Static Analysis

Chengyu Song

Slides modified from
David Wagner and Dawn Song

Finding vulnerabilities

- Dynamic analysis
  - Fuzzing
  - Symbolic execution
    - **Clang static analyzer** (https://clang-analyzer.llvm.org/available_checks.html)
- Static analysis

Bottlenecks of dynamic analysis

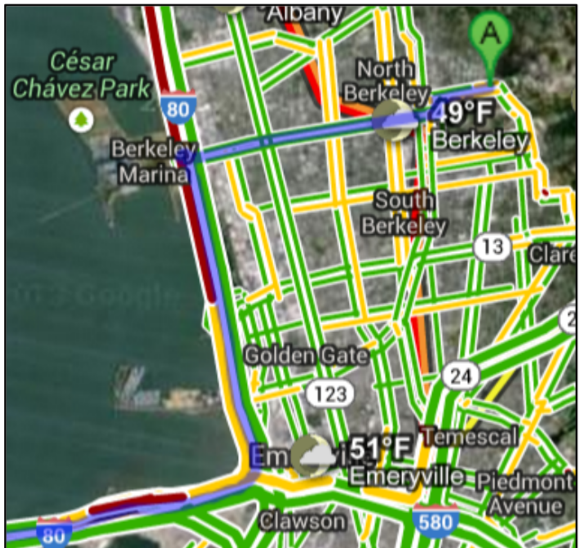
Weather

Traffic

Roads

Terrain

....



Information Overload

“Data”

Route Explosion

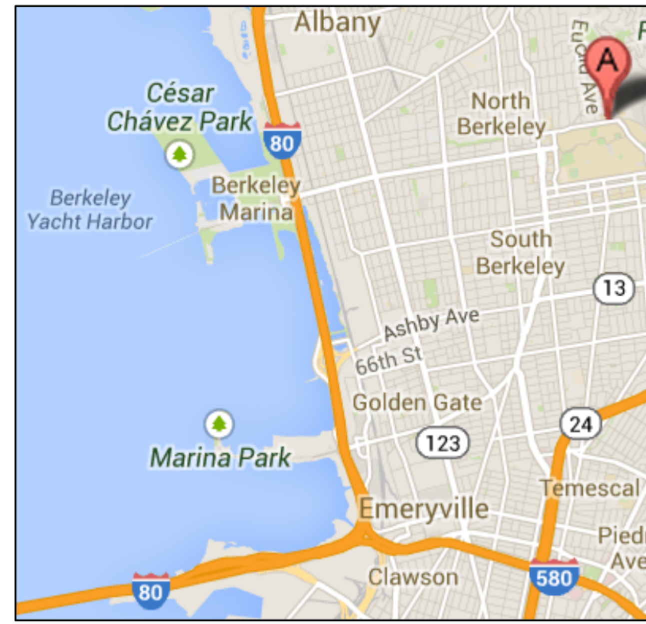
“Control”

Static analysis

Loss of information allows for more efficient computation of some answers

Static analysis algorithms operate directly on abstract representations

For example, we can analyze all possible road-routes without even sitting in a car



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

Syntactic example: optional arguments

- The system call `open()` has optional arguments

```
int open(const char *path, int oflag, ...);
```

- Typical mistake:

```
fd = open("file", O_CREAT);
```

- Result: file has random permissions
- To detect this problem: Look for `oflag == O_CREAT` without mode argument

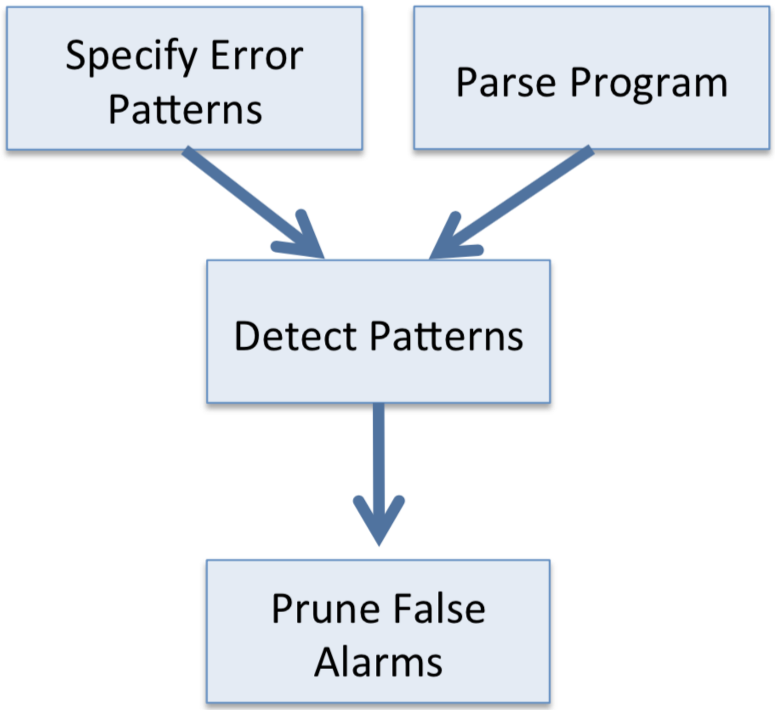
Syntactic example: name confusion

```
/*
 * javax.security.auth.kerberos.KerberosTicket, 1.5b42
 */

if (flags != null) {
   if (flags.length >= NUM_FLAGS)
        this.flags = (boolean[]) flags.clone();
   else {
        this.flags = new boolean[NUM_FLAGS];
        // Fill in whatever we have
        for (int i = 0; i < flags.length; i++)
            this.flags[i] = flags[i];
   }
} else
   this.flags = new boolean[NUM_FLAGS];

if (flags[RENEWABLE_TICKET_FLAG]) {
   if (renewTill == null)
```

source: *Squashing Bugs with Static Analysis,* William Pugh, 2006

- `flags` is a parameter, `this.flags` is a field
- Problem: check does not prevent `null` dereference
- Result: Potential Null Pointer Dereference
- Detection: find similar names on code paths where security-relevant conditions are checked

Syntactic analysis



Error patterns: Heuristically observed common error patterns in practice

Parsing: generates data structure used for error detection

Detection: match pattern against program representation

Pruning: Used to eliminate common false alarms

Error pattern types

Error Type	Examples
Typos	= vs == , &x vs. x , missing/extra semi-colons
API Usage	chroot, multiple locking, etc.
Copy-Paste	variable names/increments not updated
Identifier confusion	global and local variables, fields and parameters

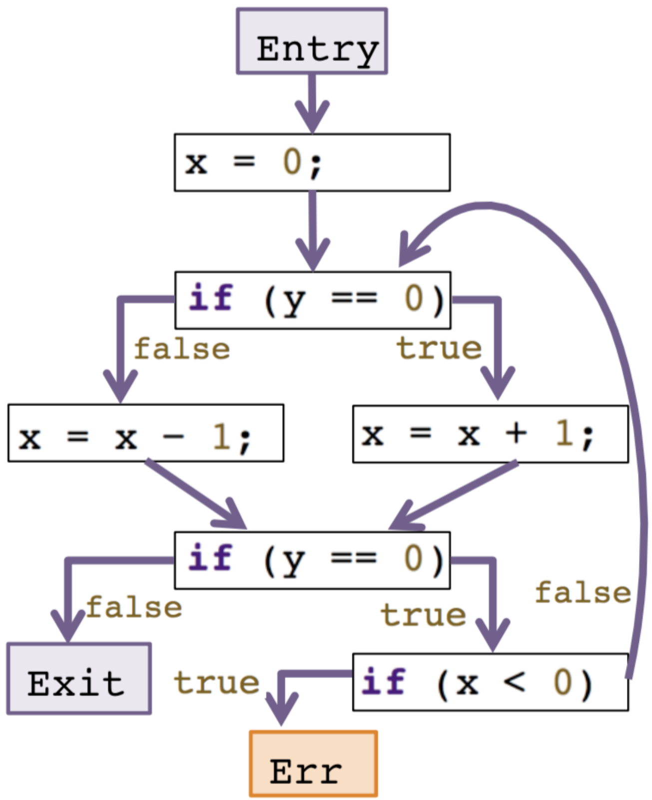
Pattern representation and detection

Representation	Types of Algorithms
String	Subsequence mining, edit distance, matching
Parse Tree	Pattern matching,
Control Flow Graphs	Automata algorithms, sub-graph isomorphism

Semantic analysis

- Interpret statements and updates facts
    - How to abstract data
    - How to handle control

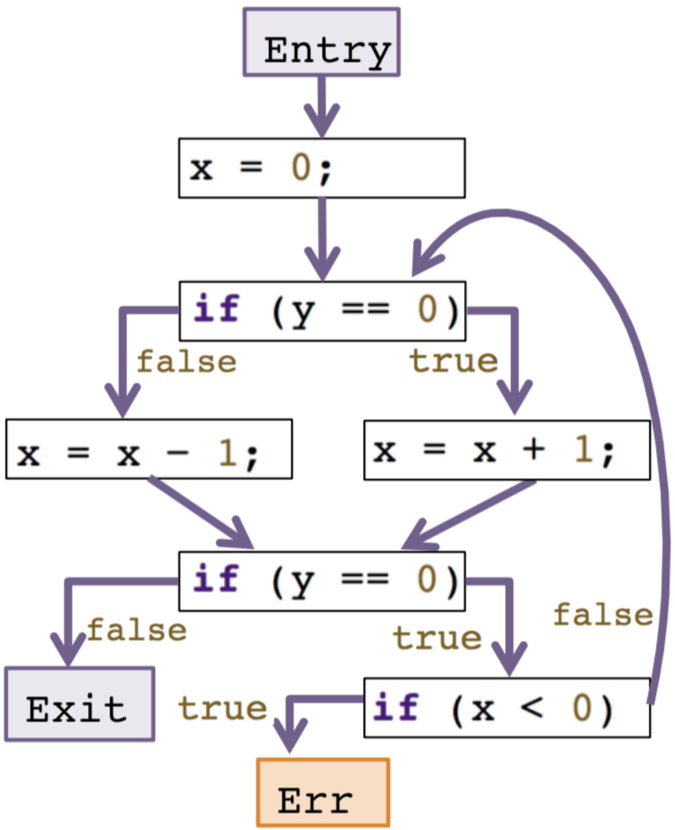
Example



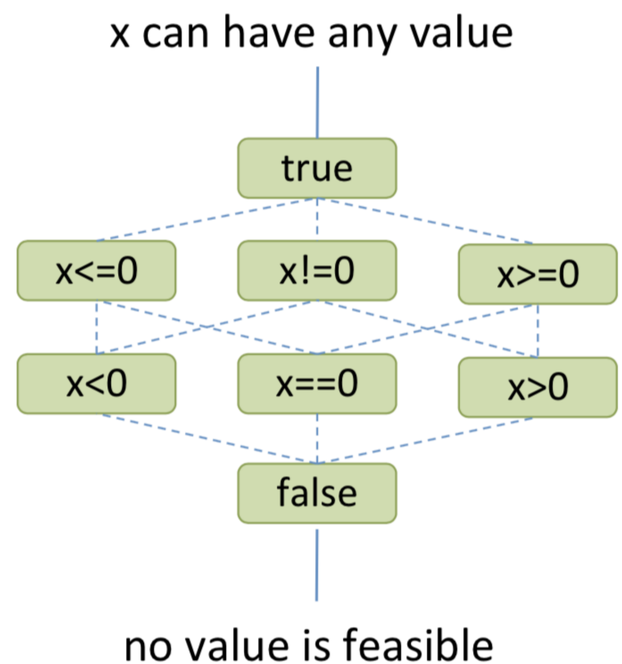
How can we automatically check if the error location is reachable in this program?

An analysis must reason about
- control flow
  - branches
  - a loop
- data
  - increment, decrement
  - comparisons with 0

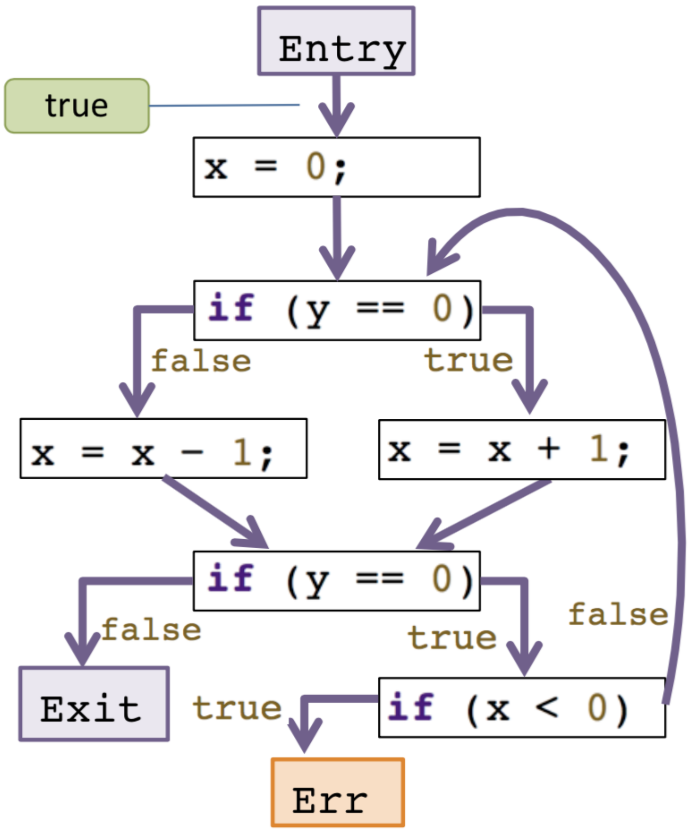
Abstracting data



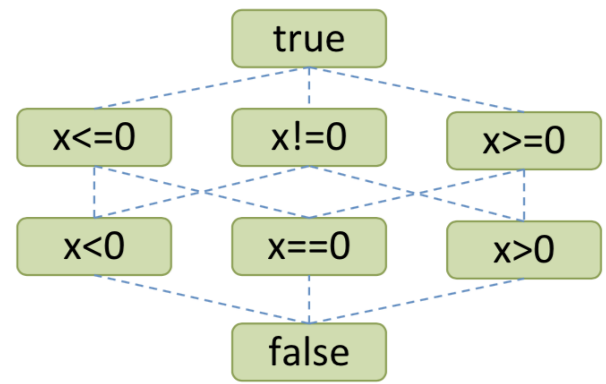
Only track relevant properties of x



Sign analysis (1)

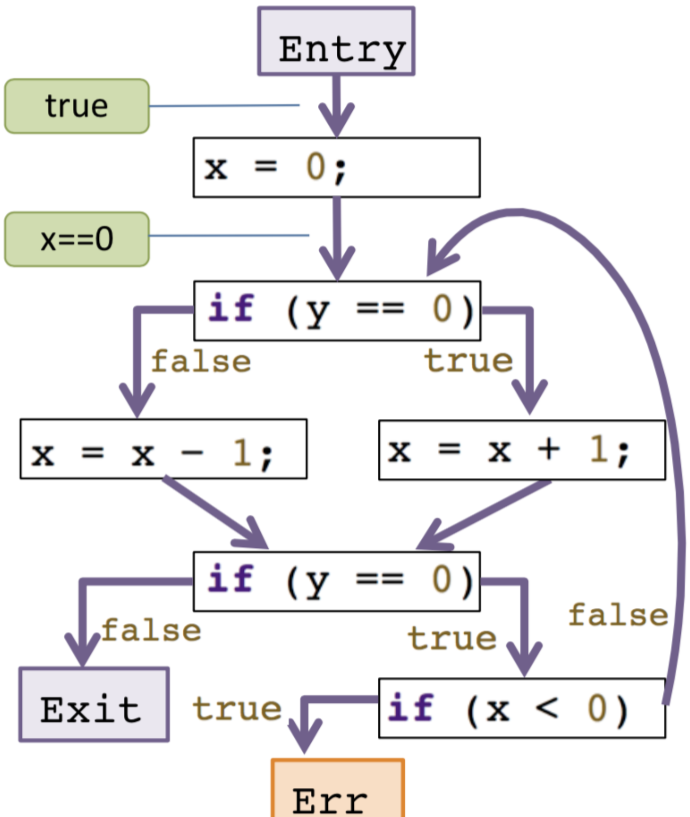


Analysis: update data about x based on control flow

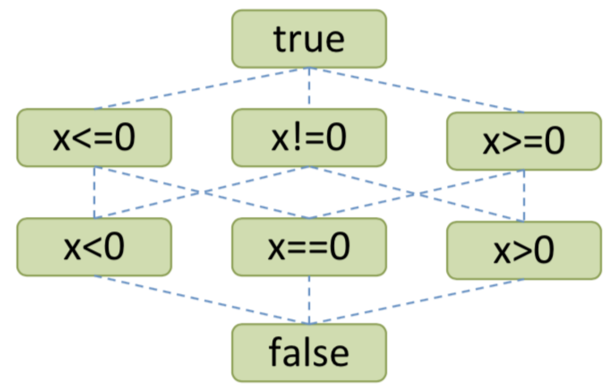


Assuming arbitrary initialization, anything can be true about x

Sign analysis (2)

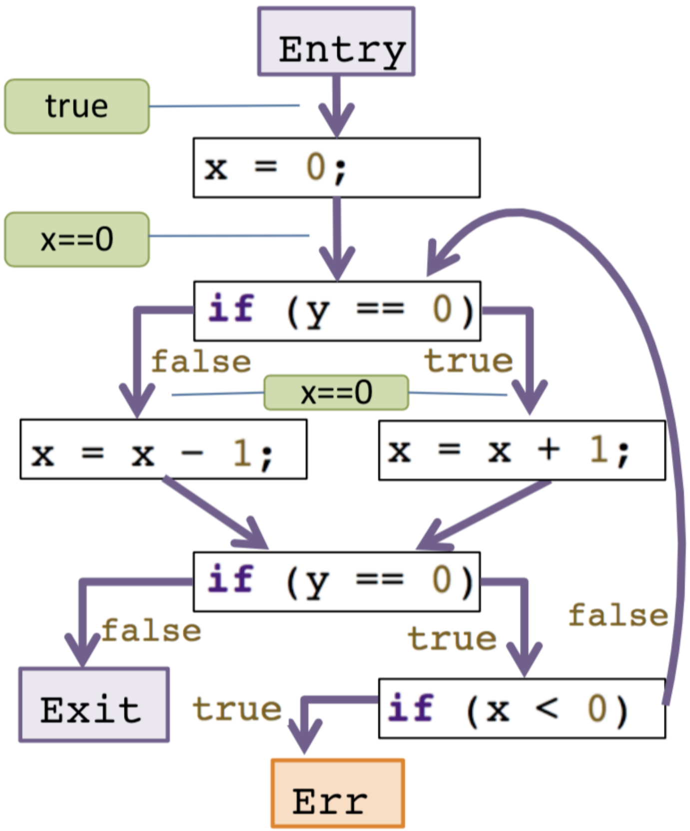


Analysis: update data about x based on control flow

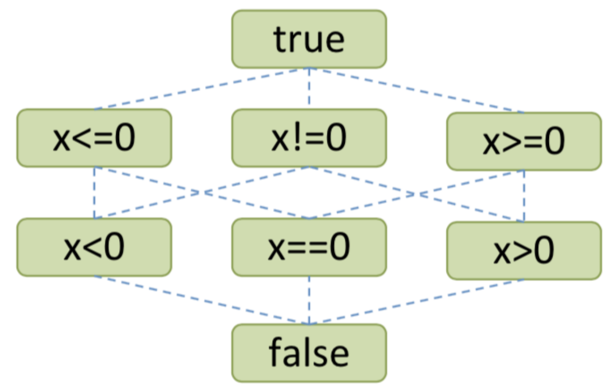


The assignment *updates* the fact about x

Sign analysis (3)

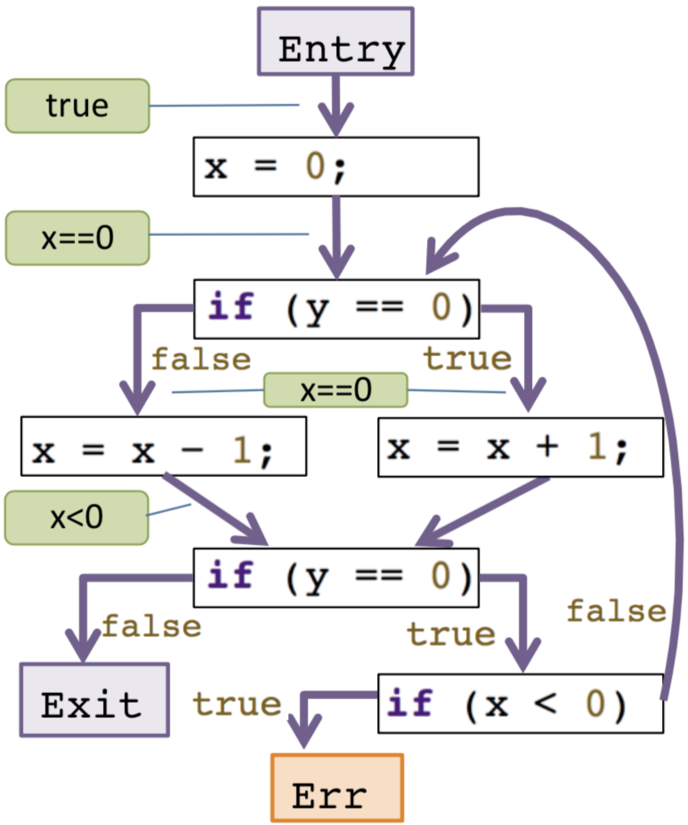


Analysis: update data about x based on control flow

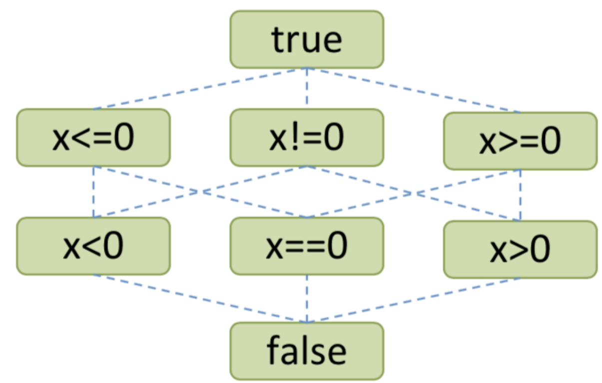


The condition does not affect x so the fact “flows through”

Sign analysis (4)

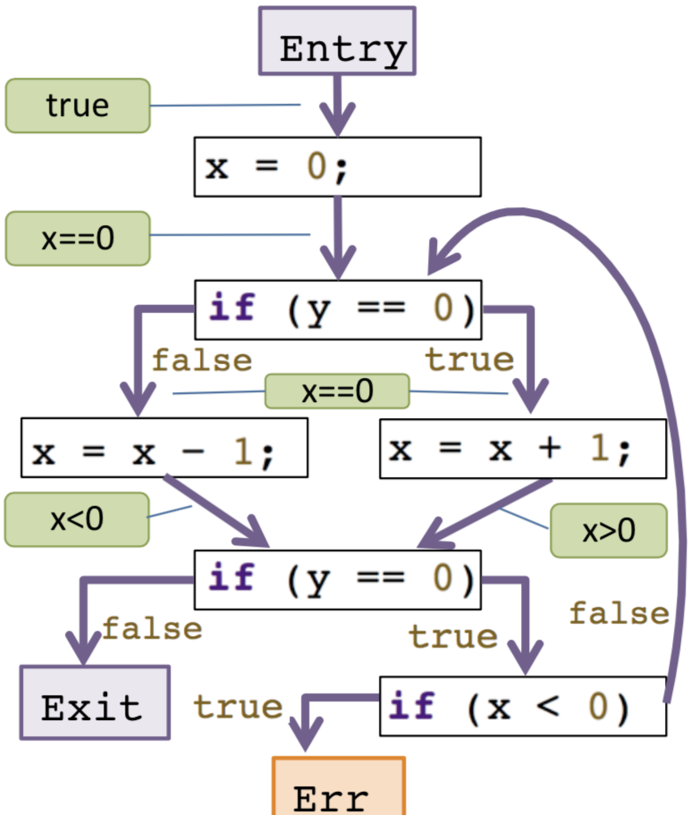


Analysis: update data about x based on control flow

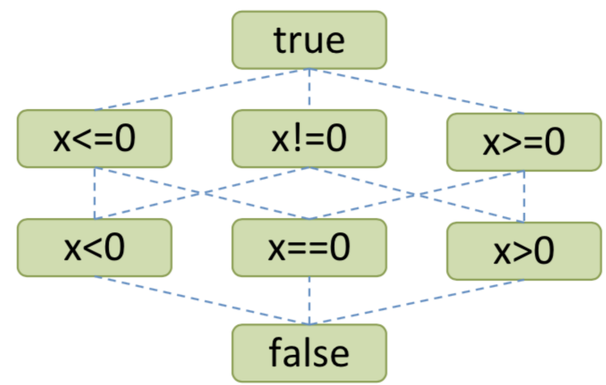


Loss of precision! We cannot write x==-1 so we *approximate* it by x<0

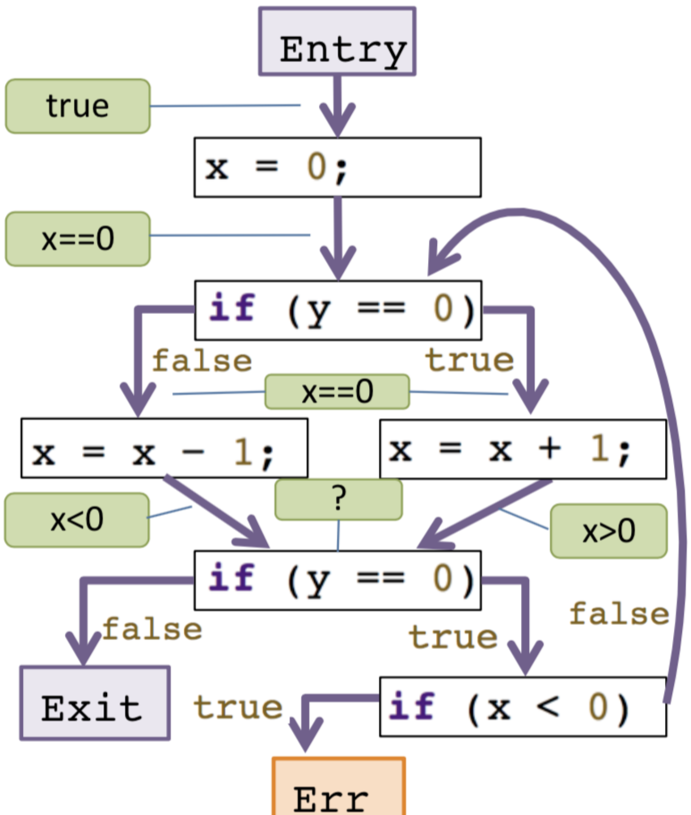
Sign analysis (5)



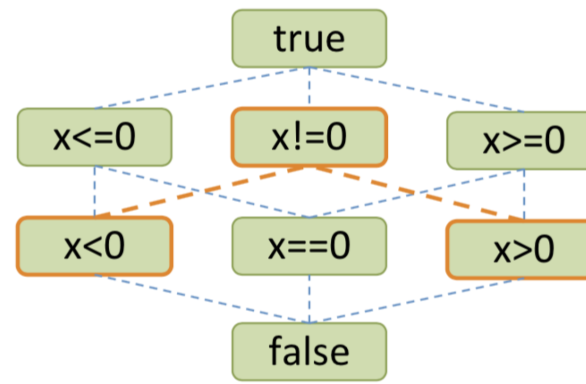
Analysis: update data about x based on control flow



Sign analysis (6)

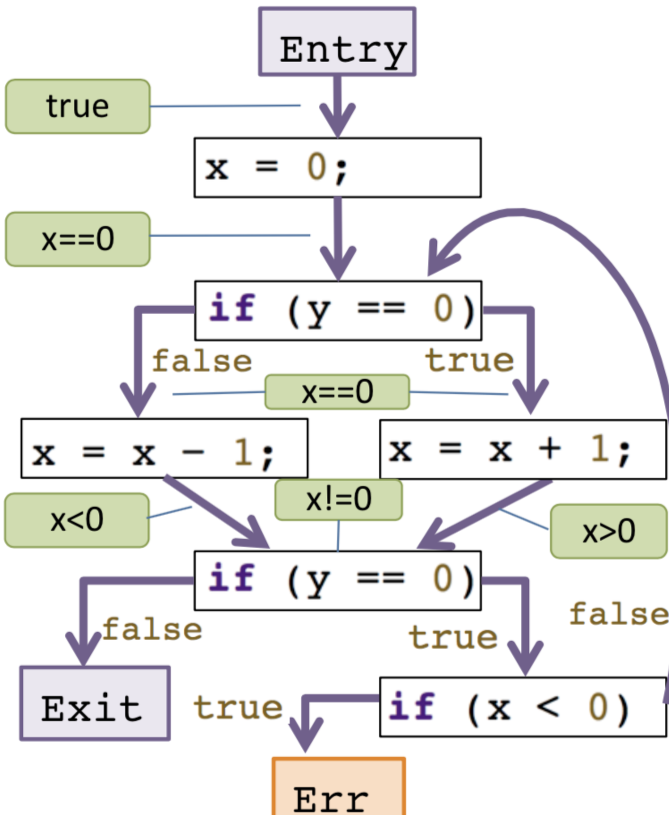


Analysis: update data about x based on control flow

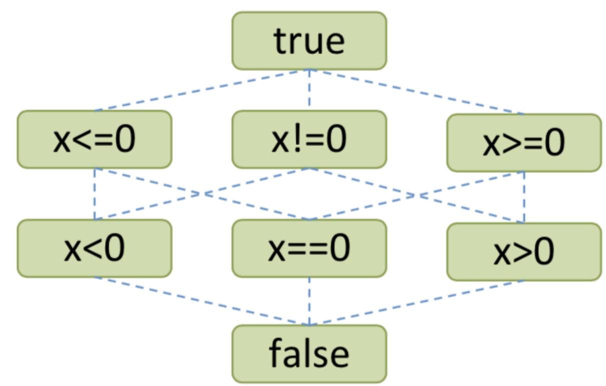


At the *join point* x is either strictly positive or strictly negative

Sign analysis (7)

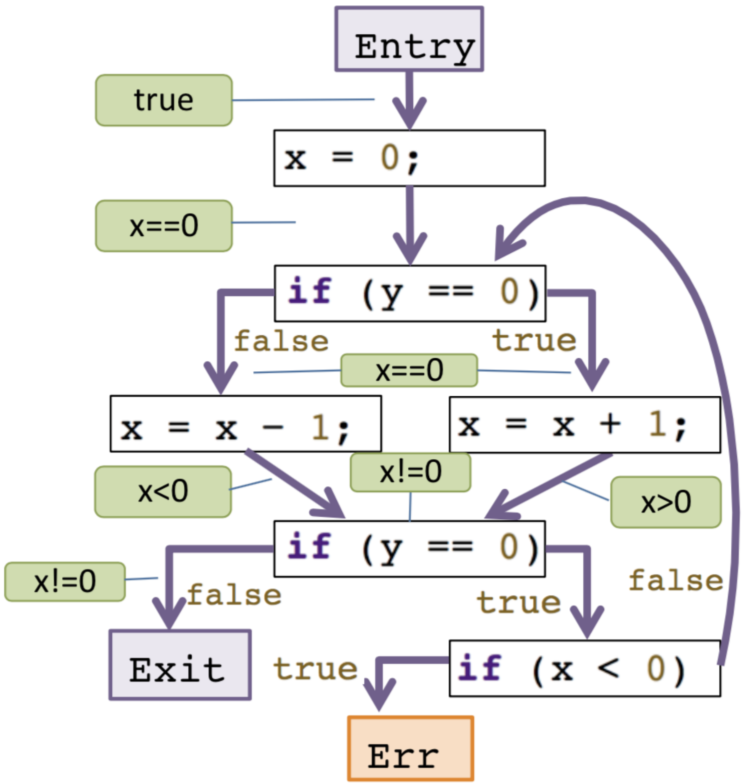


Analysis: update data about x based on control flow

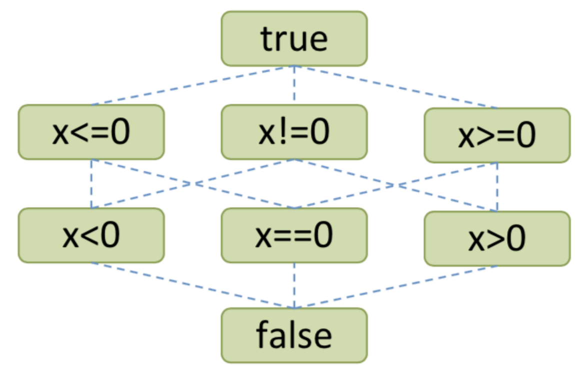


At the *join point* x is either strictly positive or strictly negative

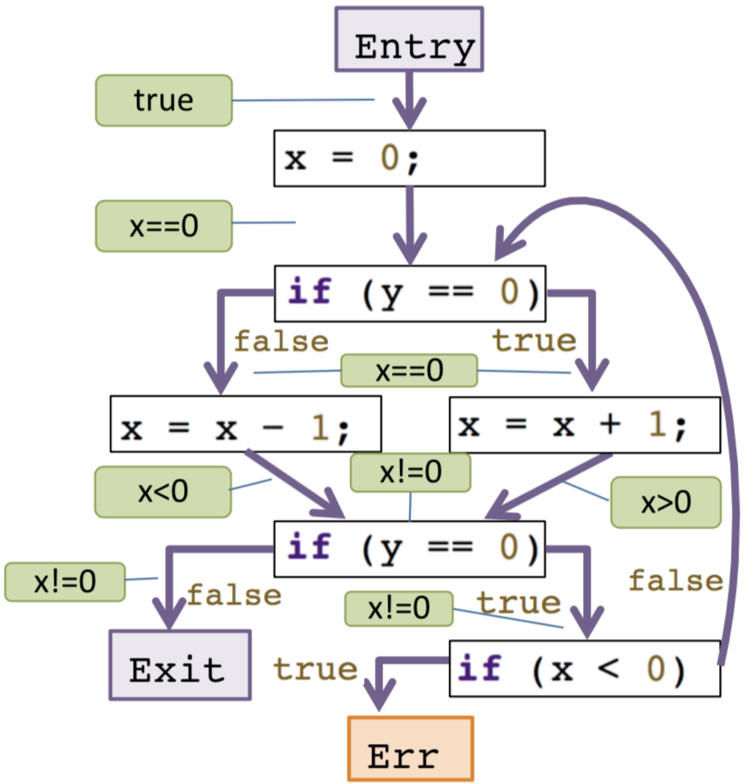
Sign analysis (8)



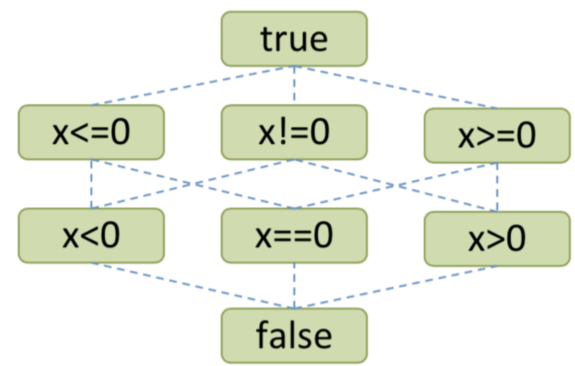
Analysis: update data about x based on control flow



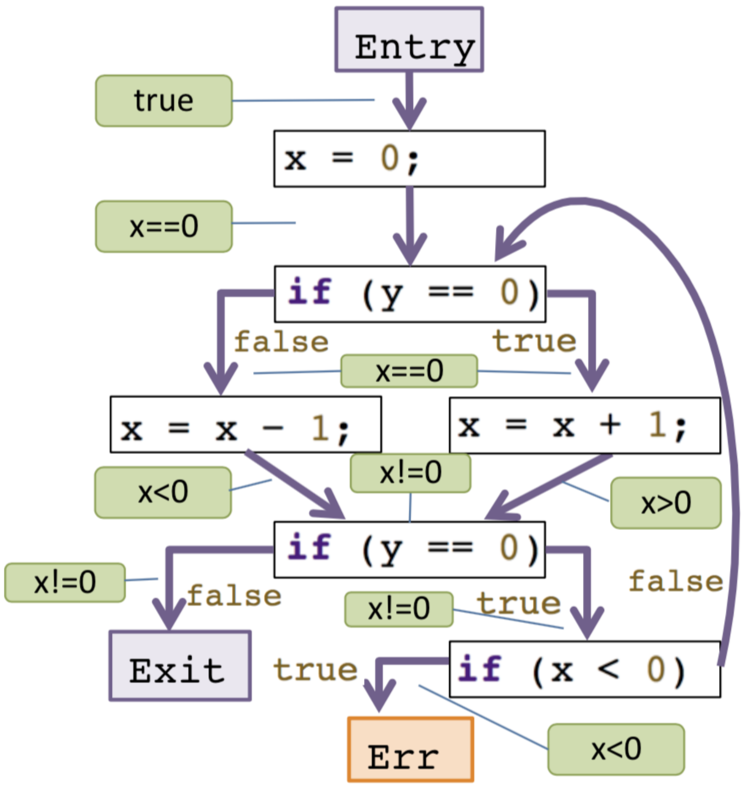
Sign analysis (9)



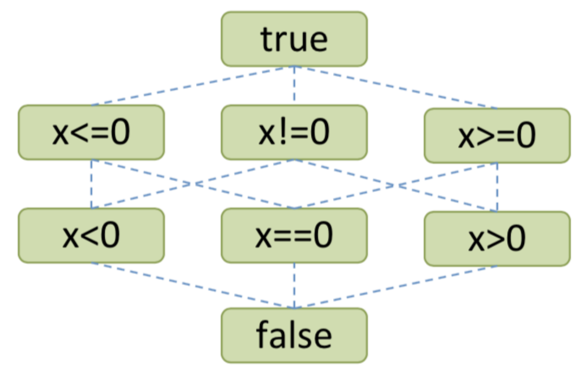
Analysis: update data about x based on control flow



Sign analysis (10)

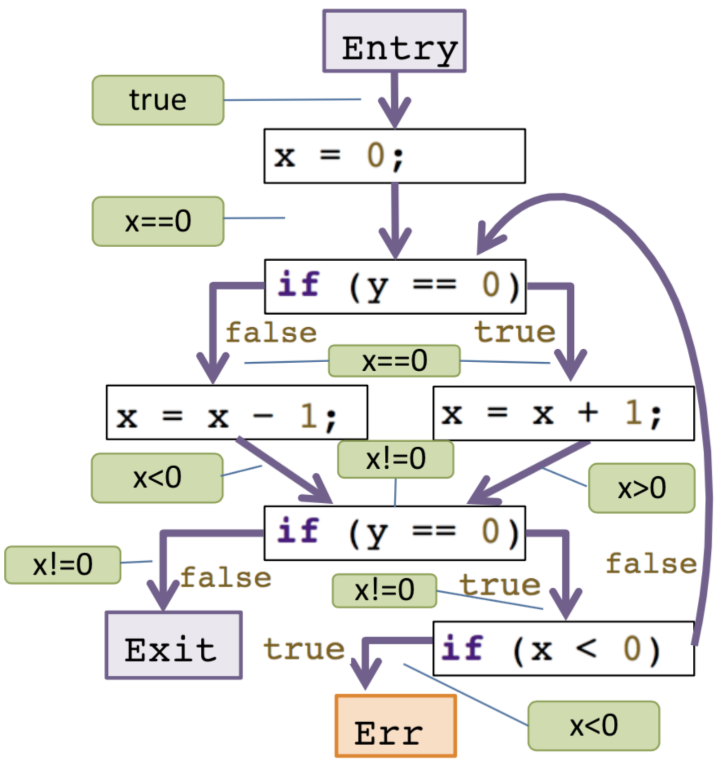


Analysis: update data about x based on control flow

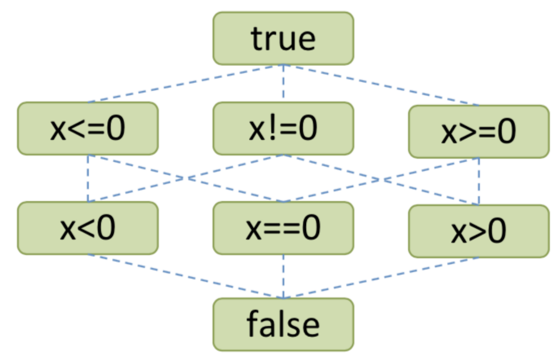


The conditional restricts x

Sign analysis (11)



Analysis: update data about x based on control flow



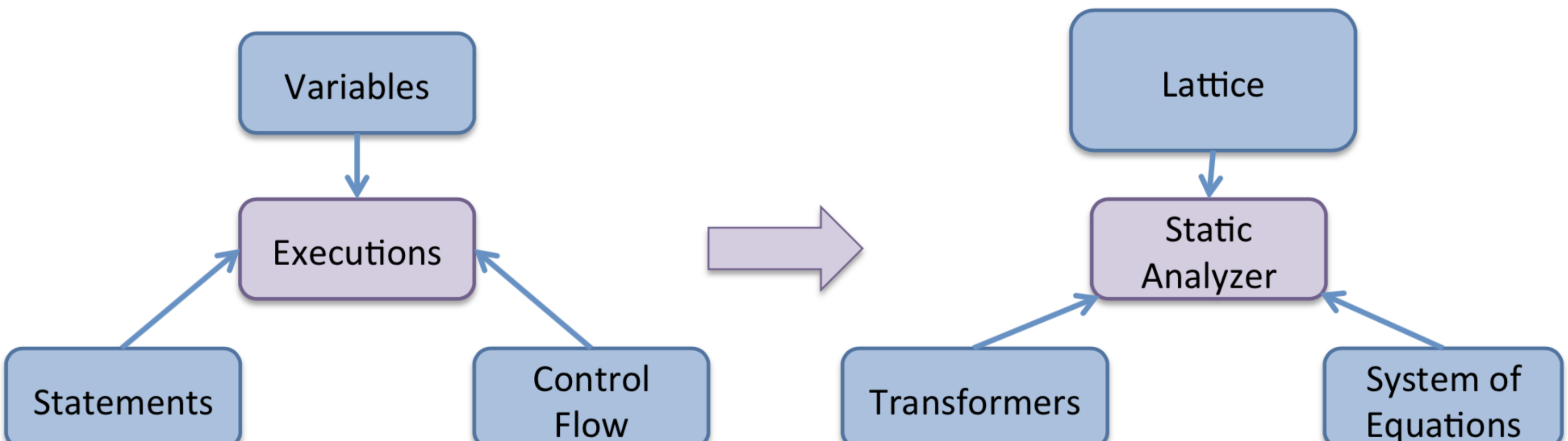
The analysis concludes that it *may be possible* to reach Err with x<0

Static analysis vs. symbolic execution

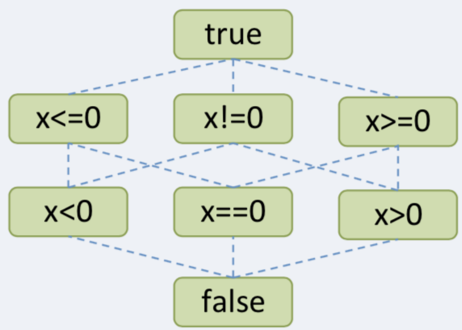
- Data was not precisely represented
- Some variables were ignored
- Control flow paths were joined
- It is not clear if there is an error
- It is not clear which path leads to the error

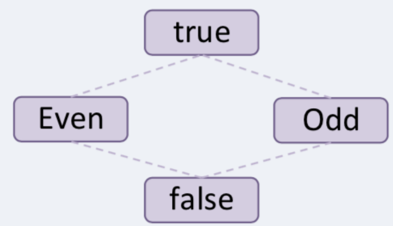
Architecture of static analysis

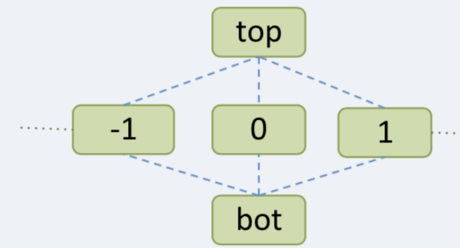
The behavior of a program can be approximated by separately approximating variable values, statements and control flow.



Lattices in static analysis





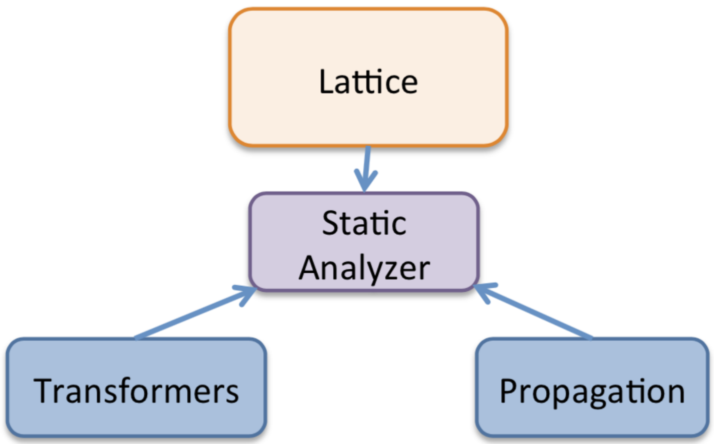


Signs
- positive/negative/zero
- cannot represent non-zero values
- no relationships between variables

Parity
- even or odd
- cannot represent values
- no relationships between variables

Constants
- a single value
- cannot represent more values: x==3||x==4
- no relationships between variables

Lattices in static analysis (cont.)



A lattice is a set with

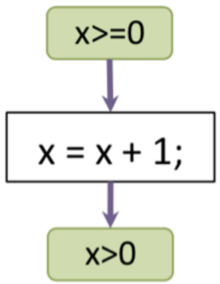
- a *partial order* for comparing elements
- a least upper bound called *join*
- a greatest lower bound called *meet*

In static analysis

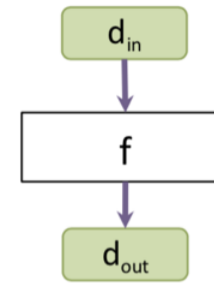
- lattice elements abstract states
- order is used to check if results change
- meet and join are used at branch and join points

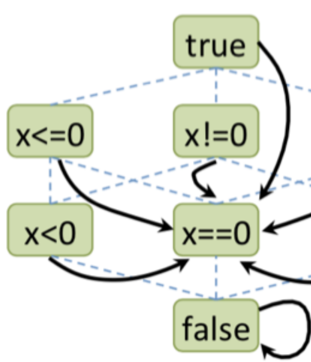
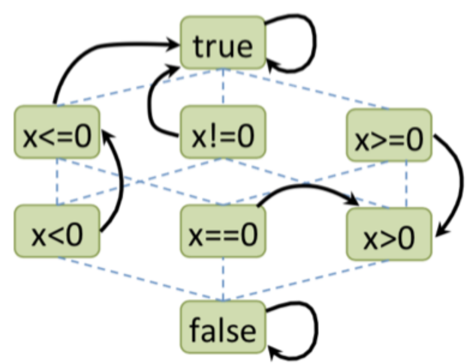
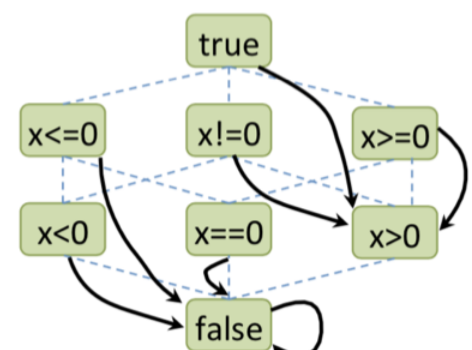
Most analyses use only meet or only join

Transforms in static analysis



A *transformer* (or *transfer function*) describes how a statement modifies lattice elements



`x = 0;`	`x = x+1;`	`if (x > 4)`
		

Information flow analysis

- How information propagates in software
  - Taint analysis (2 states lattice, tainted, not-tainted)
  - Source: where tainted data is introduced
  - Sink: where tainted data should not be used
  - Cleanser/sanitizer: where tainted -> not tainted

Taint analysis: application

- Privacy leak in Android apps
- Use of untrusted data
    - Format string from Internet
    - Memory from user space
    - Command/SQL injection attacks (more in web session)
- Uninitialized data

Take away

- Static analysis
  - No execution of the program
  - Analyzes all the code
  - Use abstraction (loss of precision) to scale (coverage)
  - Has false positives (may be a bug)

Soundness and completeness

- Soundness: if the program contains an error, the analysis will report an error.
    - "Sound for reporting correctness"
- Completeness: if the analysis reports an error, the program will contain an error.
    - "Complete for reporting correctness"

Note: these terms have different meaning in other contexts

Soundness and completeness (cont.)

	Complete	Incomplete
Sound	Report all errors Report no false alarms **UNDECIDABLE** (Ex: manual verification)	Report all errors May report false alarms (Ex: Abstract interpretation)
Unsound	May not report all errors Report no false alarms (Ex: symbolic execution)	May not report all errors May report false alarms (Ex: Syntactic analysis)

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

How to reason about safety

- 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

Functions in verification

- Mathematical function: `f(x) -> y`
- Individual statement can be considered as a function
  - Preconditions: what must hold for correctness of the statement
  - Postcondition: what holds after execution of the statement
  - Stmt #1's postcondition should logically imply Stmt #2's precondition
- **Invariants**: conditions that always hold at a given point in a function

Memory safety

- Memory access/dereference as a function

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

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

Memory safety (cont.)

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

```
/* p != NULL &&
   p does not point to freed object &&
   p does not point to unintialized memory &&
   p is with the upper and lower bounds */
byte deref(byte *p) {
  return *p;
}
```

Verification (1)

- Proving precondition -> postcondition
- Given preconditions and postconditions
  - Specifying what obligations caller has (precondition) and what callers are entitled to rely upon (postcondition)
- Verify: no matter how function is called
  - If precondition is met at function's entrance
  - then postcondition is guaranteed to hold upon function's return

Verification (2)

- Basic idea:
  - Write down a precondition and postcondition for every line of code
  - Use logical reasoning

Verification (3)

- Requirement
    - Each statement's postcondition must match (imply) precondition of any following statement
    - At every point between statements, write down *invariants* that must be true at that point
        - Invariant is postcondition for preceding statement, and precondition for next one

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

```
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;
}
```

Let’s simplify, given that `a` never changes.

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;
}
```

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!

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;
}
```

A more complicated loop might need us to use *induction*:

Base case: first entrance into loop.

Induction: show that *postcondition* of last statement of loop plus loop test condition implies invariant.

Summary

- Software security: **vulnerabilities**
  - Exploits: the most popular way of getting attacked, including malware
  - Memory vulnerabilities: root causes, how to exploit, defense mechanisms
  - How to find vulnerabilities: fuzzing, symbolic execution, static analysis, verification
  - Other vulnerabilities?
    - In future sessions