Secure Software Design and Engineering
(CY-321)

Static Analysis

Dr. Zubair Ahmad

Static Analysis

Static analysis perform the analysis without running the program

- A **syntactic analysis** uses the code text but does not interpret statements
- A **semantic analysis** interprets statements and updates facts based on statements in the code

Static Analysis

Static analysis tools parse the code and build internal representations such as:

- **Abstract Syntax Trees (ASTs)** – represent the structure of the code.
- **Control Flow Graphs (CFGs)** – show the order in which statements execute.
- **Data Flow Graphs (DFGs)** – trace how data moves through the program

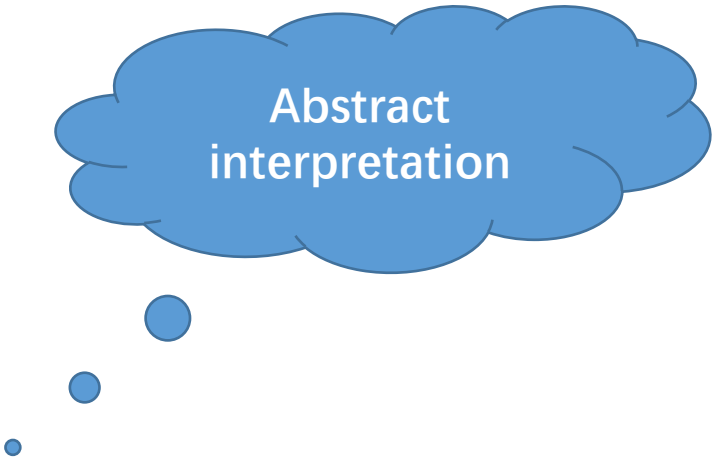
Static Analysis

Imagine you want to analyze a program without running it, to figure out things like:

Could a variable be null?

Is it always positive?

Could this code ever divide by zero?



But instead of tracking **real values**, you track **abstract values** things like "positive", "zero", "negative", or "unknown"

Static Analysis (Abstract Interpretation)

To organize and combine all these abstract values. Thats where **lattices** come in.

A **lattice** is just a structure that tells you:

What values are possible,

Which are more general or more specific,

And how to merge values from different paths.

Example

```
x = 5
if x > 0:
    x = -1
else:
    x = 0
```

Static Analysis

Soundness

A sound static analysis will not miss any real issues it reports everything that *could possibly happen*.

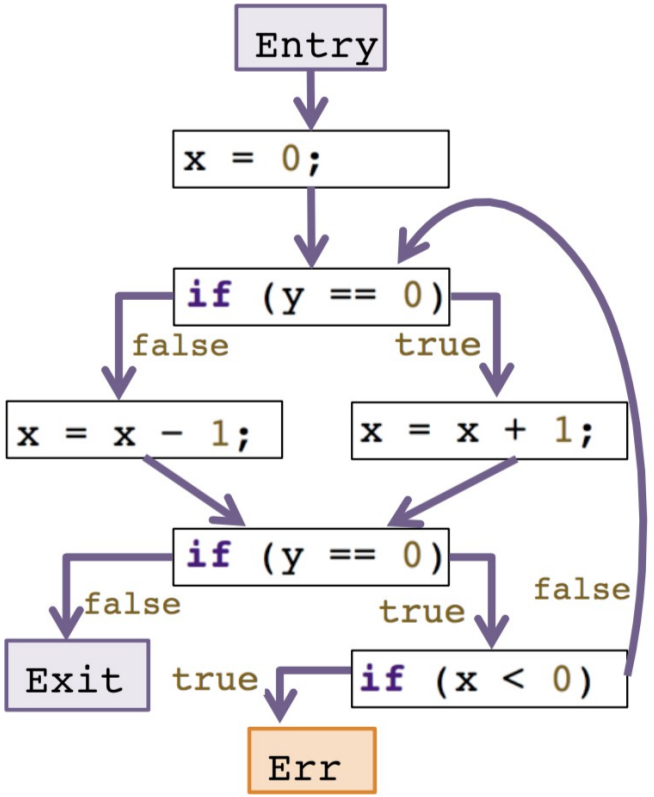
It may report **false positives** (things that aren't actual bugs) but wont miss actual problems

Completeness

A complete static analysis will never report anything that isnt a real issue so it produces no false positives.

But it may miss some actual bugs (**false negatives**).

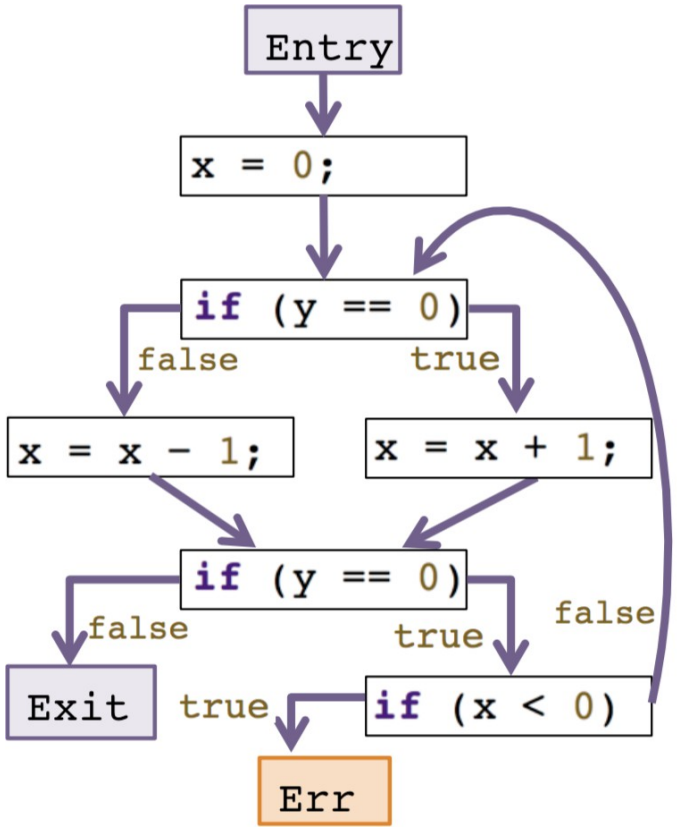
Example



How can we automatically check if the error location is reachable in this program?

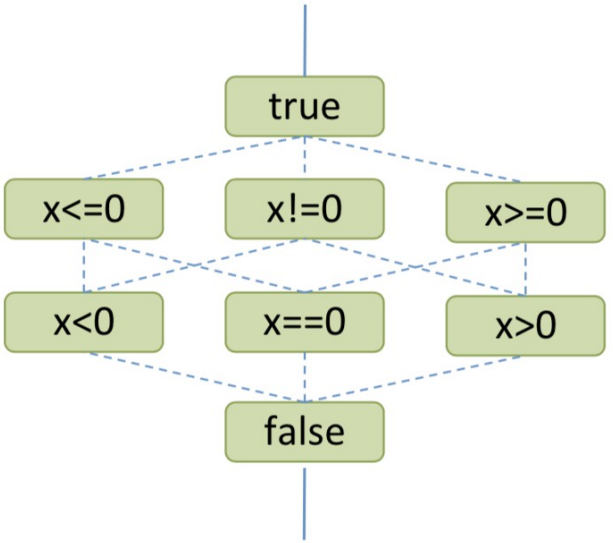
An analysis must reason about

- control flow
  - branches
  - a loop
- data
  - increment, decrement
  - comparisons with 0



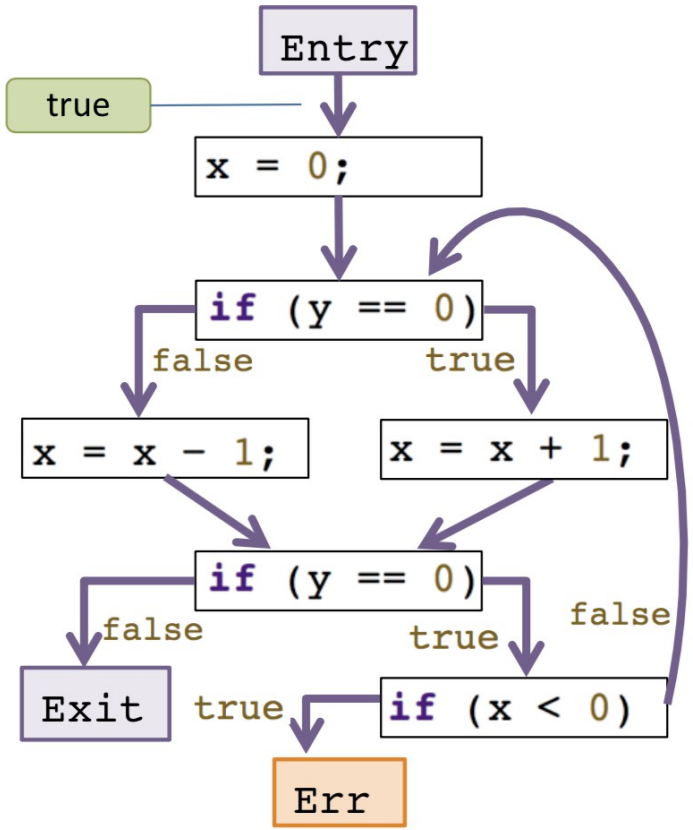
Only track relevant properties of x

x can have any value

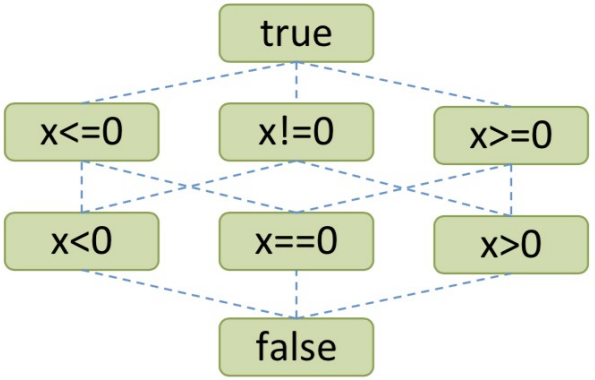


no value is feasible

Sign Analysis (1)

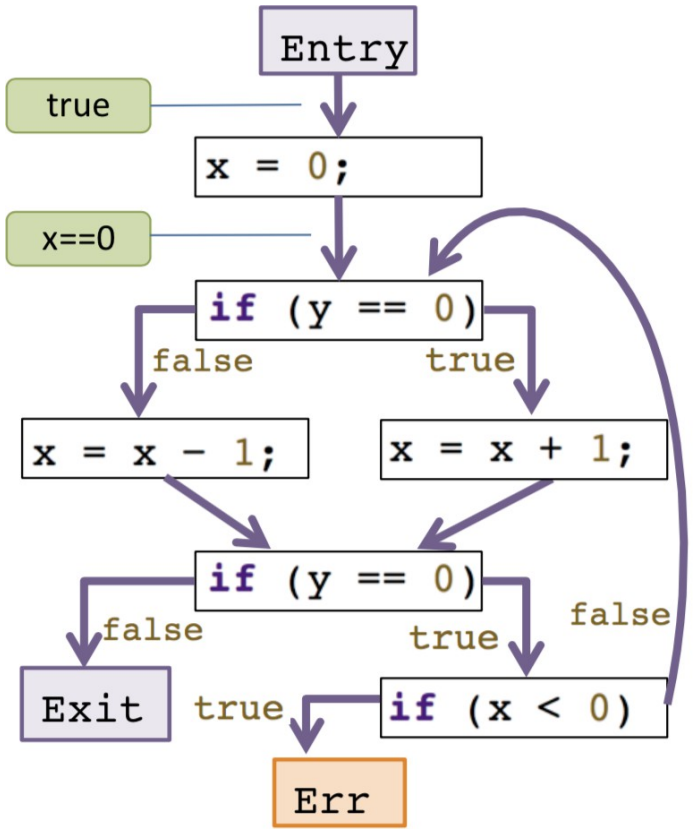


Analysis: update data about x based on control flow

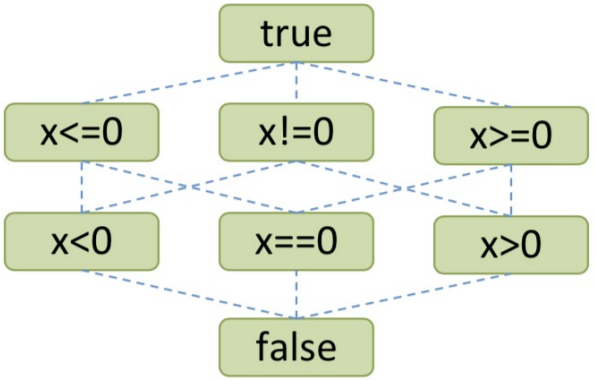


Assuming arbitrary initialization, anything can be true about x

Sign Analysis (2)

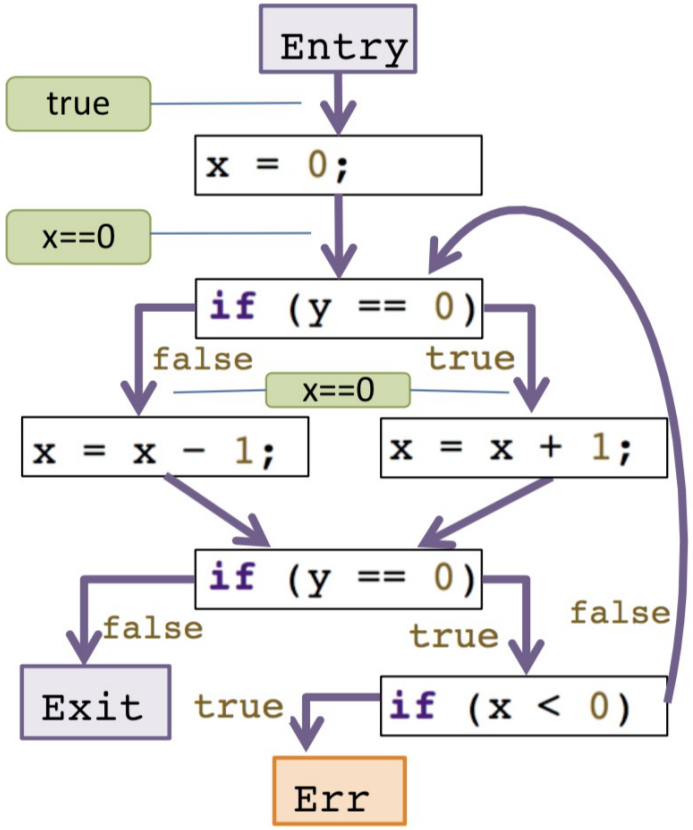


Analysis: update data about x based on control flow

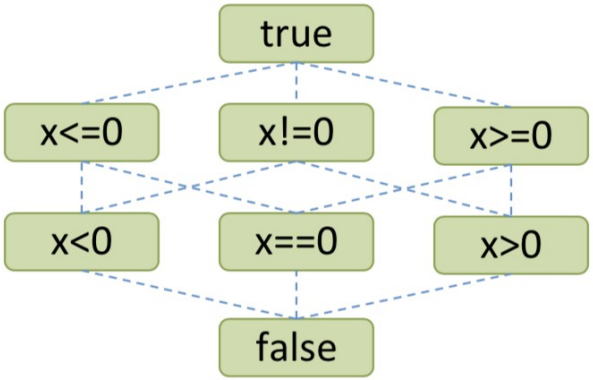


The assignment *updates* the fact about x

Sign Analysis (3)

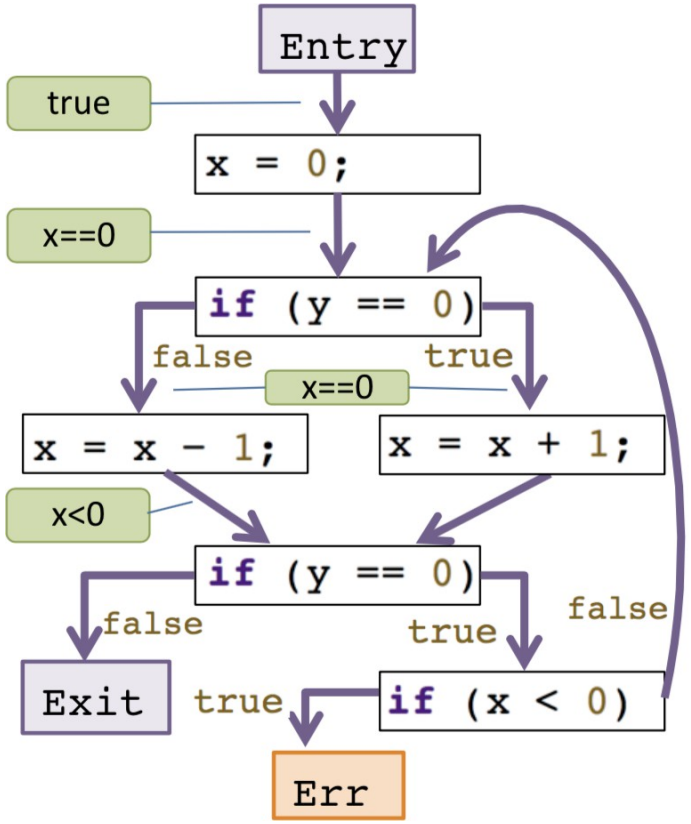


Analysis: update data about x based on control flow

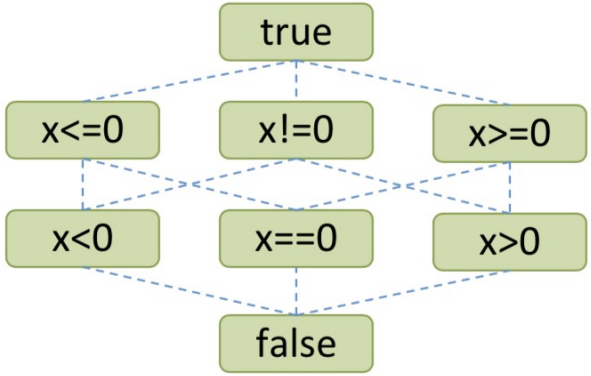


The condition does not affect x so the fact “flows through”

Sign Analysis (4)

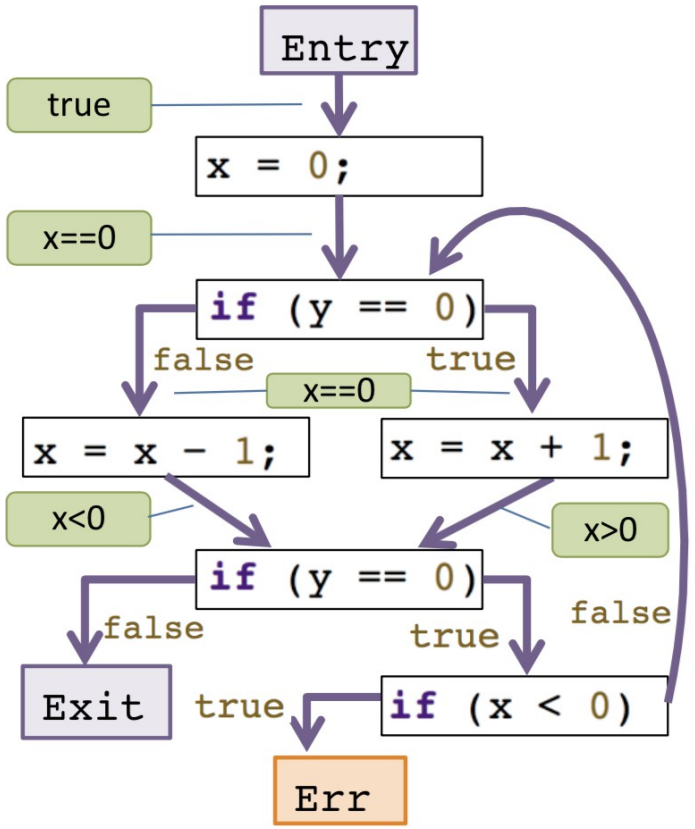


Analysis: update data about x based on control flow

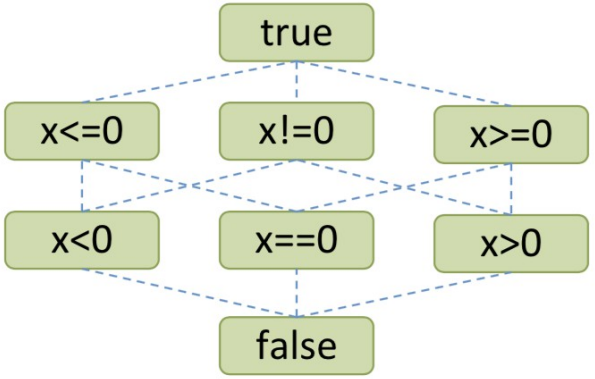


Loss of precision! We cannot write x==-1 so we *approximate* it by x<0

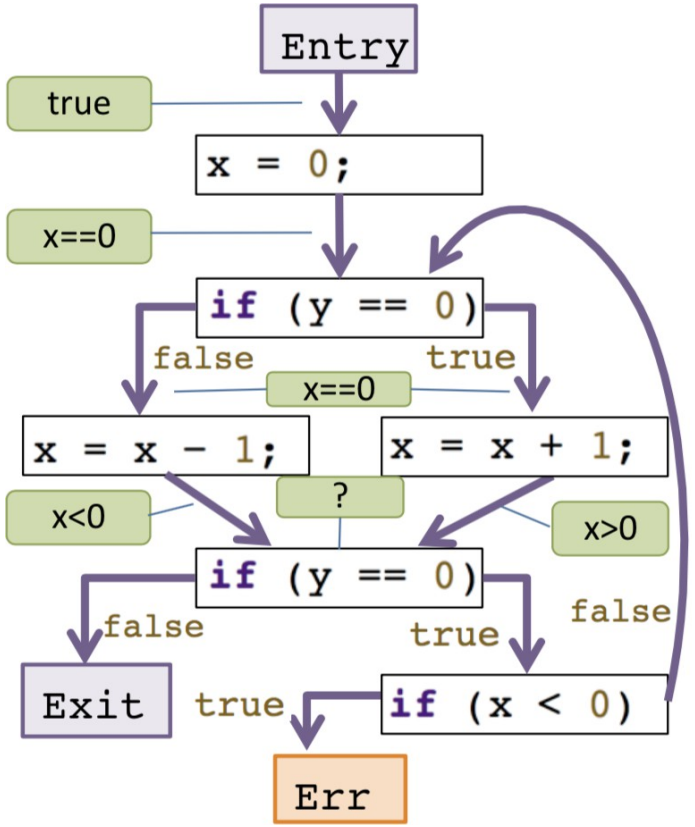
Sign Analysis (5)



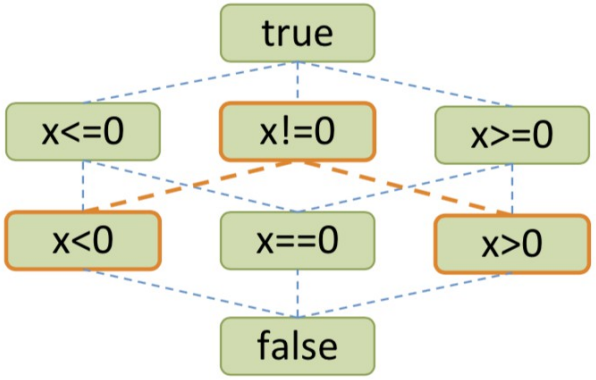
Analysis: update data about x based on control flow



Sign Analysis (6)

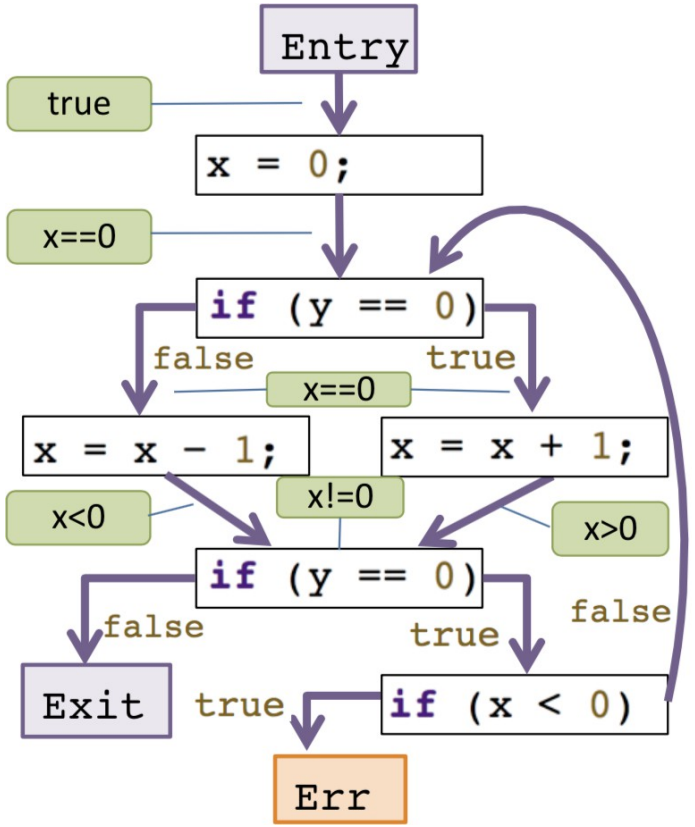


Analysis: update data about x based on control flow

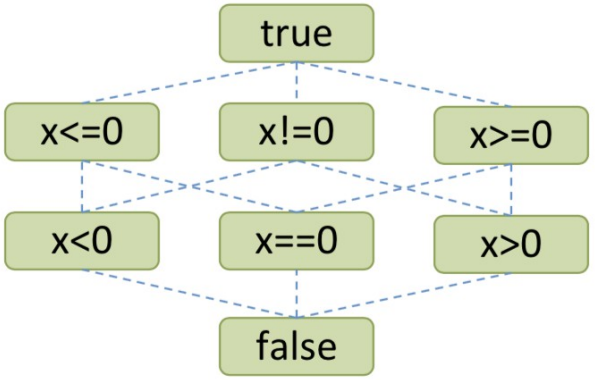


At the *join point* x is either strictly positive or strictly negative

Sign Analysis (7)

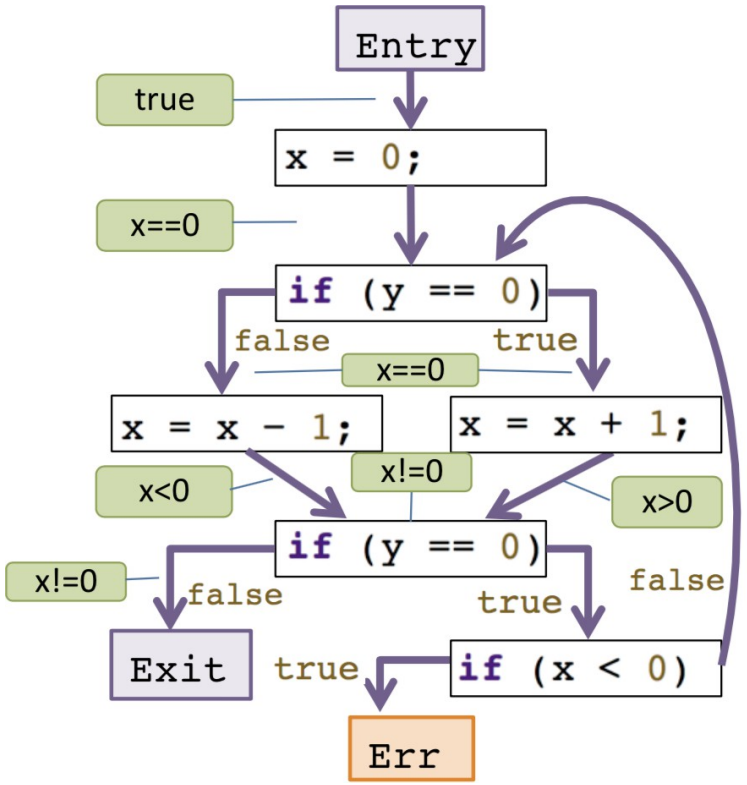


Analysis: update data about x based on control flow

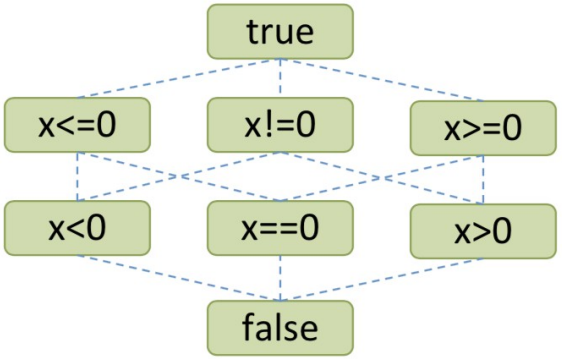


At the *join point* x is either strictly positive or strictly negative

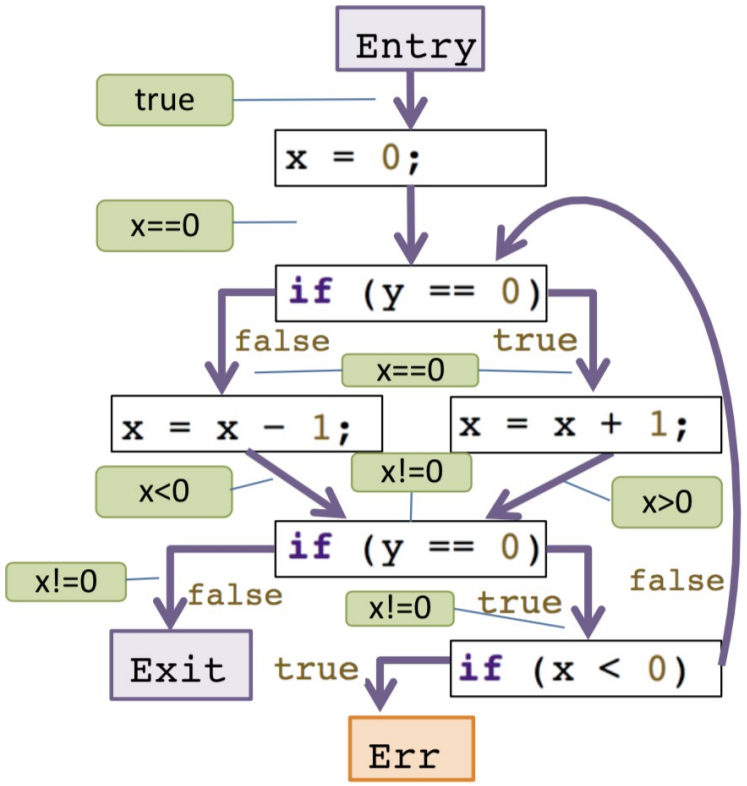
Sign Analysis (8)



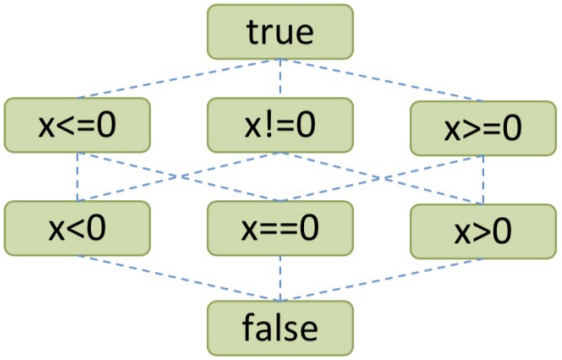
Analysis: update data about x based on control flow



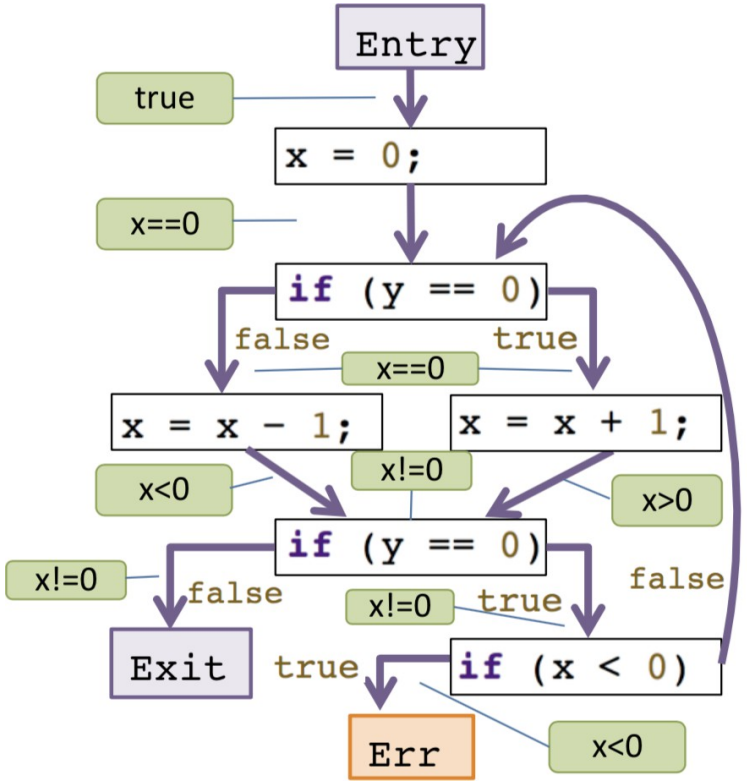
Sign Analysis (9)



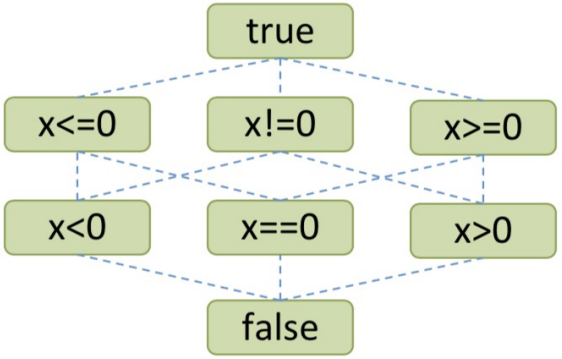
Analysis: update data about x based on control flow



Sign Analysis (10)

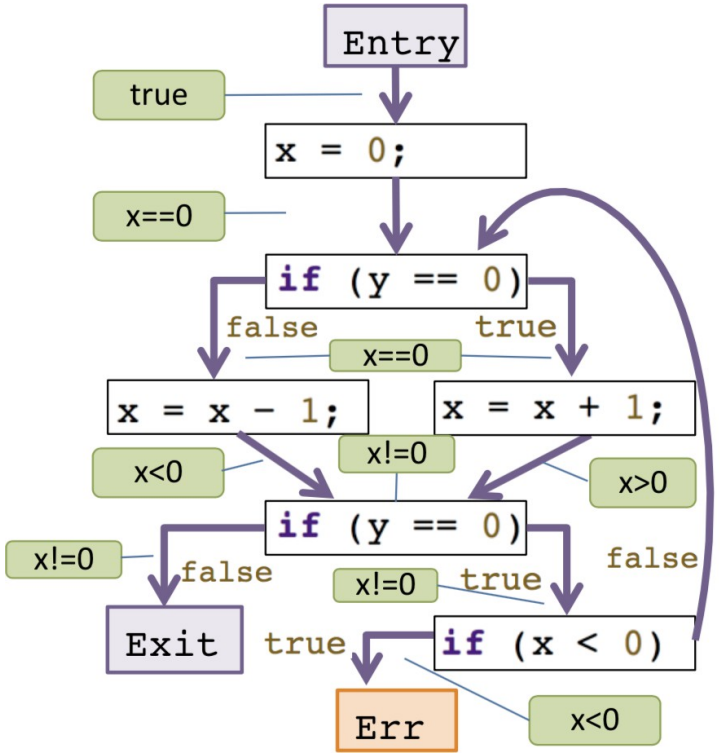


Analysis: update data about x based on control flow

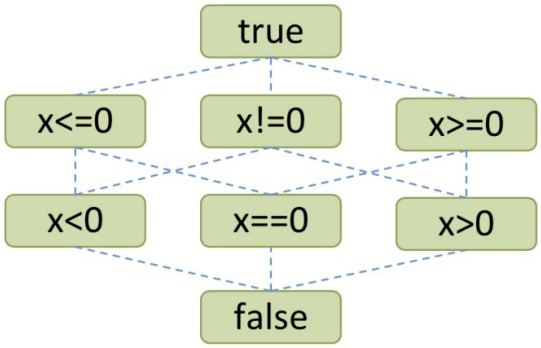


The conditional restricts x

Sign Analysis (11)



Analysis: update data about x based on control flow



The analysis concludes that it *may be possible* to reach Err with x<0

Program Verification

- Properties: true for **every** possible execution
  - Safety: nothing bad happens (e.g., buffer overflow)
  - Liveness: something good **eventually** happens
- Program verification in security
  - How to prove safety properties

Program Verification

- Approach: build up confidence on a function-by-function/module-by-module basis
- Modularity provides **boundaries** for our reasoning
  - **Preconditions**: what must hold for function to operate correctly
  - **Postconditions**: what holds after function completes
- These basically describe a contract for using the module
  - Most basic contract? Argument number and types

Example

- Memory access/dereference as a function

```
byte deref(byte *p) {
      Return *p;
}
```

- What is the precondition for the correctness of this function?

Example

- How to proof the following function won't have buffer overflow?

```
int sum(int a[], size_t n) {
  int total = 0;
    for(size_t i=0; i<n; i++)
  total += a[i];
     return total;
}
```

Example

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

Example

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access?

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

Example

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

Example

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* ?? */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires?
(3) Propagate requirement up to beginning of function

Example

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

Example

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

Example

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

Example

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

Example

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

?

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

Example

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

✓

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

Example

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

✓

Let's simplify given that the `0 <= i` part is clear.

Example

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

Example

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

Example

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

How to prove our candidate invariant?

`n <= size(a)` is straightforward because `n` never changes.

Example

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

Example

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

What about `i < n` ?

Example

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

What about `i < n` ? That follows from the loop condition.

Example

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

At this point we know the proposed invariant will always hold...

Example

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant: a != NULL &&
         0 <= i && i < n && n <= size(a) */
    total += a[i];
  return total;
}
```

… and we're done!

Questions??

zubair.ahmad@giki.edu.pk

Office: G14 FCSE lobby