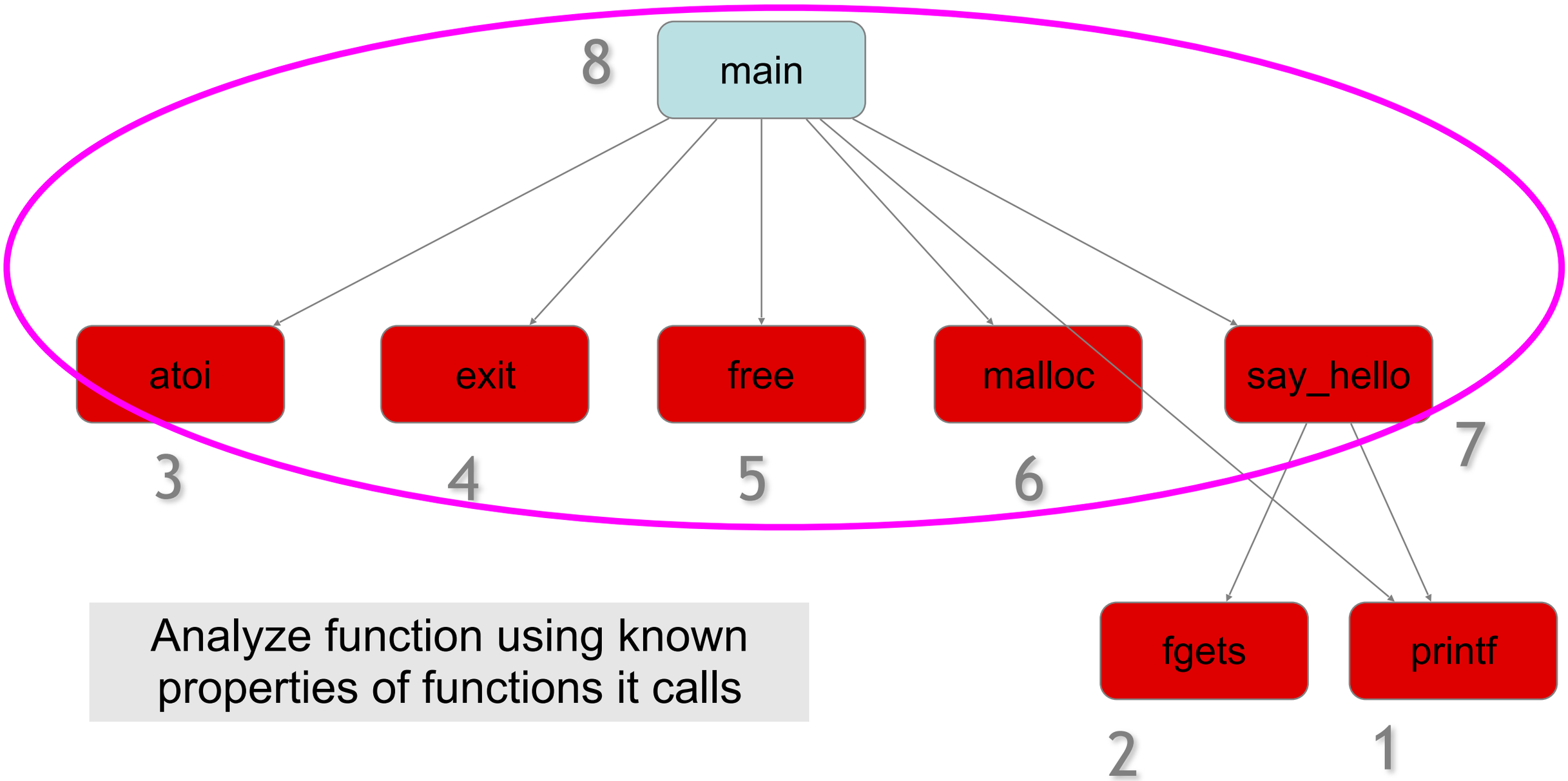
Tool has built-in summaries of library function behavior

Bottom Up Analysis



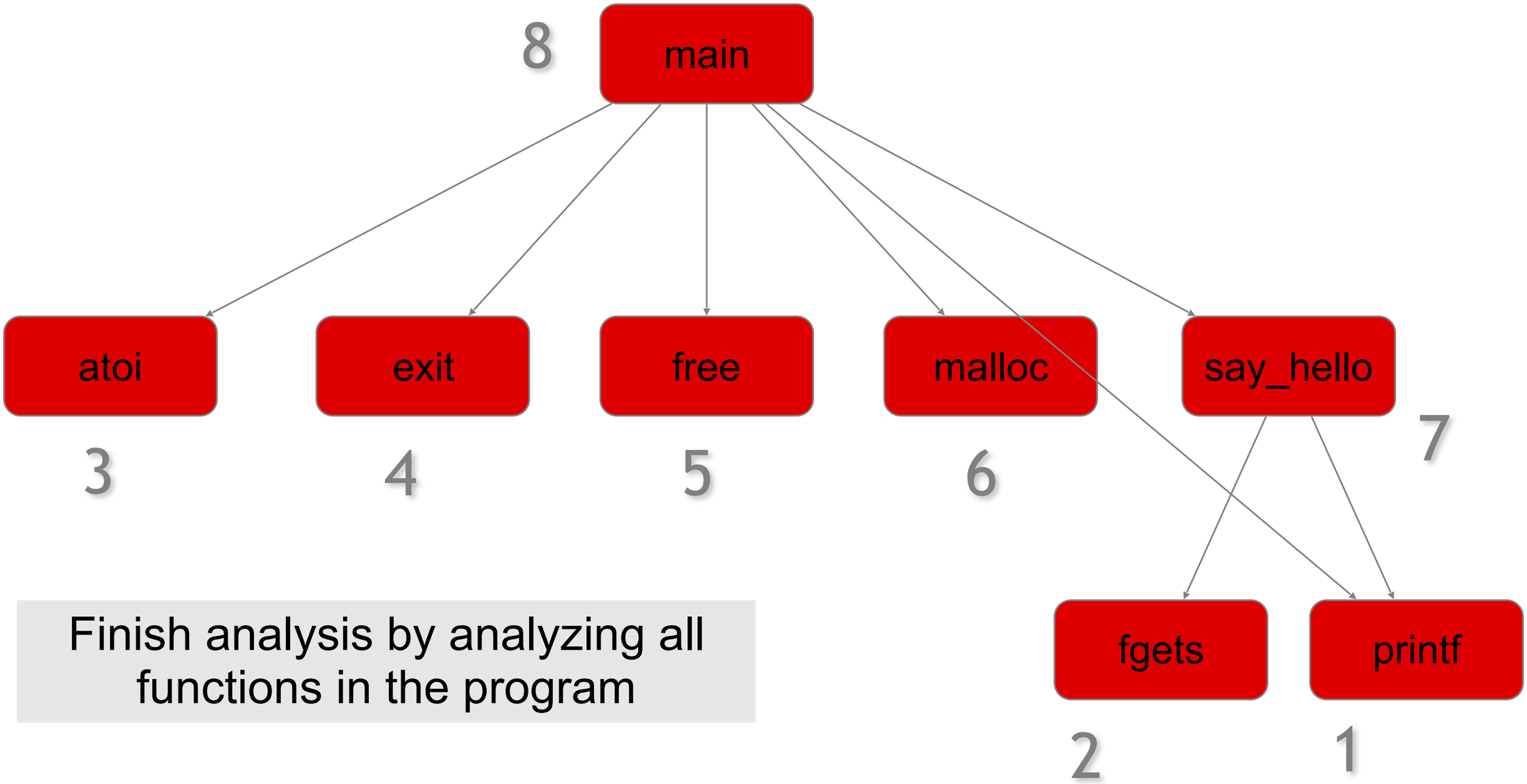
Analyze function using known properties of functions it calls

Bottom Up Analysis



Analyze function using known properties of functions it calls

Bottom Up Analysis

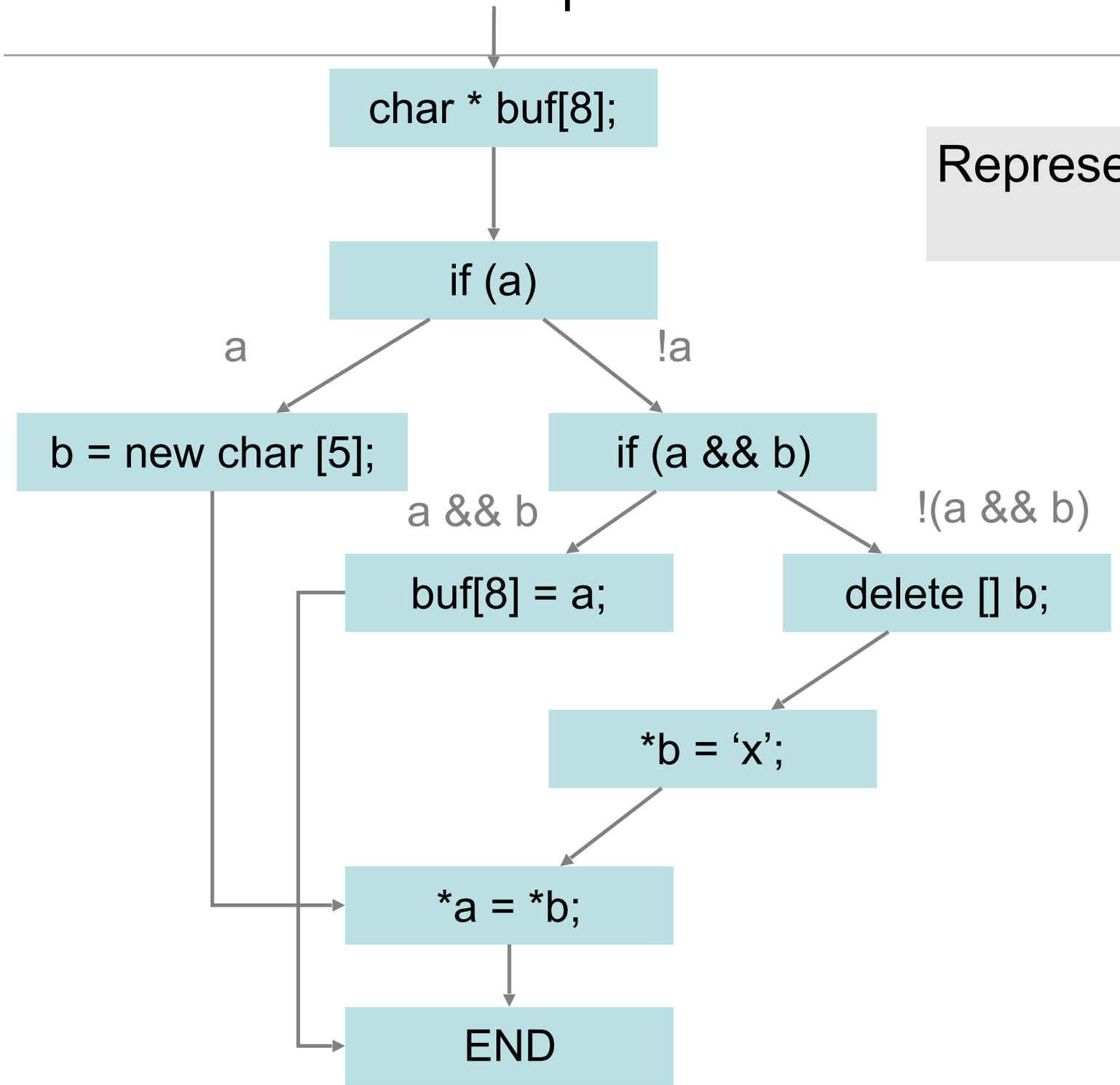


Finish analysis by analyzing all functions in the program

Finding Local Bugs

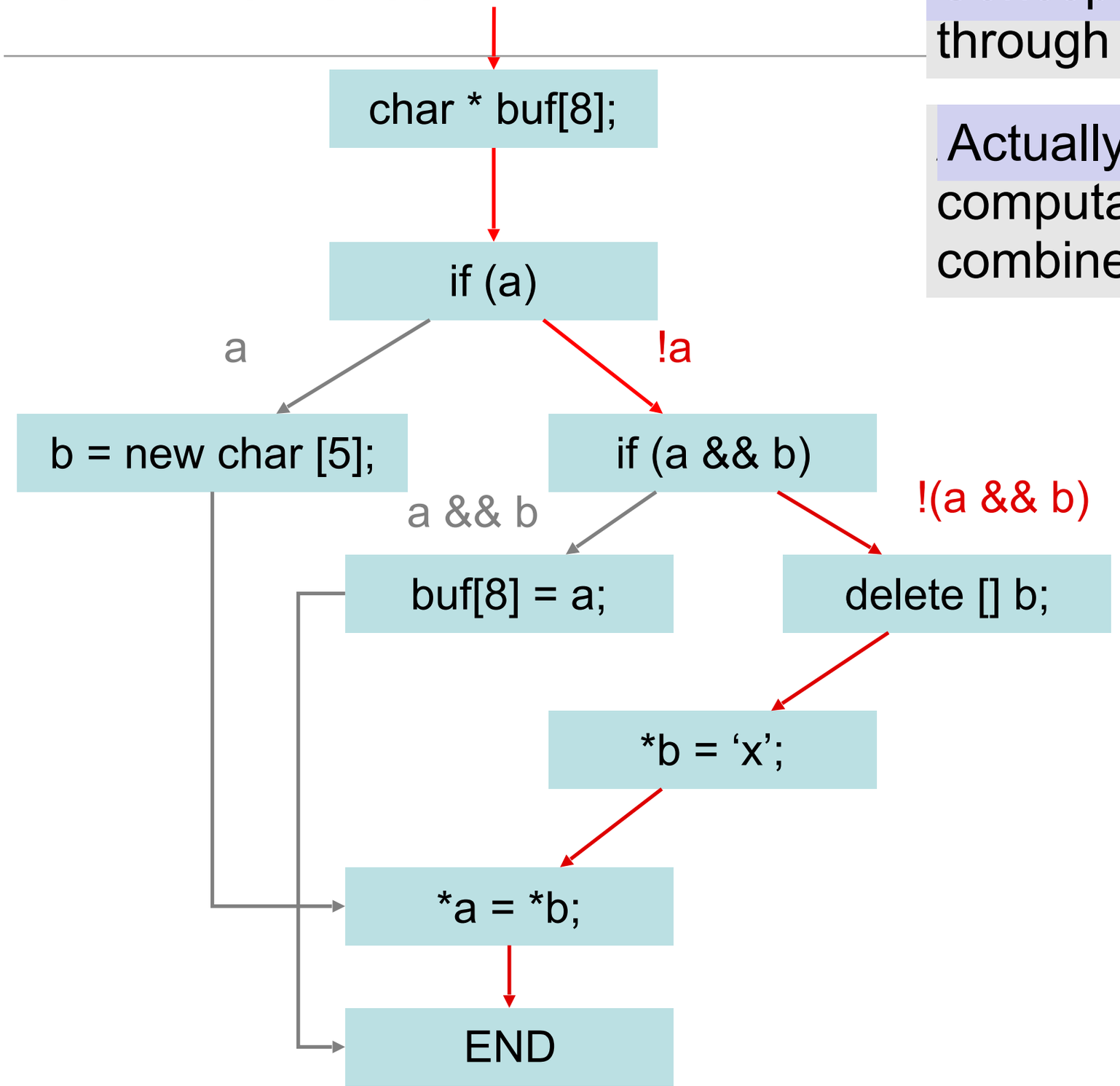
```
#define SIZE 8
void set_a_b(char * a, char * b) {
   char * buf[SIZE];
   if (a) {
      b = new char[5];
   } else {
      if (a && b) {
         buf[SIZE] = a;
         return;
      } else {
         delete [] b;
      }
      *b = 'x';
   }
   *a = *b;
}
```

Control Flow Graph



Represent logical structure of code in graph form

Path Traversal

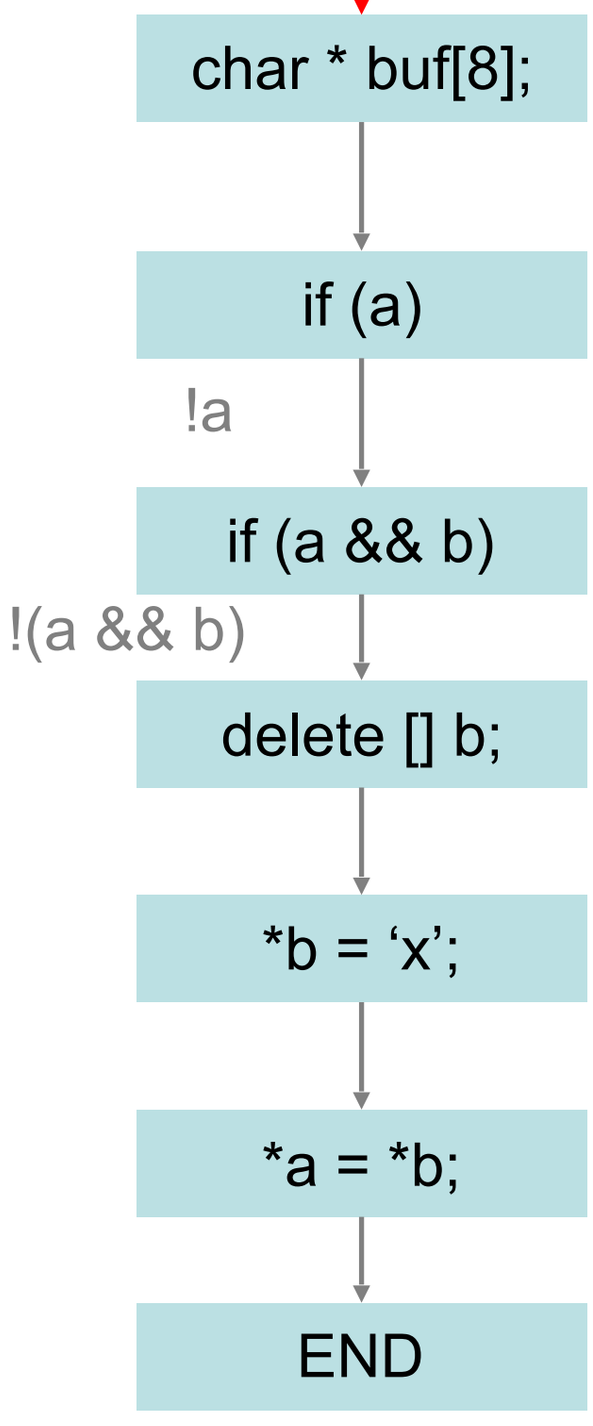


Conceptually Analyze each path through control graph separately

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

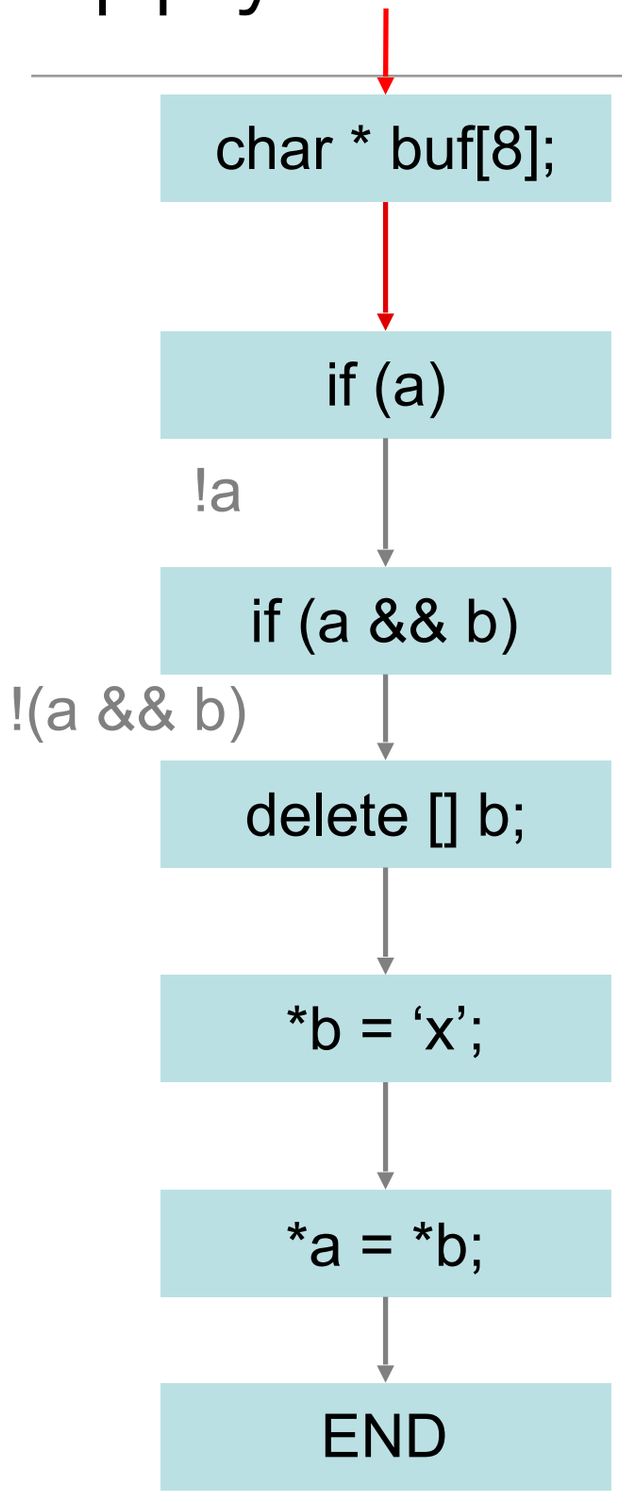
- Defined by a state diagram, with state transitions and error states

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

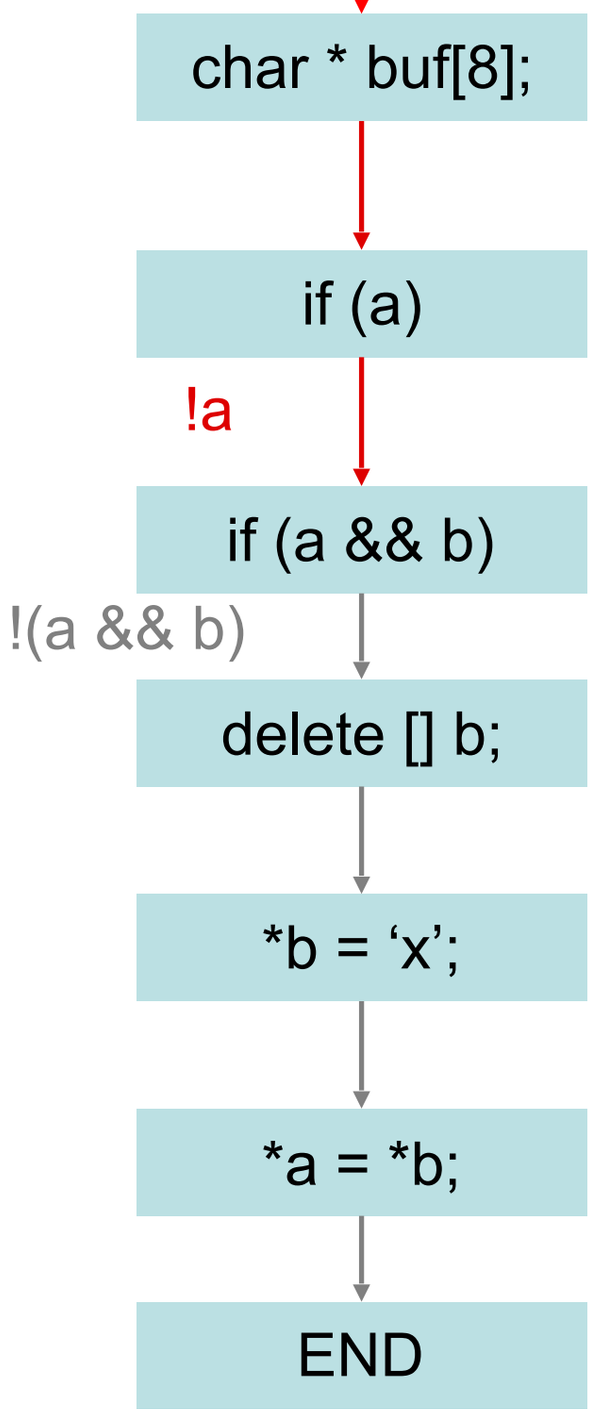
Null pointers | Use after free | Array overrun



“buf is 8 bytes”

Apply Checking

Null pointers | Use after free | Array overrun

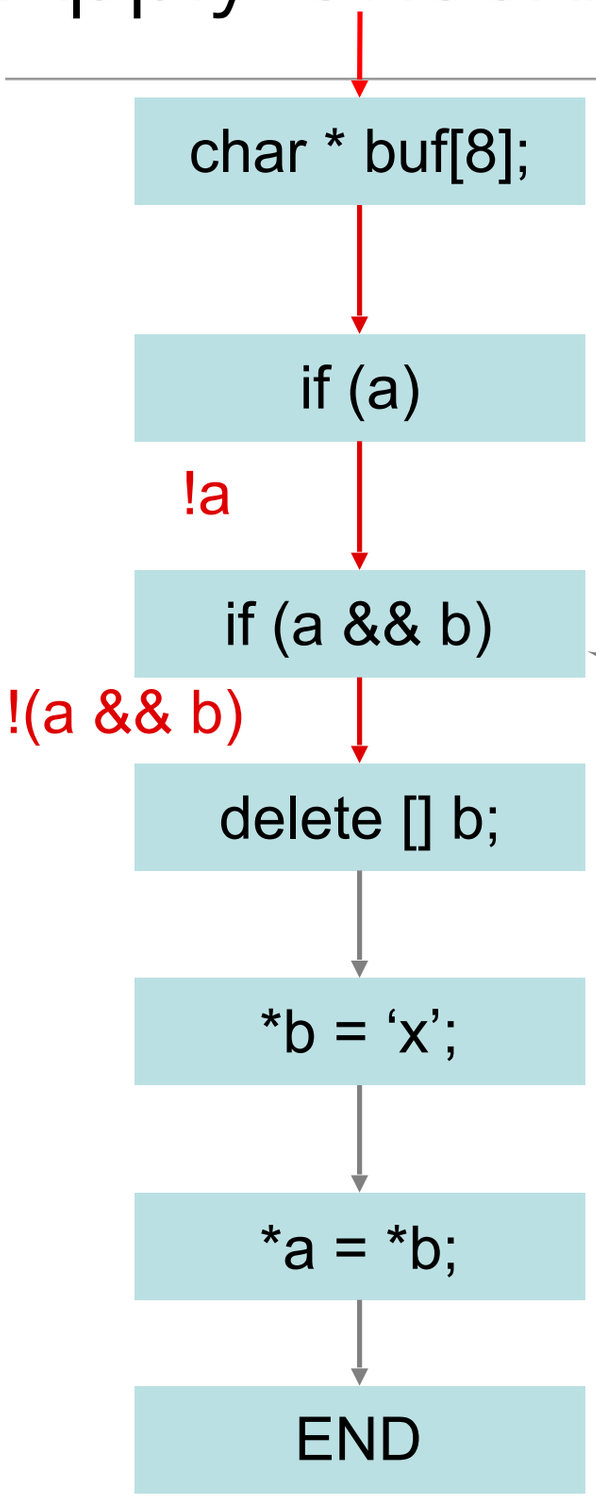


“a is null”

“buf is 8 bytes”

Apply Checking

Null pointers | Use after free | Array overrun



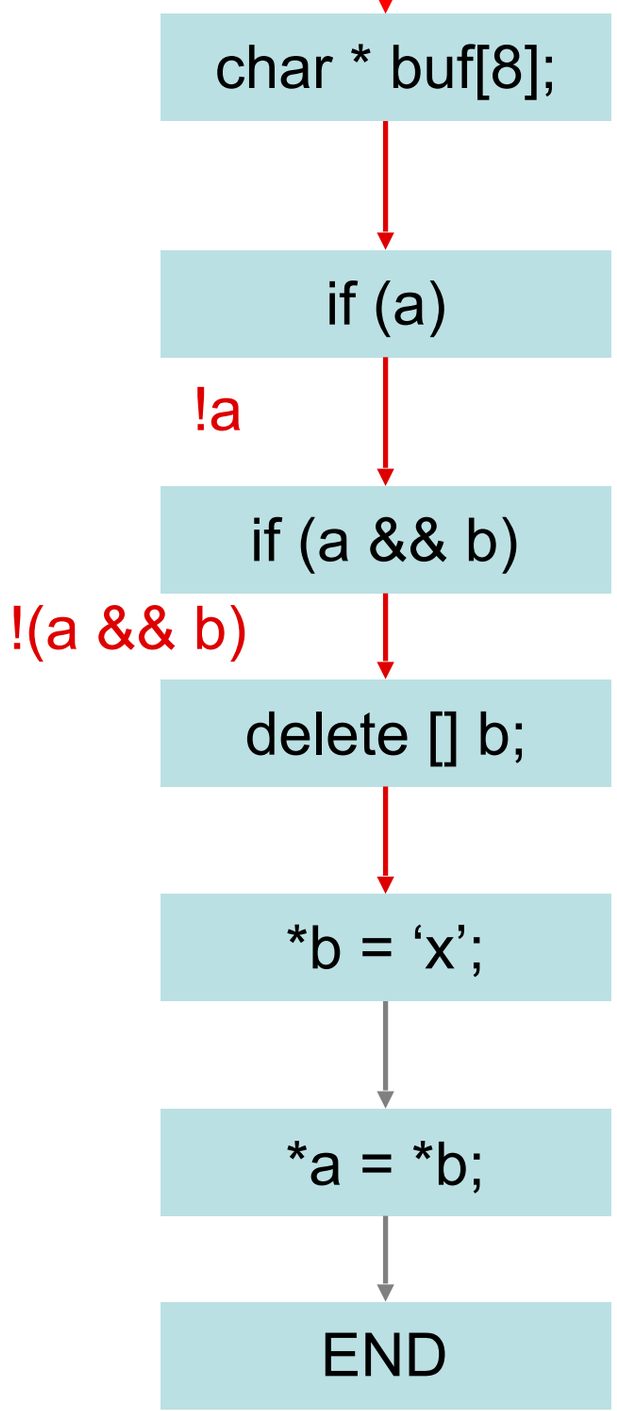
“a is null”

Already knew a was null

“buf is 8 bytes”

Apply Checking

Null pointers | Use after free | Array overrun

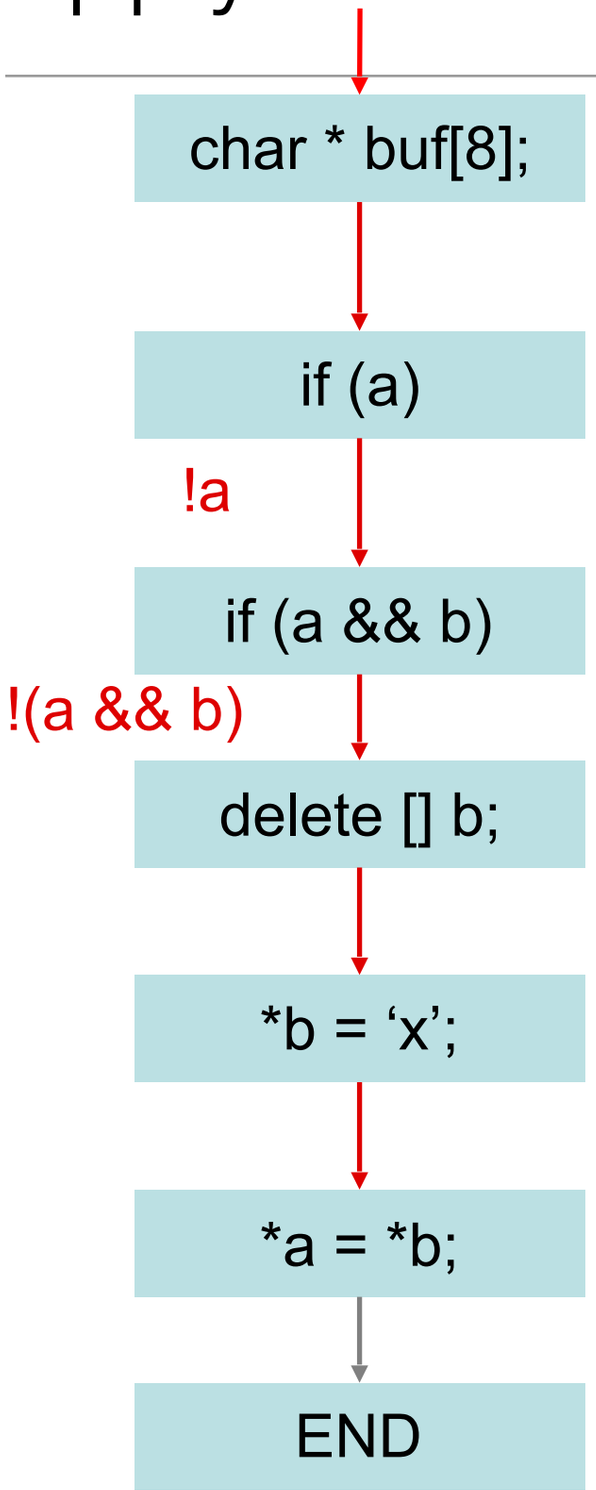


“buf is 8 bytes”

“a is null”

“b is deleted”

Apply Checking



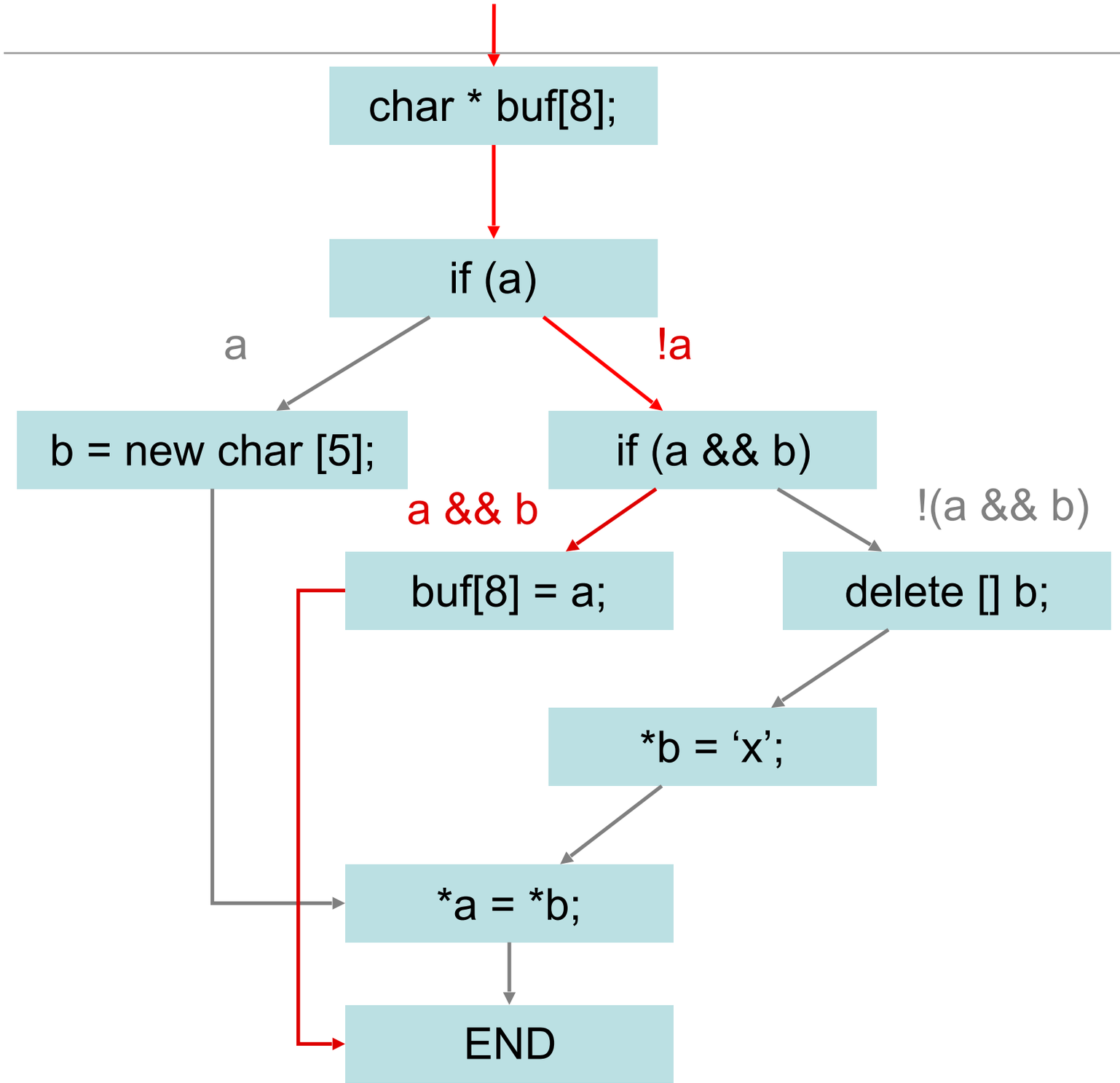
Null pointers	Use after free	Array overrun
		“buf is 8 bytes”
“a is null”		
	“b is deleted”	
	“b dereferenced!”	

False Positives

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

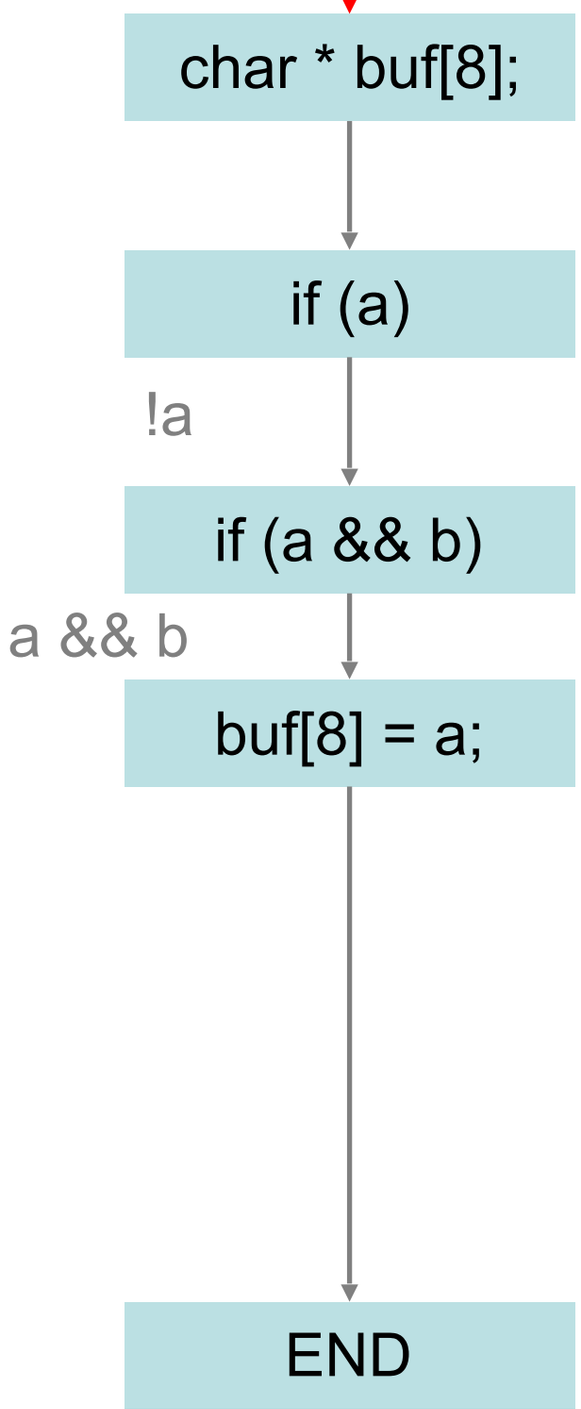
- Many sources of false positives
  - False paths
  - Idioms
  - Execution environment assumptions
  - Killpaths
  - Conditional compilation
  - “third party code”
  - Analysis imprecision
  - …

A False Path

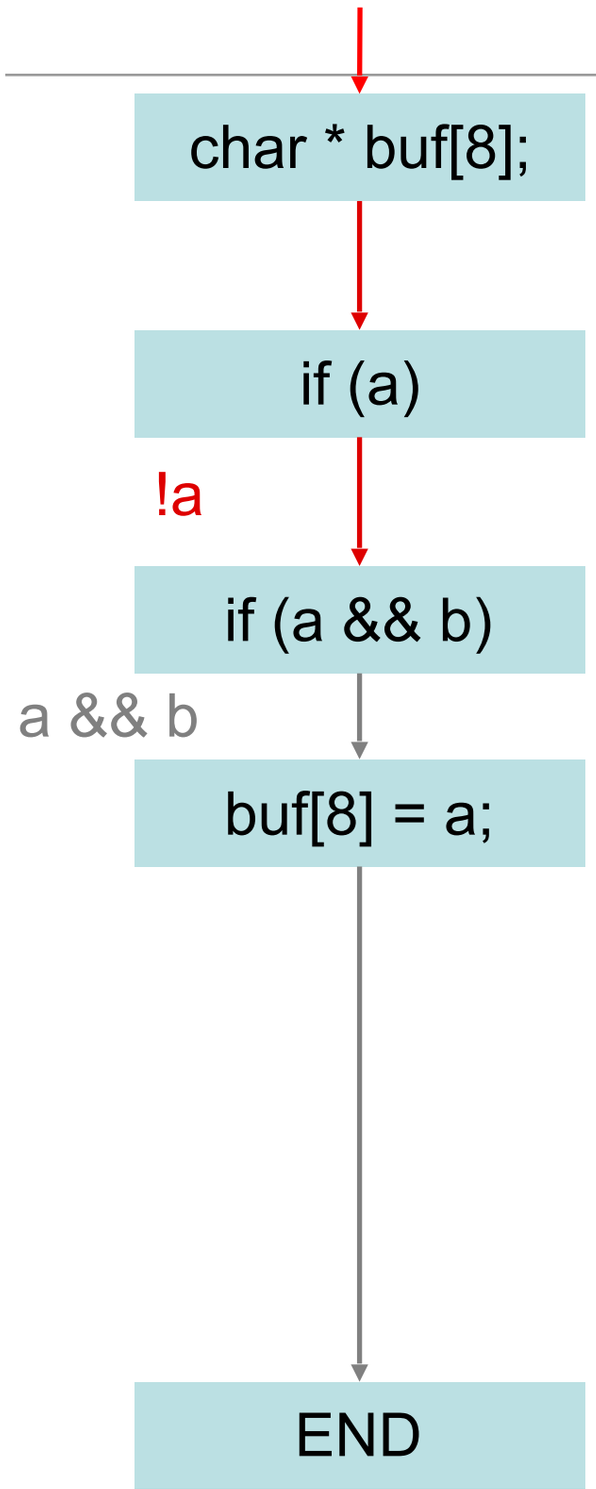


False Path Pruning

Integer Range | Disequality | Branch



False Path Pruning



Integer Range

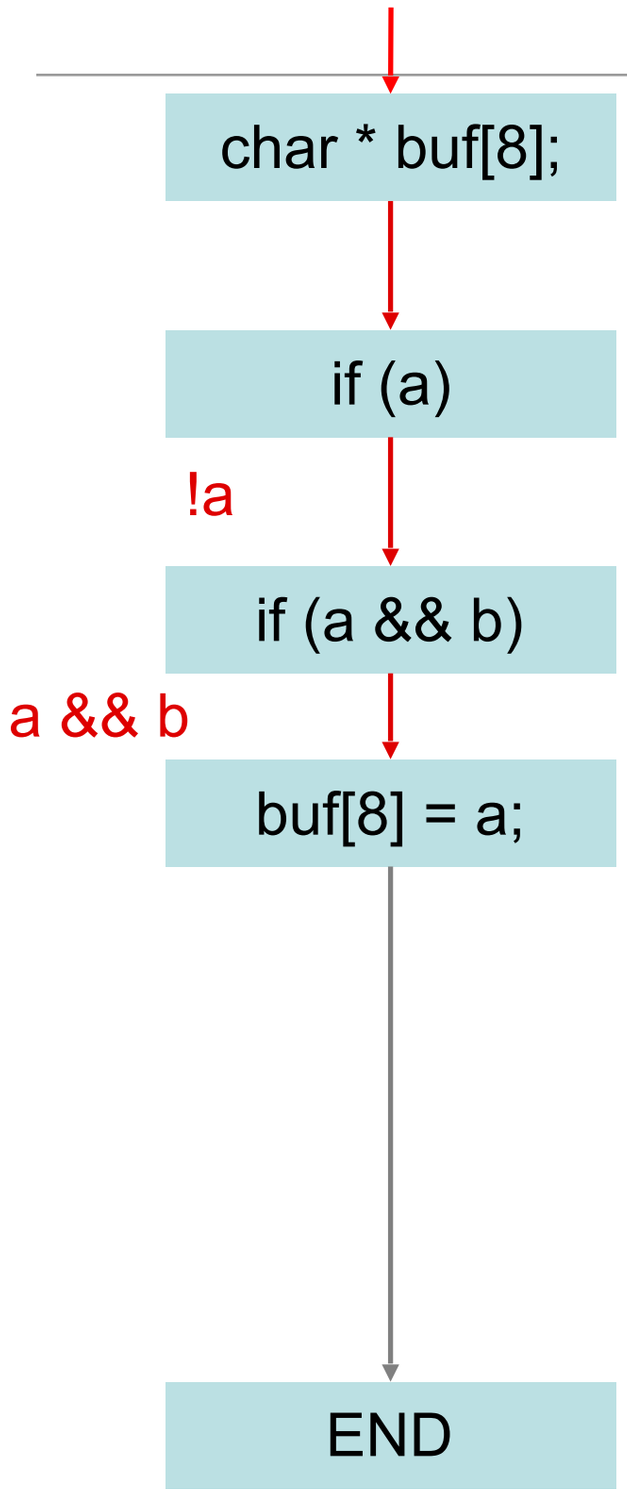
“a in [0,0]”

Disequality

Branch

“a == 0 is true”

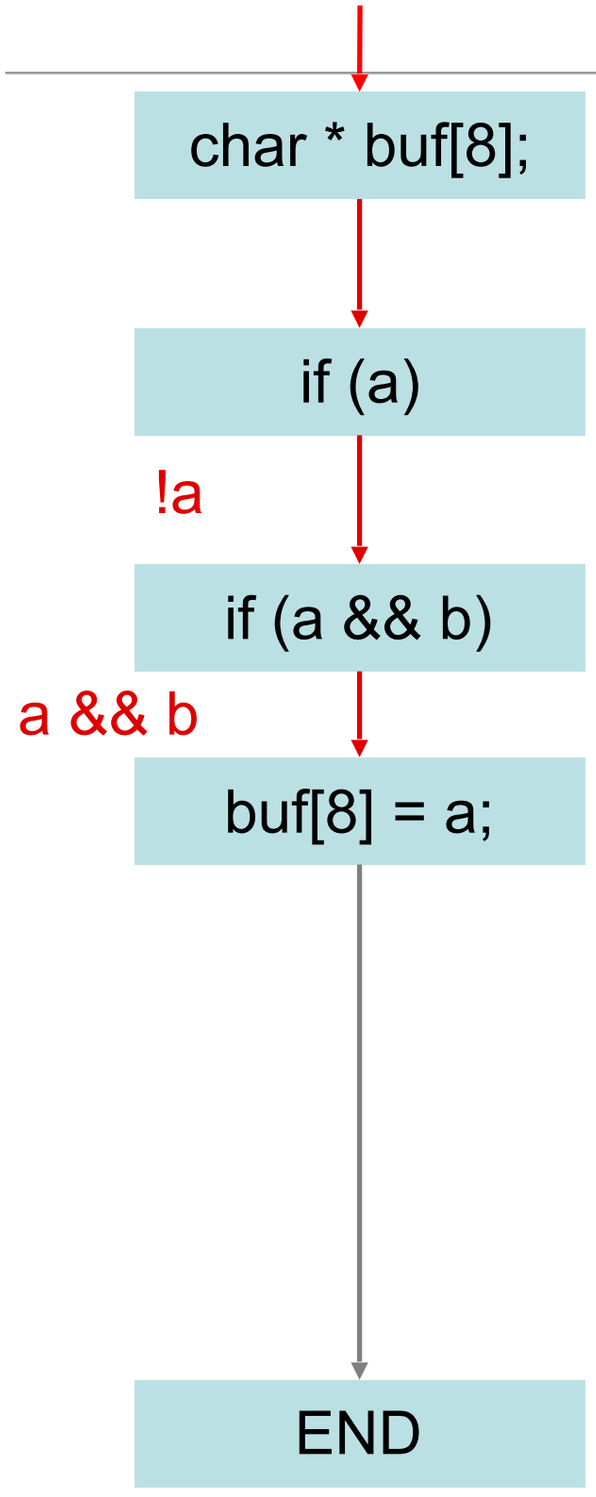
False Path Pruning



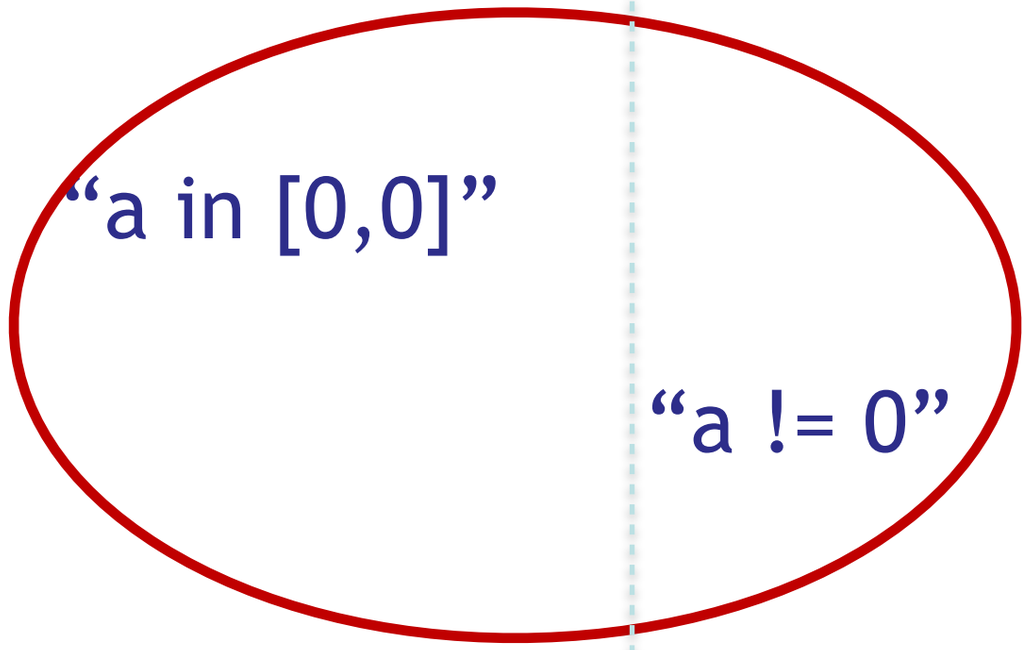
Integer Range	Disequality	Branch
“a in [0,0]”	“a != 0”	“a == 0 is true”

False Path Pruning

Integer Range | Disequality | Branch



Impossible



“a == 0 is true”

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));
if(p)  *p = 42;
```

1/4 deref

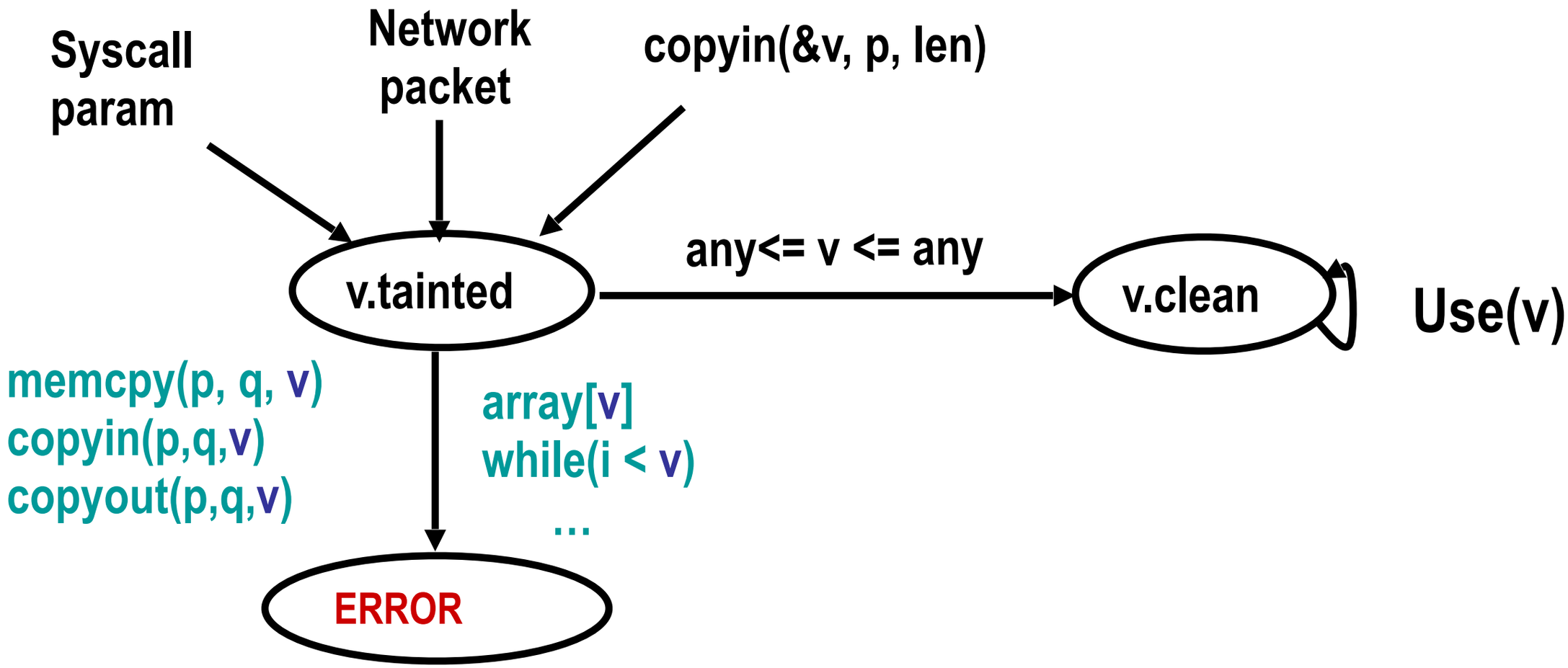
```
int *p = malloc(sizeof(int));
if(p) *p = 42;
```

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

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

Sanitize integers before use

Warn when unchecked integers from untrusted sources reach trusting sinks



Linux: 125 errors, 24 false; BSD: 12 errors, 4 false

Example security holes

- Remote exploit, no checks

```
/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
msg = skb->data;
...
memcpy(cmd.parm.setup.phone,
        msg->msg.connect_ind.addr.num,
        msg->msg.connect_ind.addr.len - 1);
```

Example security holes

- Missed lower-bound check:

```
/* 2.4.5/drivers/char/drm/i810_dma.c */

if(copy_from_user(&d, arg, sizeof(arg)))
   return -EFAULT;
if(d.idx > dma->buf_count)
   return -EINVAL;
buf = dma->buflist[d.idx];
Copy_from_user(buf_priv->virtual, d.address, d.used);
```

Results for BSD and Linux

- All bugs released to implementers; most serious fixed

Violation	Linux		BSD	
		Bug Fixed		Bug 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

This is all very nice, but how do you analyze the actual code?

How to find all code?

- Better: intercept and rewrite build commands

```
make –w  >     out
replay.pl out  # replace ‘gcc’ w/ ‘prevent’
```

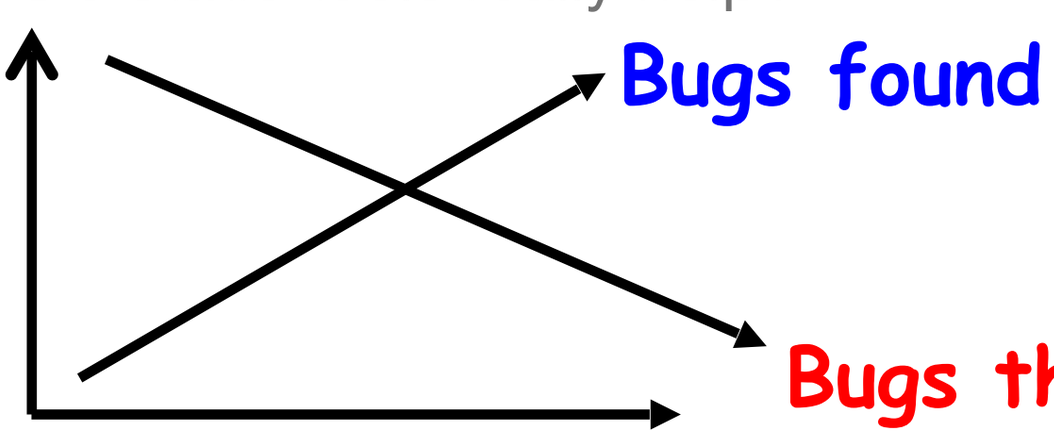
  - In theory: see all compilation calls and all options etc.
- Worked fine for a few customers.
  - Then: “make?”
  - Then: “Why do you only check 10K lines of our 3MLOC system?”
  - “Why do I have to re-install my OS from CD after I run your tool”

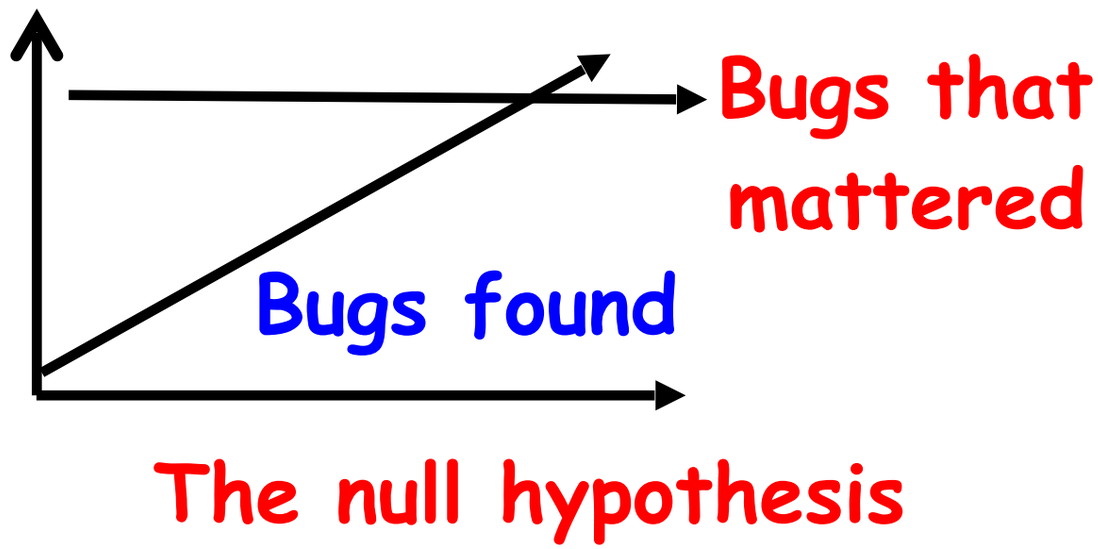
Some cursory experiences

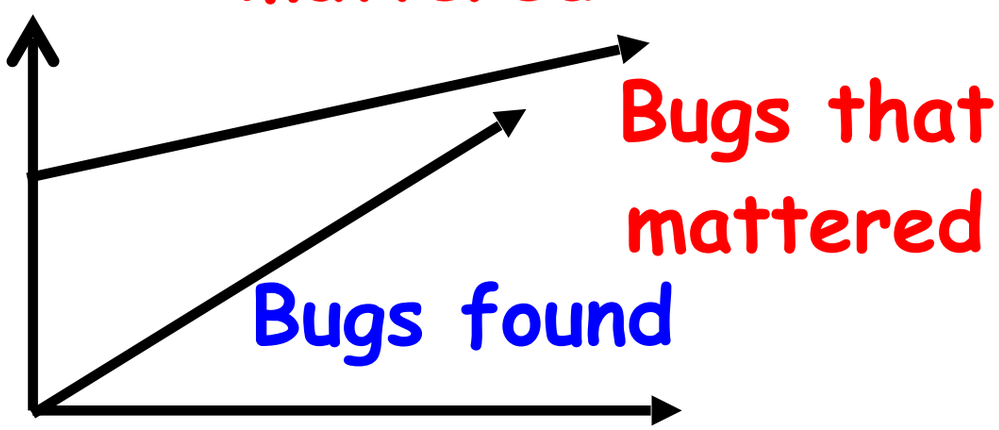
- Bugs are everywhere
  - Initially worried we'd resort to historical data…
  - 100 checks? You'll find bugs (if not, bug in analysis)
  - People don't fix all the bugs
- Finding errors often easy, saying why is hard
  - Have to track and articulate all reasons.

Two big open questions

- How to find the most important bug?
  - Main metric is bug counts or type
  - How to flag the 2-3 bugs that will really kill system?
- Do static tools really help?







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)

- [Engler'08] A couple billion lines of code later: static checking in the real world, Dawson Engler, Slides from Usenix Security 2008.