COMS 4995 - AI for Software Security

How Traditional Static Analysis Works, and Why/When It Fails

Zhuo Zhang

Jan 27, 2026

Today's goal

- Build intuition for **static program analysis**
- Learn the core program representations:
  - **AST**, **CFG**, **Call graph**
- Then: 3 classic analyses + failure modes + how AI helps
  - **Source $\rightarrow$ sink (taint/dataflow)**: null deref
  - **Abstract interpretation (intervals)**: buffer overflow
  - **Symbolic execution**: deep bug trigger inputs

Static analysis in one sentence

"Reason about program behavior **without running it**."

The 3-way tradeoff (remember this)

You usually can't maximize all three:

- **Soundness** (no missed bugs)
- **Precision** (few false alarms)
- **Scalability** (fast enough for real code)

Why failure is inevitable

Halting Problem:

- you can't build a program that always tells whether another program will ever stop or run forever.

Why failure is inevitable (Cont 1)

Why halting problem is undecidable?

1. Pretend we have a magic tool that can always tell whether any program will eventually stop.
2. Use that tool to build a “trick’’ program that deliberately does the opposite of what the tool predicts about itself.
3. When the trick program analyzes itself, the prediction is forced to be wrong, showing such a tool cannot exist.

Why failure is inevitable (Cont 2)

Many program analyses aim to answer:

- "Does this program ever reach line X?''
- "Can this loop terminate?''

These can be reduced to answering a **halting problem** instances.

Why failure is inevitable (Cont 3)

So: tools approximate.

Pipeline overview (what tools actually do)

Typical static analyzer pipeline:

1) Parse $\rightarrow$ **AST**
2) Lower to IR $\rightarrow$ build **CFG**
3) Add interprocedural info $\rightarrow$ **call graph**
4) Run analyses (dataflow / abstract interpretation / symbolic exec variants)
5) Report warnings

Checkpoint poll

Which one feels most mysterious?

A) AST

B) CFG

C) Call graph

D) “why it lies”

AST: what it is

AST = Abstract Syntax Tree

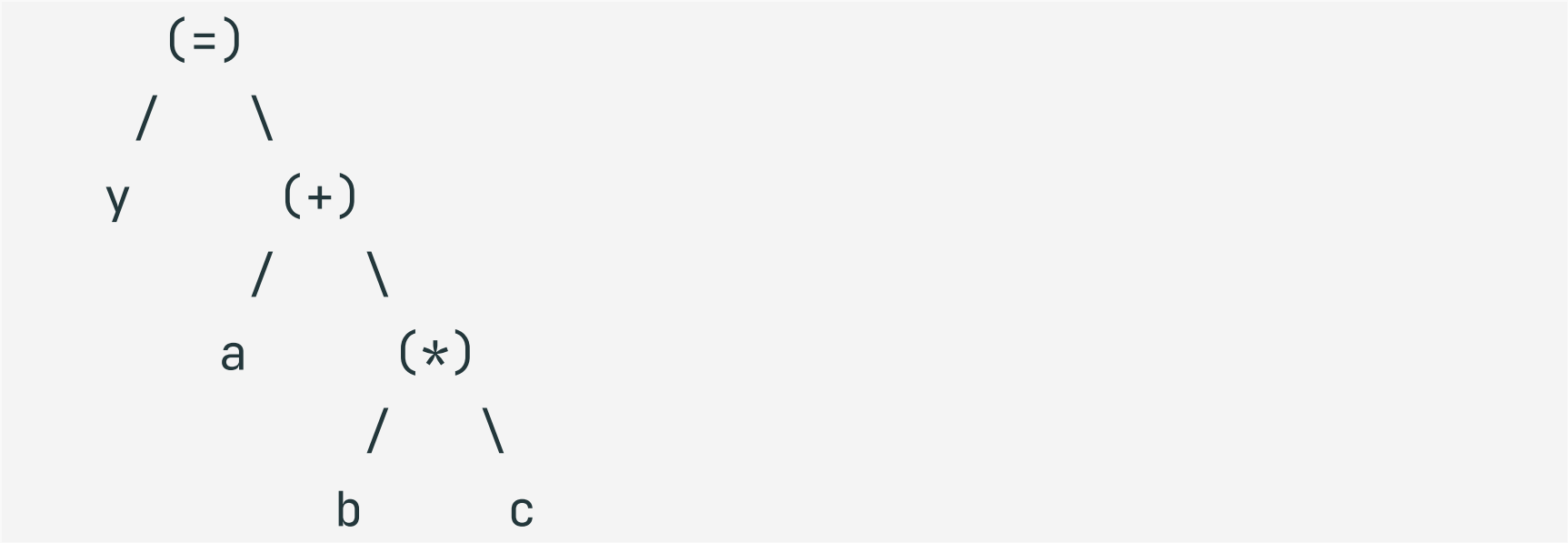
- Tree structure of "what the programmer wrote"
- Captures precedence, nesting, blocks, etc.
- Not about execution order (that's CFG)

Tiny C expression

```c
int y = a + b * c;
```

Quick: does + or * bind tighter?

AST sketch for `a + b * c`



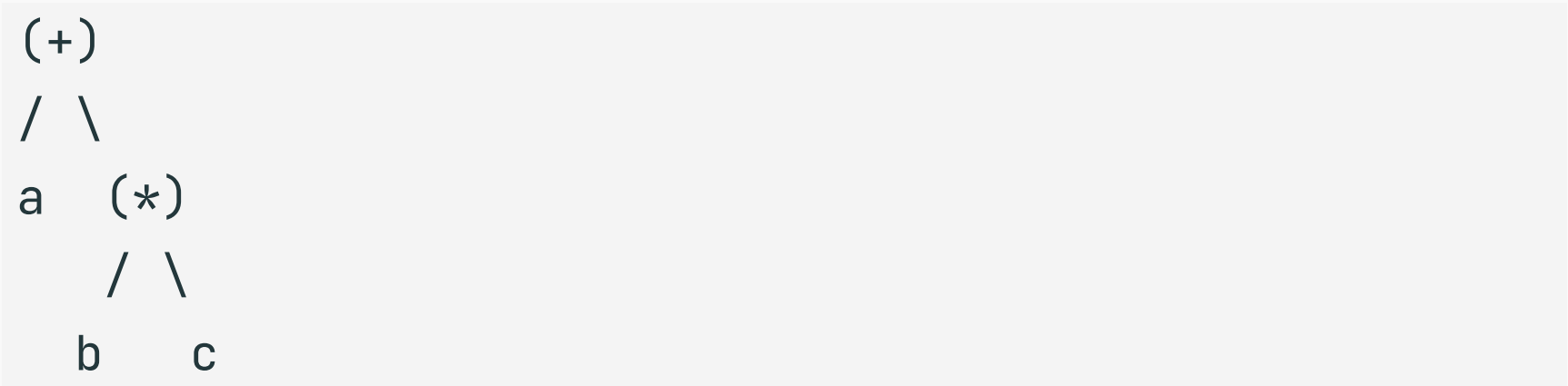
In-class Q1: which AST matches?

Code:

```
(a + b) * c;
```

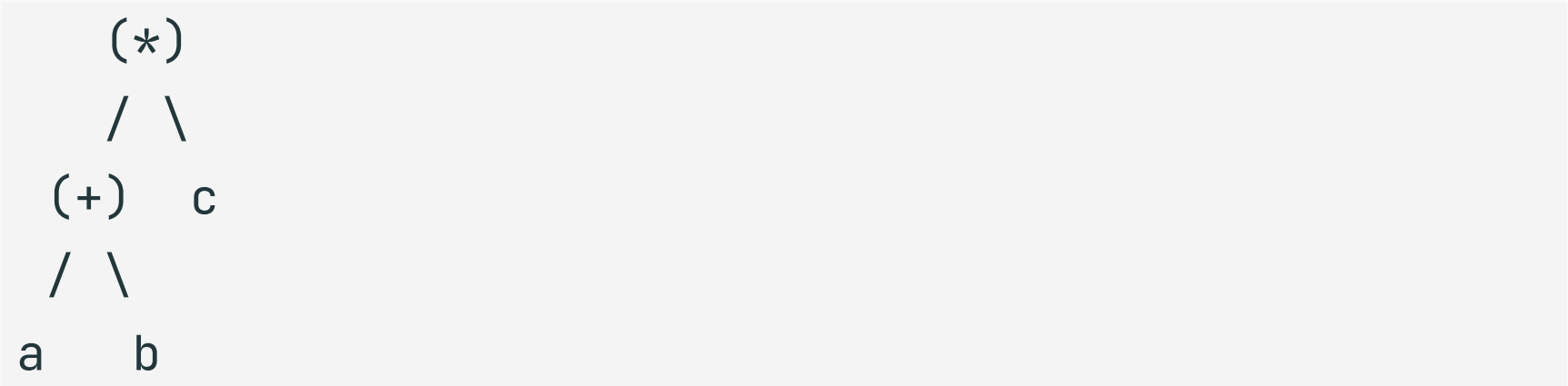
In-class Q1: which AST matches? (Options)

A)



In-class Q1: which AST matches? (Options)

B)



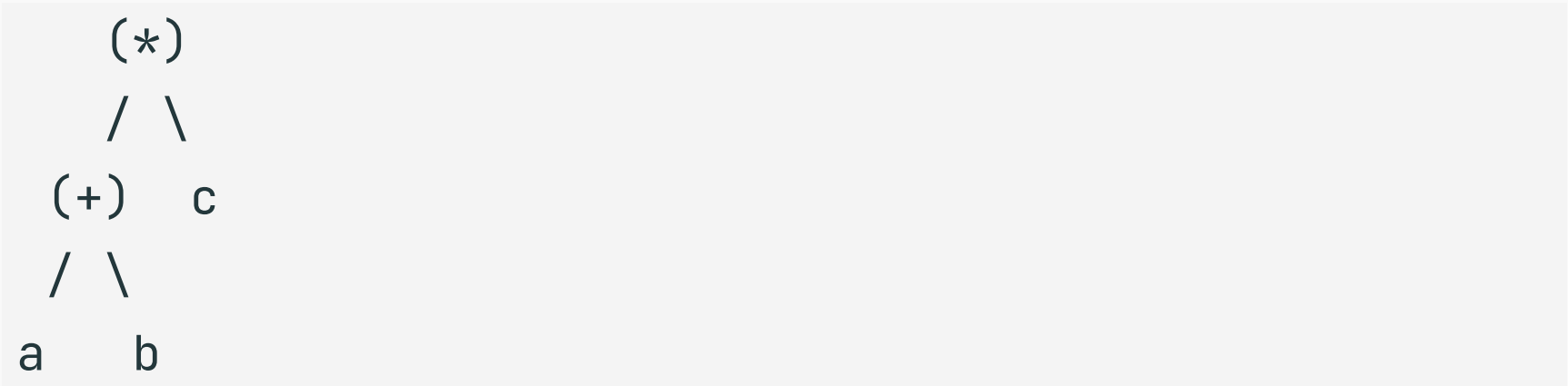
In-class Q1: which AST matches? (Options)

C)

```
  (*)
  / \
 a  (+)
/ \
b  c
```

Q1 answer

Correct: **B**



AST: statements add structure

```
int f(int x) {
  int y = 0;
  if (x > 0) y = 1;
  else       y = 2;
  return y;
}
```

AST sketch (statements)

```
Function f
  Params: x
  Block
    Decl y = 0
    If (x > 0)
      Then: Assign y = 1
      Else: Assign y = 2
    Return y
```

In-class Q2: what's NOT in AST?

A) operator precedence

B) nested blocks

C) "next executed statement" edges

D) which variable is assigned

Q2 answer

Correct: **C**

Execution order edges live in the **CFG**.

A fun AST gotcha (C)

```
if (x)
  if (y) z();
  else   w();
```

Question: which `if` does the `else` belong to?

The “dangling else” rule

In C: `else` matches the **nearest unmatched** `if`.

So it parses like:

```
if (x) {
  if (y) z();
  else   w();
}
```

If you want different grouping:

```
if (x) {
  if (y) z();
} else {
  w();
}
```

CFG: what it is

CFG = Control Flow Graph

- Models **possible execution order**
- Nodes: **basic blocks** (straight-line sequences)
- Edges: possible jumps (branches, loops, returns)

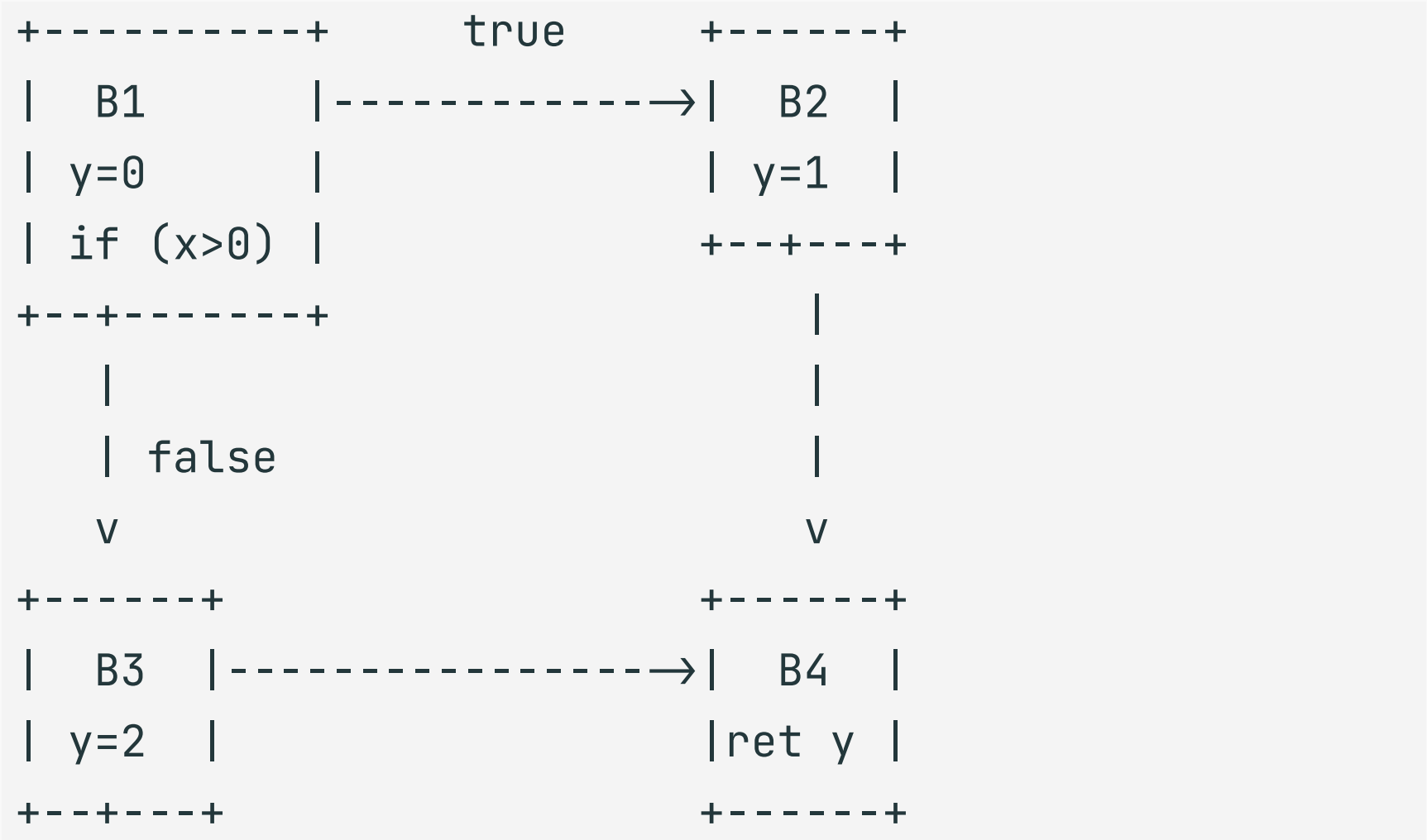
Same function; now think CFG

```
int f(int x) {
  int y = 0;
  if (x > 0) y = 1;
  else       y = 2;
  return y;
}
```

Basic blocks (typical)

- B1: `y=0; if (x>0) goto B2 else goto B3`
- B2: `y=1; goto B4`
- B3: `y=2; goto B4`
- B4: `return y`

CFG sketch (ASCII)



In-class Q4: CFG vs AST?

?

In-class Q4: CFG vs AST? (Cont)

A) captures execution order

B) tree $\rightarrow$ graph

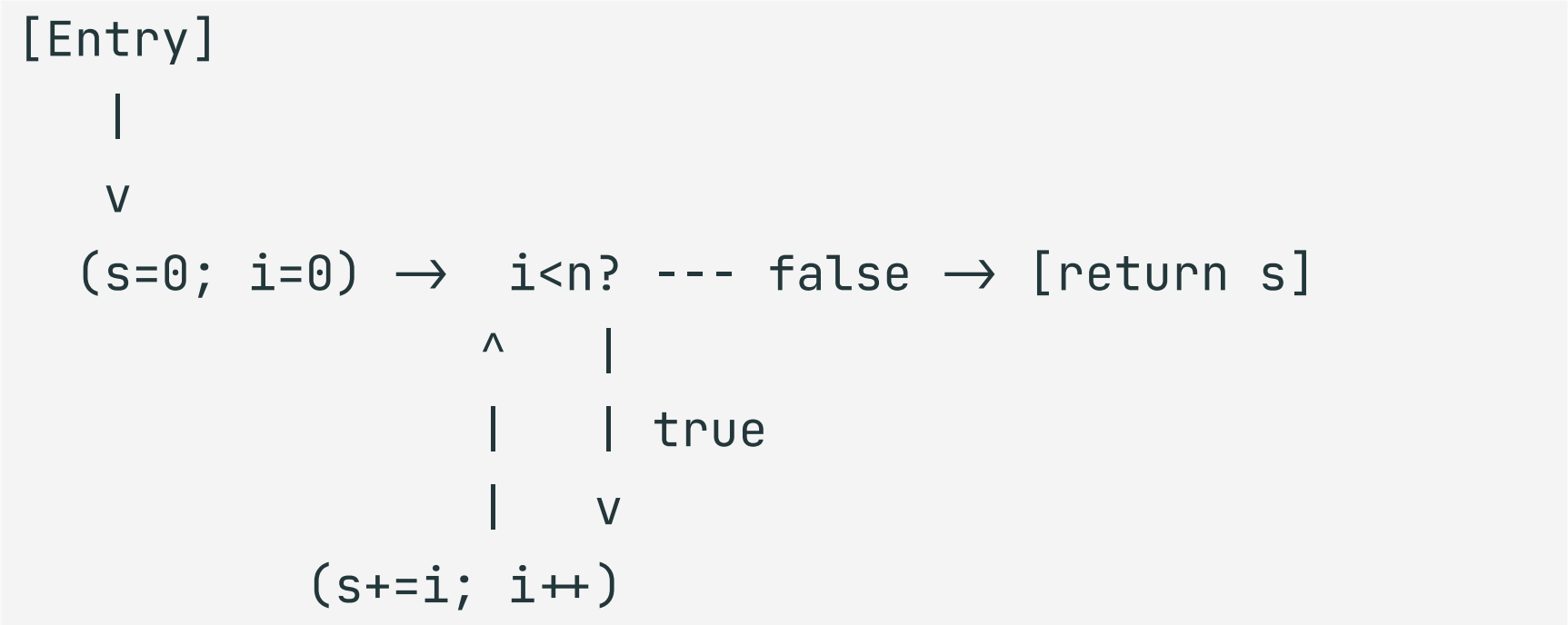
C) introduces back edges for loops

CFG exercise: a loop

```
int sum_to(int n){
  int s = 0;
  for (int i=0; i<n; i++){
    s += i;
  }
  return s;
}
```

Where is the back-edge?

Loop CFG sketch (ASCII)



In-class Q5: “basic block” means...

A) one statement per node

B) no internal branches except at end

C) only functions, not loops

D) only for assembly

Q5 answer

Correct: **B**

Call graph: what it is

Call graph = possible calls between functions

- Node: function
- Edge: "may call"

Great for interprocedural analysis (across functions).

Easy case: direct calls

```
int g(int);

int f(int x){ return g(x) + 1; }
int g(int y){ return y*y; }
```

Call graph edge?

Answer

```
f → g
```

Hard case: function pointer

```
int add1(int x){ return x+1; }
int add2(int x){ return x+2; }

int (*pick(int k))(int){
  if (k) return add1;
  else   return add2;
}

int h(int k, int v){
  int (*fp)(int) = pick(k);
  return fp(v);
}
```

What edges must be included conservatively?

Conservative call graph edges

At least:

- `h → pick`
- `h → add1`
- `h → add2`

Because `fp(v)` could call either target.

In-class Q6: why is call graph hard?

A) recursion

B) indirect calls (function pointers / dynamic dispatch)

C) whitespace sensitivity

D) constant folding

Q6 answer

Correct: **B** (recursion is fine; indirect calls are the real pain)

Mini recap checkpoint

Match representation to question:

- AST answers: “how is code structured syntactically?”
- CFG answers: “what order might statements execute?”
- Call graph answers: “which functions might be called?”

Now we hunt bugs

We'll start with a bug class that feels concrete:

- **Null pointer dereference**

We'll turn it into a **source** $\rightarrow$ **sink** problem.

Bug hunt #1: spot the crash

```c
#include <stdio.h>
#include <stdlib.h>

char *read_name(FILE *f){
  char *p = malloc(16);
  fgets(p, 16, f);
  return p;
}

int main(){
  FILE *f = fopen("names.txt","r");
  char *s = read_name(f);
  printf("%c\n", s[0]);
}
```


Where can it crash? (list all spots)

Likely crash points

- `fopen` may return **NULL** $\rightarrow$ `f` null passed to `read_name`
- `s` may be **NULL** if `read_name` returned NULL (or if it propagates)
- `s[0]` deref is a **sink**

Source vs sink (make it a game)

Define:

- **Sources**: expressions that may produce `NULL` (e.g., `malloc`, `fopen`, `getenv`, many APIs)
- **Sinks**: operations requiring non-null `*p`, `p->x`, `p[i]`, `memcpy(p, ...)`, etc.

Goal: detect **source** $\rightarrow$ **sink** flow without check.

Taint-style view (null-taint)

We'll use a `taint` metaphor:

- value is **nullable-tainted** if it may be NULL
- taint propagates through assignments and returns
- checks may "sanitize" (on some paths)

Dataflow (toy rules)

On CFG:

- `x = malloc(...)` $\rightarrow$ `x` becomes **nullable**
- `x = y` $\rightarrow$ `x` nullable if `y` nullable
- `if (x` $\neq$ `NULL)`:
  - on **true branch**, treat `x` as non-null
- sink `x[0]` $\rightarrow$ warn if `x` may be null *on that path*

Sensitivity (static analysis)

Sensitivity is how much the analysis **distinguishes different situations**

- **Flow sensitivity**
- **Path sensitivity**
- **Context sensitivity**

Flow sensitivity

Flow sensitivity: distinguishes facts at **different program points**

Statement order matters!

In-class Q7: flow sensitivity

If analysis is **flow-insensitive**, it:

A) tracks facts per program point

B) ignores statement order (one big soup)

C) always considers paths separately

D) requires SMT solvers

Q7 answer

Correct: **B**

Flow-insensitive = ignores order → often much less precise.

Path sensitivity

Path sensitivity: distinguishes facts on **different branches/paths**

Conditions matter!

In-class Q8: path sensitivity

Given:

```
p = malloc(16);
if (p) p[0] = 'A';
```

A **path-insensitive** analysis that merges facts will likely:

A) warn

B) not warn

C) crash itself

Q8 answer

Often **A (warn)**

Because merging keeps “p may be null’’ even inside the guarded block (depending on tool design).

A cleaner null example (with early return)

```c
char *p = malloc(16);
if (!p) return NULL;
p[0] = 'A';
```

A path-sensitive analysis should: warn or not?

Answer

Should **not warn** at `p[0]` if it understands that path implies `p` $\neq$ `NULL`.

Interprocedural challenge

```c
char *mk(){ return malloc(1024 * 1024 * 1024); }

void use(char *q){ q[0] = 'X'; }

int main(){
  char *p = mk(); use(p);
}
```

To warn, the analyzer must connect:

- return value of `mk` to argument of `use`

Summaries (the key idea)

Interprocedural dataflow often uses **function summaries**:

- `mk`: “returns nullable”
- `use`: “requires non-null q at deref”

Then check call sites.

Context sensitivity

Context sensitivity: distinguishes facts under **different calling contexts**.

Same function called from different places is analyzed separately!

In-class Q9: context sensitivity

Context-insensitive means:

A) different call sites to same function can be distinguished

B) variables have types

C) loops are unrolled

D) AST nodes have parents

Q9 answer

Correct: **A**

Why this analysis fails

We'll do 4 common ones:

1. aliasing
2. unknown function semantics ("sanitizers'')
3. path explosion
4. imprecise heap / pointers

Failure mode #1: aliasing

```
char *p = malloc(16);
char *q = p;

... // complex code that doesn't change p or q

if (!q) return;
p[0] = 'A';
```

If tool doesn't track aliasing well, it might not realize `p` and `q` share nullness facts.

Failure mode #2: “sanitizer” functions

```c
void check_not_null(void *x){
  if (!x) abort();
}

char *p = malloc(16);
check_not_null(p);
p[0] = 'A';
```

If tool doesn’t know `check_not_null` semantics, it may warn incorrectly.

Failure mode #3: path explosion

Many branches cause exponential paths:

```
if (a) {...} else {...}
if (b) {...} else {...}
if (c) {...} else {...}
```

Even a great path-sensitive idea can become too slow.

Failure mode #4: “unknown” libraries/environment

```
p = get_ptr_from_library();
use(p); // deref inside
```

If the library contract isn't modeled, analysis is forced to guess:

- assume nullable $\rightarrow$ false positives
- assume non-null $\rightarrow$ false negatives

Where AI helps (source → sink / null deref)

AI won't magically make the problem decidable. But it can improve usefulness:

- **Learn common sanitizer patterns**
- **Infer API contracts** (nullable vs non-null)
- **Rank warnings** (triage 10k → top 20)
- **Suggest fixes** ("add check here")

AI help #1: warning ranking (triage)

Given many warnings, prioritize likely true positives using:

- code patterns near sink
- historical fixes / repo conventions
- similarity to known bugs
- call context features

AI help #2: contract inference

Infer `this function never returns NULL''` oraborts on NULL'' by:

- analyzing implementation
- learning from usage patterns:
  - "everyone checks return value'' $\rightarrow$ maybe nullable
  - "nobody checks'' $\rightarrow$ maybe non-null or people are wrong

In-class Q10: AI *often* does NOT give...

A) better prioritization

B) learned summaries for wrappers

C) guaranteed soundness

D) better UX