COMS 4995 - AI for Software Security

Symbolic Execution vs. Abstract Interpretation

Zhuo Zhang

Jan 29, 2026

Part 1 — The program, CFG, and the bug

Example program

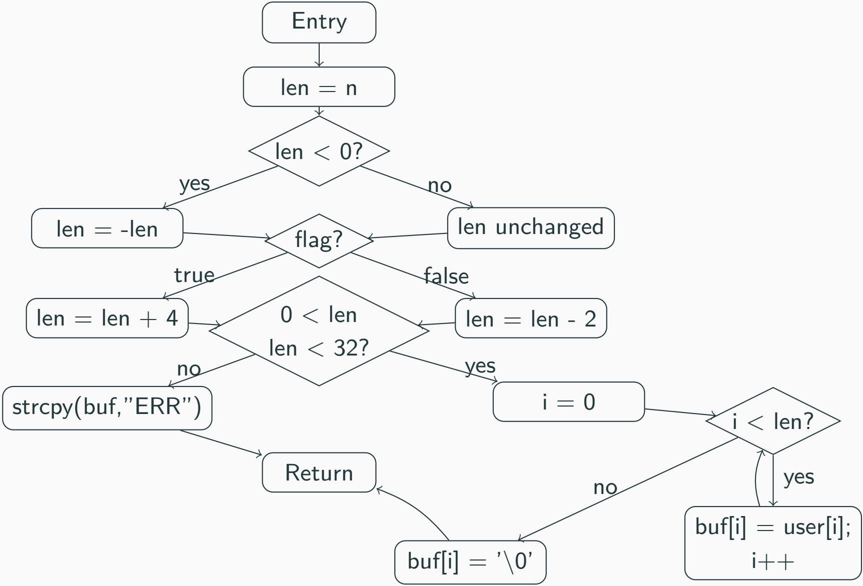
```c
int foo(const char *user, int n, int flag) {
    char buf[16];
    int len = n;

    if (len < 0) len = -len;

    if (flag)  len = len + 4;
    else       len = len - 2;

    if (0 < len && len < 32) {
        int i = 0;
        while (i < len) {
            buf[i] = user[i];
            i++;
        }
        buf[i] = '\0';
    } else {
        strcpy(buf, "ERR");
    }
    return (int)buf[0];
}
```

CFG (control-flow graph)



Side-by-side

Write happens here:

```c
while (i < len) {
    buf[i] = user[i];   // i is NOT bounded by 16
    i++;
}
buf[i] = '\0';          // also writes at index i
```

Why it overflows: `buf` has valid indices `0..15`, but `i` can reach `16..31`.

Concrete failing scenario: `len = 20` → loop writes `buf[16]`, `buf[17]`, … (stack smash territory)

Part 2 — Symbolic execution

Metaphor: two ways to "run" a program

- **Concrete execution:** you watched *one person's walk*.
- **Symbolic execution:** you tracked *every possible walk*, but had to do forking at every branches.

Symbolic execution: core idea

Replace inputs with symbols:

- `N` for `n`
- `F` for `flag` (boolean)

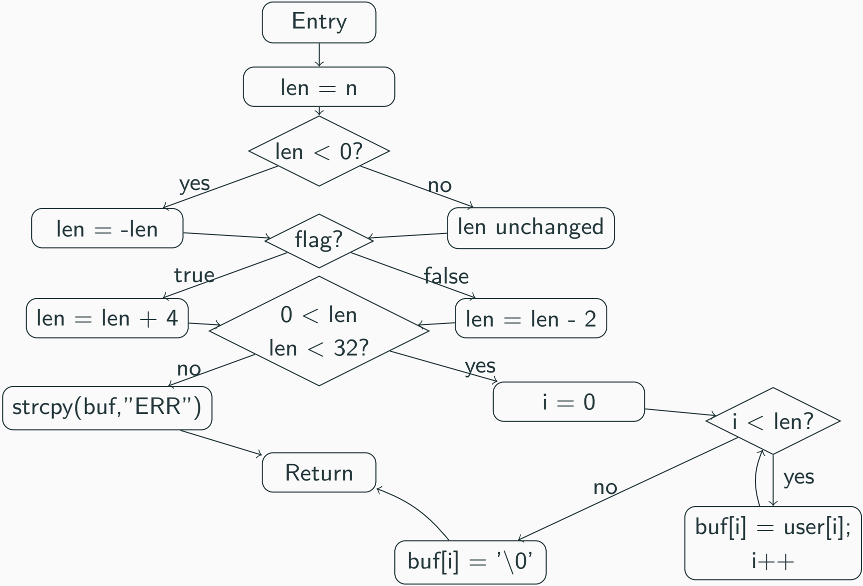
Track a **symbolic state**:

- symbolic store (expressions for variables)
- path condition `PC` (constraints that must hold)

At each branch:

- fork states and add constraint (`cond` / `!cond`)
- use an SMT solver to check feasibility and (optionally) produce a model (test input)

Symbolic setup for this program



But... the loop makes symbolic execution blow up

Inside the `while (i < len)`:

- each iteration hits a branch (`i < len`)
- if `len` is symbolic, the executor conceptually explores:
  - paths with 0 iters, 1 iter, 2 iters, ... up to many iters
- if nested loops / multiple branches exist: it's exponential ("path explosion")

This program is small; real code has:

- multiple loops, function calls, recursion
- complex conditionals
- library modeling gaps

So we ask: can we avoid enumerating every route?

But… the loop makes symbolic execution blow up

```c
int foo(const char *user, int n, int flag) {
    char buf[16];
    int len = n;

    if (len < 0) len = -len;

    if (flag)  len = len + 4;
    else       len = len - 2;

    if (0 < len && len < 32000000000) {
        int i = 0;
        while (i < len) {
            buf[i] = user[i];
            i++;
        }
        buf[i] = '\0';
    } else {
        strcpy(buf, "ERR");
    }
    return (int)buf[0];
}
```

Common “symex” mitigations (still not a silver bullet)

- **Bounded exploration** (loop unrolling limit)
  - predictable, – can miss deep bugs
- **State merging** (merge paths at join points)
  - fewer states, – constraints become harder / less precise
- **Heuristic path search** (coverage-guided, BFS/DFS hybrids)
  - finds bugs faster, – completeness suffers
- **Concolic execution** (concrete + symbolic)
  - scalable-ish, – can still miss paths

This motivates a different idea: `don't track exact formulas per path`

Part 3 — Abstract interpretation: stop chasing paths

Abstract interpretation: core idea

Symbolic execution tracks **exact expressions** per path.

Abstract interpretation tracks **summaries** over *sets* of states.

- Instead of: `len = (N ⩾ 0 ? N : -N) + 4`
- You might track: `len` ∈ `[1, 31]` on the then-branch

You trade:

- **precision** (exactness) ↓
- for **scalability** and **guarantees** (soundness) ↑

Metaphor: “weather map analysis”

- **Concrete execution:** you watched *one person’s walk*.
- **Symbolic execution:** you tracked *every possible walk*, but had to fork at every branch.
- **Abstract interpretation:** you publish a *weather map*: “anyone walking here will experience temperatures in [20°C, 30°C].”

You lose exact trajectories, but you can cover the whole city.

The math-y skeleton (without drowning in it)

We define:

- a **concrete domain** `C` (all real program states)
- an **abstract domain** `A` (compact summaries)

With maps:

- **abstraction** $\alpha : P(C) \to A$
- **concretization** $\gamma : A \to P(C)$

We compute a sound over-approx:

> *`AbstractResult` describes a set of states that includes all real reachable states.*

That's why abstract interpretation is great for "prove no overflow" (when it succeeds).

Worklist algorithm (forward dataflow)

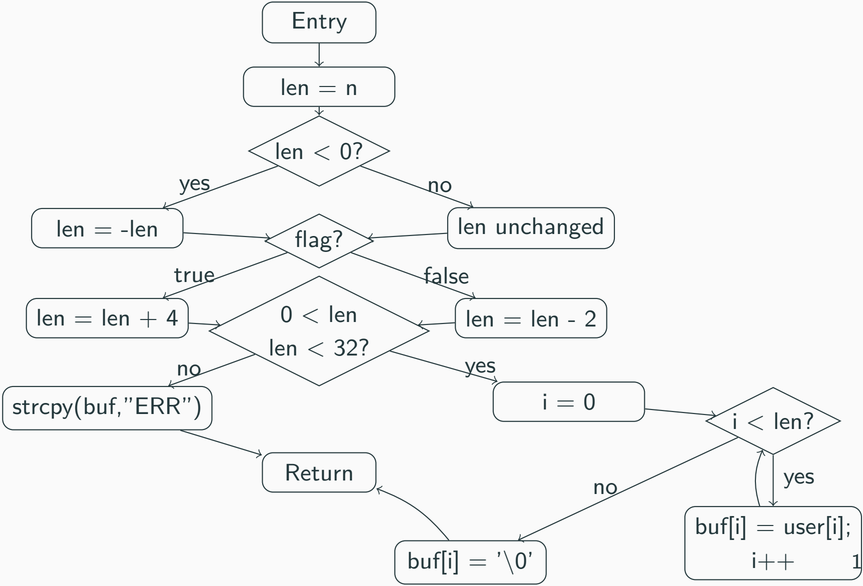
We compute an invariant at each CFG node.

```
Initialize IN[n] = ▯, OUT[n] = ▯
IN[entry] = initial abstract state

worklist = [entry]
while worklist not empty:
  p = pop(worklist)
  OUT[p] = transfer(p, IN[p])
  for each successor s of p:
    newIN = IN[s] ▯ OUT[p]          // join
    if newIN ≠ IN[s]:
      IN[s] = newIN
      push(s)
```

For loops: iteration may not terminate → we use **widening** to

Worklist algorithm (forward dataflow)



Interval domain (our abstract domain)

For each integer variable `x`, track an interval:

- $x \in [l, u]$ where `l` and `u` can be $-\infty, +\infty$

Key operators:

- **join**: `[l1,u1] ⍁ [l2,u2] = [min(l1,l2), max(u1,u2)]`
- **add**: `[l,u] + k = [l+k, u+k]`
- **sub**: `[l,u] - k = [l-k, u-k]`
- **guard refine**:
  - for `x < c`, intersect with `[-∞, c-1]`
  - for `x > c`, intersect with `[c+1, +∞]`

Intervals are fast—but they forget correlations (e.g., they don't remember `i < len` very precisely).

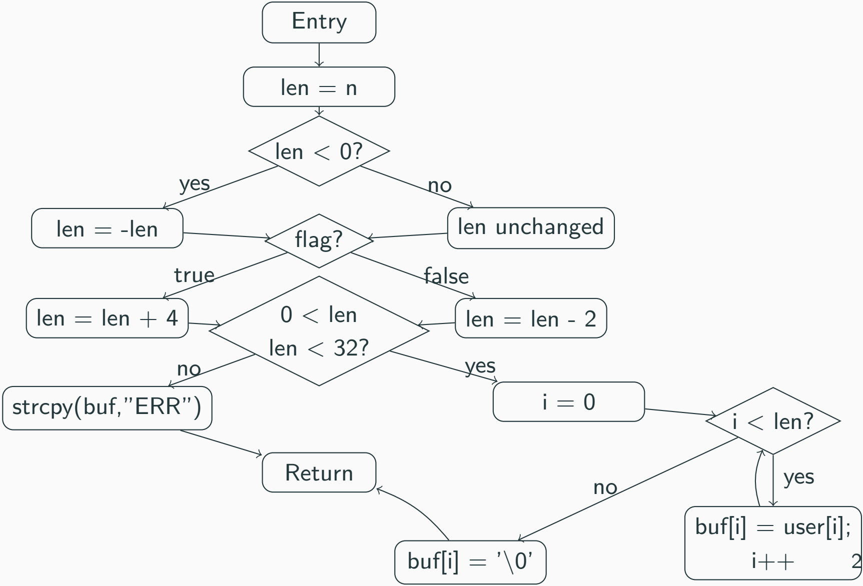
Set up analysis assumptions (so intervals can run)

We need an input range to analyze “all at once”. Example (typical in static analysis):

- `n ∈ [-∞, +∞]`
- `flag ∈ {0,1}`

We’ll focus on key variables: `len`, `i`.

Interval propagation (key program points)



Why this scales better than symbolic execution (intuition)

- Symbolic execution: cost grows with **#paths** (and loop iterations)
- Abstract interpretation: cost grows with **#CFG nodes × domain ops**
  - you iterate to a fixpoint
  - you summarize many paths into one invariant per node

So you "pay per node," not "pay per path."

Part 4 — Where abstract interpretation struggles

Failure mode #1: widening → coarse invariants (false alarms)

To ensure termination on loops, analyzers apply **widening**:

- if a bound keeps increasing, jump to +∞ (or a large summary)

Example (safe-ish structure, but widening can lose it):

```c
int i = 0;
while (i < len) {      // len unknown, might be bounded els
    i++;
}
if (i < 16) {
    buf[i] = 'A';      // safe whenever i < 16
}
```

If widening turns `i ∈ [0, +∞)` at loop head:

- the analyzer can't prove `i < 16` is reachable/safe precisely
- you may get a warning even if upstream logic actually bounds

Failure mode #2: bit operations don't fit the domain

Intervals are bad at bit-level reasoning unless you add custom transfer rules.

Example:

```
uint32_t idx = x & 0xF;   // idx is ALWAYS 0..15
buf[idx] = 'A';           // safe
```

A naïve interval analyzer might know:

- `x ∈ [0, 2^32-1]` …but not know how & `0xF` constrains values, so it may approximate:
- `idx ∈ [0, 2^32-1]` (terrible) → false alarm.

Fix: use a domain that models bitmasks / congruences / modular arithmetic, or add bit-precise reasoning.

Part 7 — How AI can help (without magic)

Where AI can plug in (practical angles)

1. **Better heuristics for symbolic execution**
   - learn which branches / paths are likely to reach “dangerous sinks” (memcpy, buffer writes)
   - prioritize solver calls that maximize coverage or bug likelihood
2. **Invariant suggestion / refinement**
   - propose candidate invariants (e.g., `len ≤ 15`) that a prover can check
   - help choose predicates for CEGAR-style refinement
3. **Domain selection / hybrid analysis**
   - detect “bit-heavy” code and switch to a more suitable abstract domain
   - or combine: abstract interpretation to prune, symbolic execution for precision (“meet in the middle”)
4. **Learned transfer functions (carefully)**

Takeaways (what to remember)

- **Symbolic execution**:
  - path-based, constraint-based
  - great for **concrete counterexamples**
  - struggles with **path explosion**
- **Abstract interpretation**:
  - summary-based, fixpoint-based
  - great for **scalability** and **sound over-approx**
  - can lose precision (widening, domain mismatch)
- **In practice**: strong tools mix both—and AI can help decide *where* and *how*.