Exam 2 Schedule

- Wednesday, Dec. 13, 11:00am-1pm
- ECSN 2.120

Exam 2 Questions

- True/false
- Multiple choice (single/multiple answer choice)
- Discussion

Exam 2 Groundrules

- No cell phones or laptops
- Closed book
- Bring a writing implement
- Remember to bring your Comet ID

Course Topics

- Software development process
- Software requirements engineering
- Architecture & design patterns
- Implementation & coding styles
- Software refactoring
- Software testing & debugging
- Software security

Learning Outcomes

- Ability to
  - understand software lifecycle development models
  - understand and apply software requirements engineering techniques
  - understand and apply software design principles
  - understand and apply software testing techniques
  - understand the use of metrics in software engineering
  - understand formal methods in software development
  - establish and participate in an ethical software development team
  - understand software project management
  - understand CASE tools for software development

CE/CS/SE 3354
Software Engineering

Software Refactoring

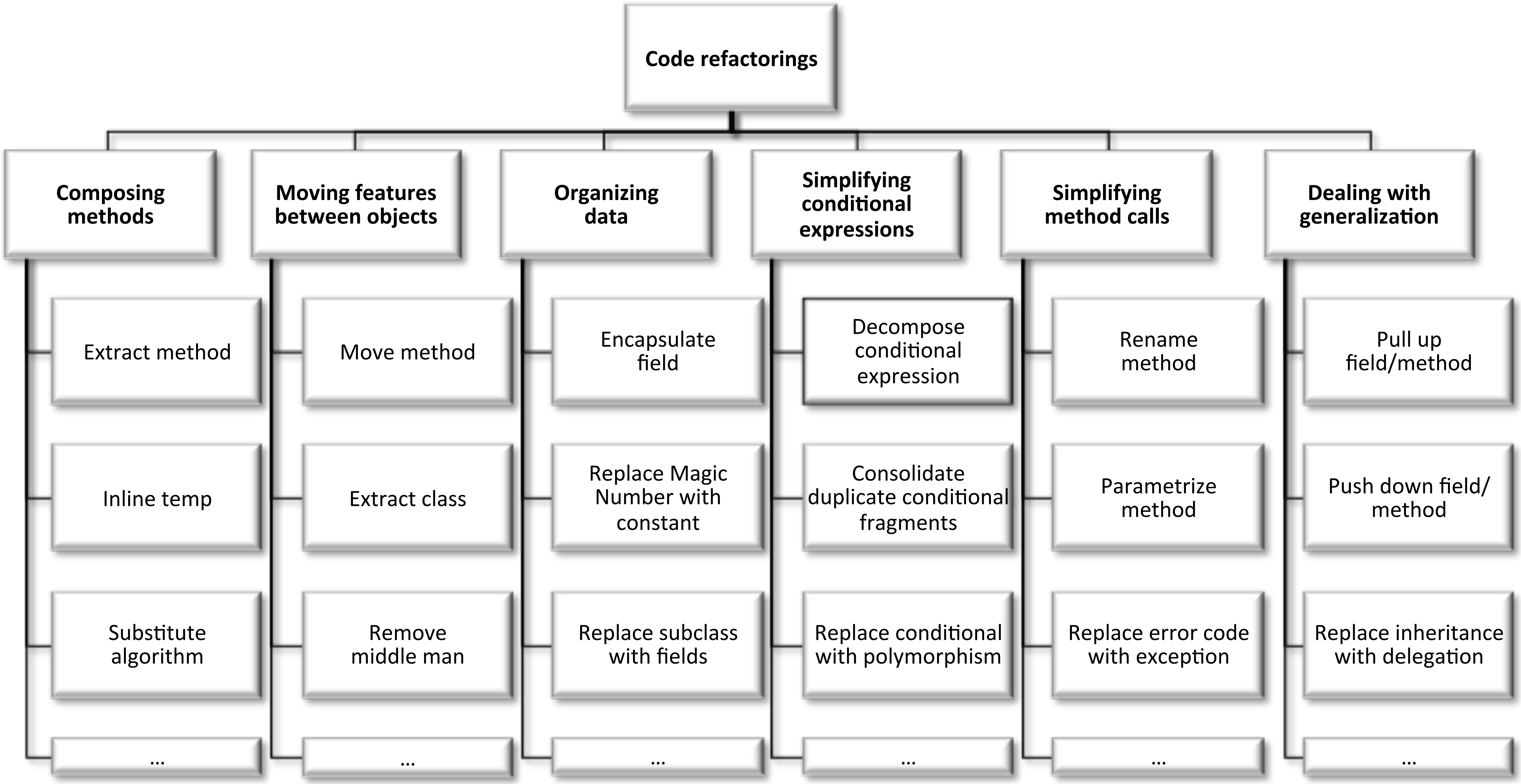
Refactoring definitions

- A program restructuring operation to support the design, evolution, and reuse of object oriented frameworks that preserve the behavioral aspects of the program [1]
- A process of changing the internal structure of software, not its observable behavior, in order to improve its internal quality [2]

[1] Opdyke, William F. ***Refactoring object-oriented frameworks***. Diss. University of Illinois at Urbana-Champaign, 1992.

[2] Fowler, M., **Refactoring: Improving the Design of Existing Code,** Addison-Wesley, 1999

A refactorings catalog [1]



[1] Fowler, M., **Refactoring: Improving the Design of Existing Code,** Addison-Wesley, 1999.
refactoring.com

Homework II: Q2

```java
public abstract class Party {
}

public class Person extends Party {
    private String name;
    private Date dob;
    private String nation;

    public void printNameAndDetails(){
        System.out.println("Name:  " + name);
        System.out.println ("Details: DOB-" + dob.toString() + ", Nation-" + nation);
    }
}

public class Company extends Party {
    private String name;
    private Date incorporated;

    public void printNameAndDetails(){
        System.out.println ("Name:  " + name);
        System.out.println ("Details: Incorporated-" + incorporated.toString());
    }
}
```

Homework II: Q2

```
public class Person extends Party {
    private String name;
    private Date dob;
    private String nationality;

    private void printNameAndDetails(){
        printName();
        printDetails();
    }
    private void printName() {
        System.out.println("Name: " + name);
    }
    private void printDetails() {
        System.out.println("DOB: " + dob.toString() + ", Nationality" + nationality);
    }
}
```

Homework II: Q2

```
public class Company extends Party {
    private String name;
    private Date incorporated;

    public void printNameAndDetails(){
        printName();
        printDetails();
    }
    private void printName() {
        System.out.println("Name:  " + name);
    }
    private void printDetails() {
        System.out.println("Incorporated:  " + incorporated.toString());
    }
}
```

Homework II: Q2

```
public abstract class Party {
    private String name;
    public void printNameAndDetails(){
        printName();
        printDetails();
    }
    private void printName() {
        System.out.println("Name: " + name);
    }
    abstract protected void printDetails();
}
public class Person extends Party {
    private Date dob;
    private String nationality;
    protected void printDetails() {
        System.out.println("DOB: " + dob.toString() + ", Nationality" + nationality);
    }
}
public class Company extends Party {
    private Date incorporated;
    protected void printDetails() {
        System.out.println("Incorporated: " + incorporated.toString());
    }
}
```

Introduce null object

- You have repeated checks for a null value
- Replace the null value with a null object that exhibits the default behavior

Introduce null object - Example: before refactoring

```
class OtherClass {

    Site site = new Site()
    //other fields...

    void method() {

        // billing plan
        Customer customer = site.getCustomer();
        BillingPlan plan;
        if (customer == null)
            plan = BillingPlan.basic();
        else
            plan = customer.getPlan();

        // customer name
        String customerName;
        if (customer == null)
            customerName = "occupant";
        else
            customerName = customer.getName();

        // ...
        int weeksDelinquent;
        if (customer == null)
            weeksDelinquent = 0;
        else
            weeksDelinquent = customer.getHistory()
                    .getWeeksDelinquentInLastYear();
    }
}
```

```
class Site{
    Customer _customer;

    Customer getCustomer() {
        return _customer;
    }
}
```

Introduce null object - Example: after refactoring

```java
class Customer {

    public boolean isNull() {
        return false;
    }

    //factory for the null customer
    static Customer newNull() {
        return new NullCustomer();
    }

}

class Site{
    Customer _customer;

    Customer getCustomer() {
        return (_customer == null) ?
            Customer.newNull():
            _customer;
    }
}
```

```java
class NullCustomer extends Customer {

    @Override
    public boolean isNull() {
        return true;
    }

    @Override
    public String getName(){
        return "occupant";
    }

    @Override
    public BillingPlan getPlan(){
        return BillingPlan.basic();
    }

    @Override
    public PaymentHistory getHistory(){
        return PaymentHistory.newNull();
    }
}
```

```java
class OtherClass {

    Site site = new Site()

    void method() {

        // billing plan
        Customer customer = site.getCustomer();
        BillingPlan plan = customer.getPlan();

        // customer name
        String customerName = customer.getName();

        // ...
        int weeksDelinquent = customer.getHistory()
                    .getWeeksDelinquentInLastYear();
    }
}
```

Replace error code with exception

- A method returns a special value to indicate an error
- Throw an exception instead

Replace error code with exception - Example: before refactoring

```
class Account {
    private int _balance;

    int withdraw(int amount) {
        if (amount > _balance)
            return -1;
        else {
            _balance -= amount;
            return 0;
        }
    }

}

class Bank {

    void doTransaction(Account account, int amount) {
        if (account.withdraw(amount) == -1)
            handleOverdrawn();
        else
            doTheUsualThing();
    }
}
```

Replace error code with exception - Example: after refactoring

```
class BalanceException extends Exception {
}

class Account {
    private int _balance;

    void withdraw(int amount) throws BalanceException {
        if (amount > _balance)
            throw new BalanceException();
        _balance -= amount;
    }

}

class Bank {

    void doTransaction(Account account, int amount) {
        try {
            account.withdraw(amount);
            doTheUsualThing();
        } catch (BalanceException e) {
            handleOverdrawn();
        }
    }
}
```

CE/CS/SE 3354
Software Engineering

Software Testing
JUnit

Why Testing?

- Testing vs. code review:
  - More reliable than code review
- Testing vs. static checking:
  - Less false positive and applicable to more problems
- Testing vs. formal verification:
  - More scalable and applicable to more programs
- You get what you pay (linear rewards)
  - While the others are not!

Testing: Concepts

- Test case
- Test fixture
- Test suite
- Test script
- Test driver
- Test result
- Test coverage

Unit Testing

```
public class IMath {

    /**
     * Returns an integer to the square root of x (discarding the fractional parts)
     */
    public int isqrt(int x) {
        int guess = 1;
        while (guess * guess < x) {
            guess++;
        }
        return guess;
    }
}
```

Testing with JUnit

```
import org.junit.Test;
import static org.junit.Assert.*;

/** A JUnit test class to test the class IMath. */
public class IMathTestJUnit1 {

    /** A JUnit test method to test isqrt. */
    @Test
    public void testIsqrt() {
        IMath tester = new IMath();
        assertTrue(0 == tester.isqrt(0));
        assertTrue(1 == tester.isqrt(1));
        assertTrue(1 == tester.isqrt(2));
        assertTrue(1 == tester.isqrt(3));
        assertTrue(10 == tester.isqrt(100));
    }

    /** Other JUnit test methods*/
}
```

Test driver

Test case

Testing with JUnit

```
public class IMathTestJUnit4 {
    private IMath tester;
    @Before /** Setup method executed before each test */
    public void setup(){
        tester=new IMath();
    }

    @Test /** JUnit test methods to test isqrt. */
    public void testIsqrt1() {
        assertEquals("square root for 0 ", 0, tester.isqrt(0));
    }
    @Test
    public void testIsqrt2() {
        assertEquals("square root for 1 ", 1, tester.isqrt(1));
    }
    @Test
    public void testIsqrt3() {
        assertEquals("square root for 2 ", 1, tester.isqrt(2));
    }
    ...
}
```

Test fixture

JUnit: Annotations

Annotation	Description
@Test	Identify test methods
@Test (timeout=100)	Fail if the test takes more than 100ms
@Before	Execute before each test method
@After	Execute after each test method
@BeforeClass	Execute before each test class
@AfterClass	Execute after each test class
@Ignore	Ignore the test method

JUnit: Assertions

Assertion	Description
fail([msg])	Let the test method fail, optional msg
assertTrue([msg], bool)	Check that the boolean condition is true
assertFalse([msg], bool)	Check that the boolean condition is false
assertEquals([msg], expected, actual)	Check that the two values are equal
assertNull([msg], obj)	Check that the object is null
assertNotNull([msg], obj)	Check that the object is not null
assertSame([msg], expected, actual)	Check that both variables refer to the same object
assertNotSame([msg], expected, actual)	Check that variables refer to different objects

CE/CS/SE 3354 Software Engineering

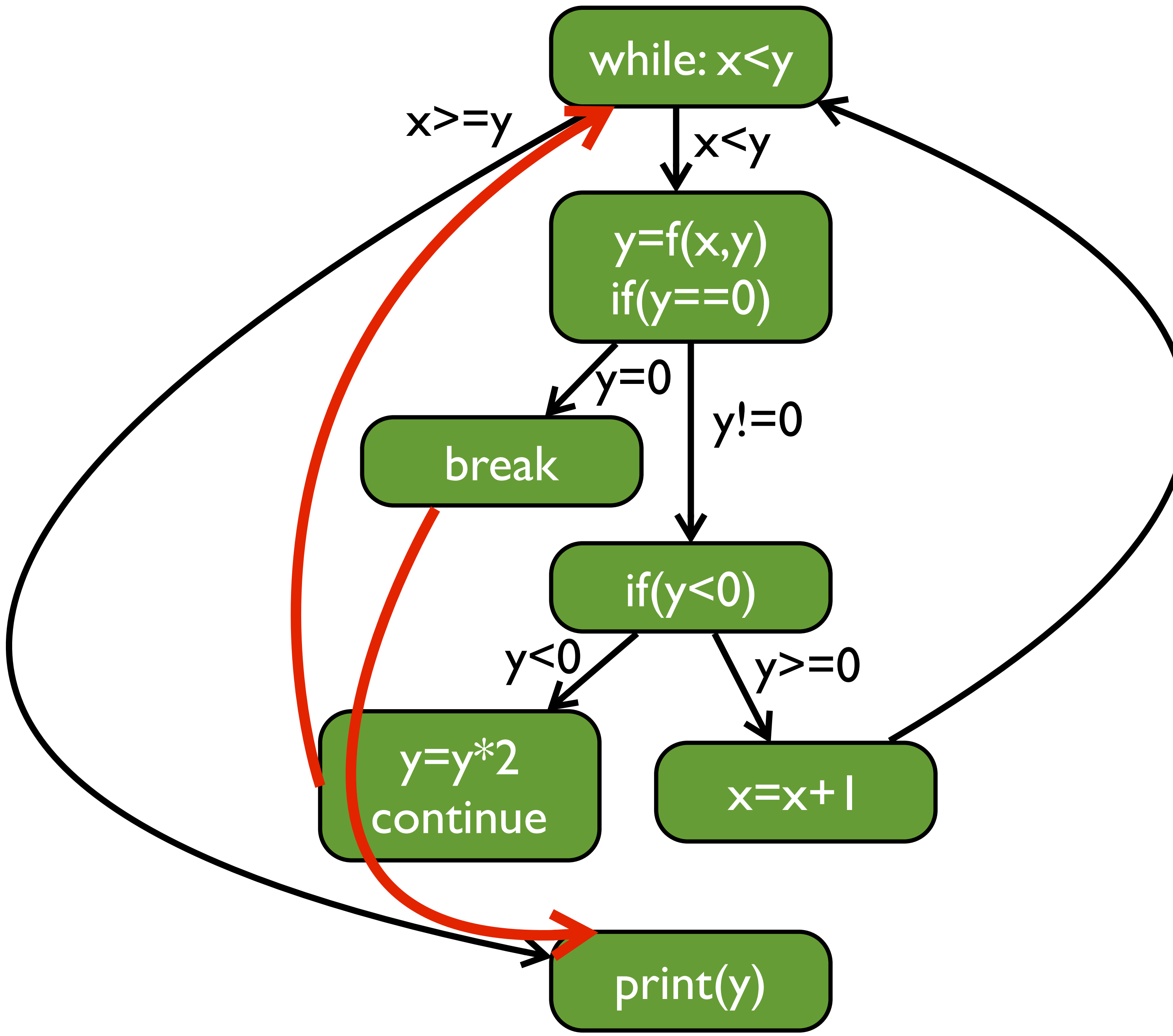
Code Coverage

Control Flow Graphs

- A CFG models all executions of a program by describing control structures
  - Basic Block : A sequence of statements with only one entry point and only one exit point (no branches)
  - Nodes : Statements or sequences of statements (basic blocks)
  - Edges : Transfers of control

CFG

```
while (x < y)
{
   y = f (x, y);
   if (y == 0) {
      break;
   } else if (y<0) {
      y = y*2;
      continue;
   }
   x = x + 1;
}
print (y);
```



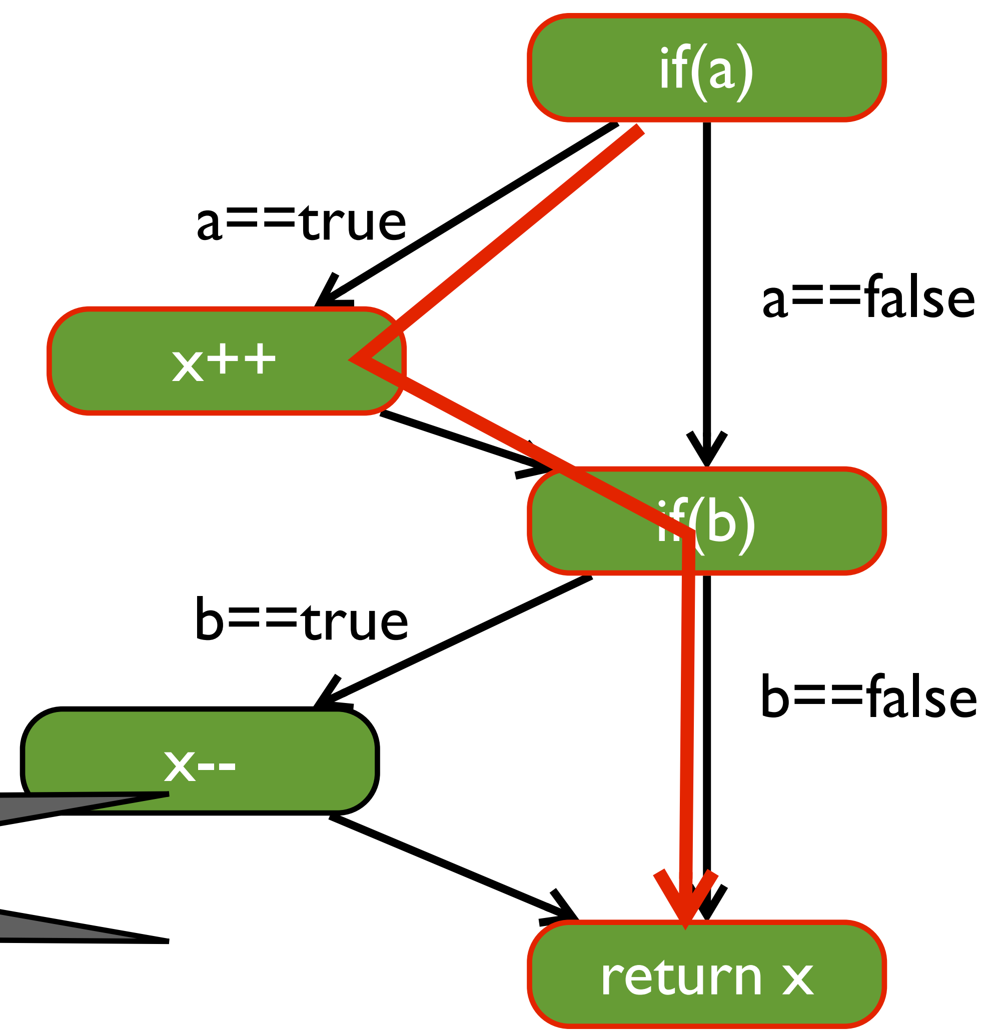
CFG-based Coverage: Statement Coverage

- The percentage of statements covered by the test

tester.testMe(0, true, false)

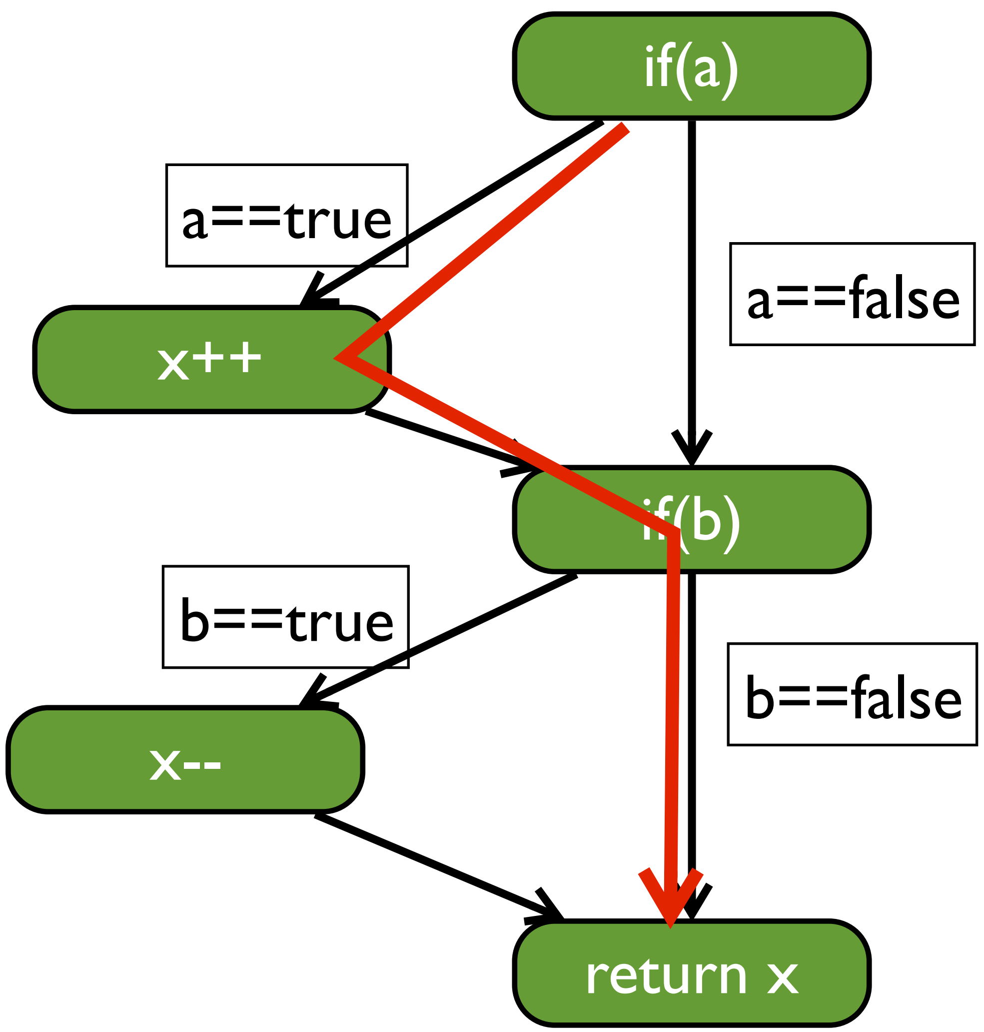
x=0 a=true b=false

SCov=4/5=80%



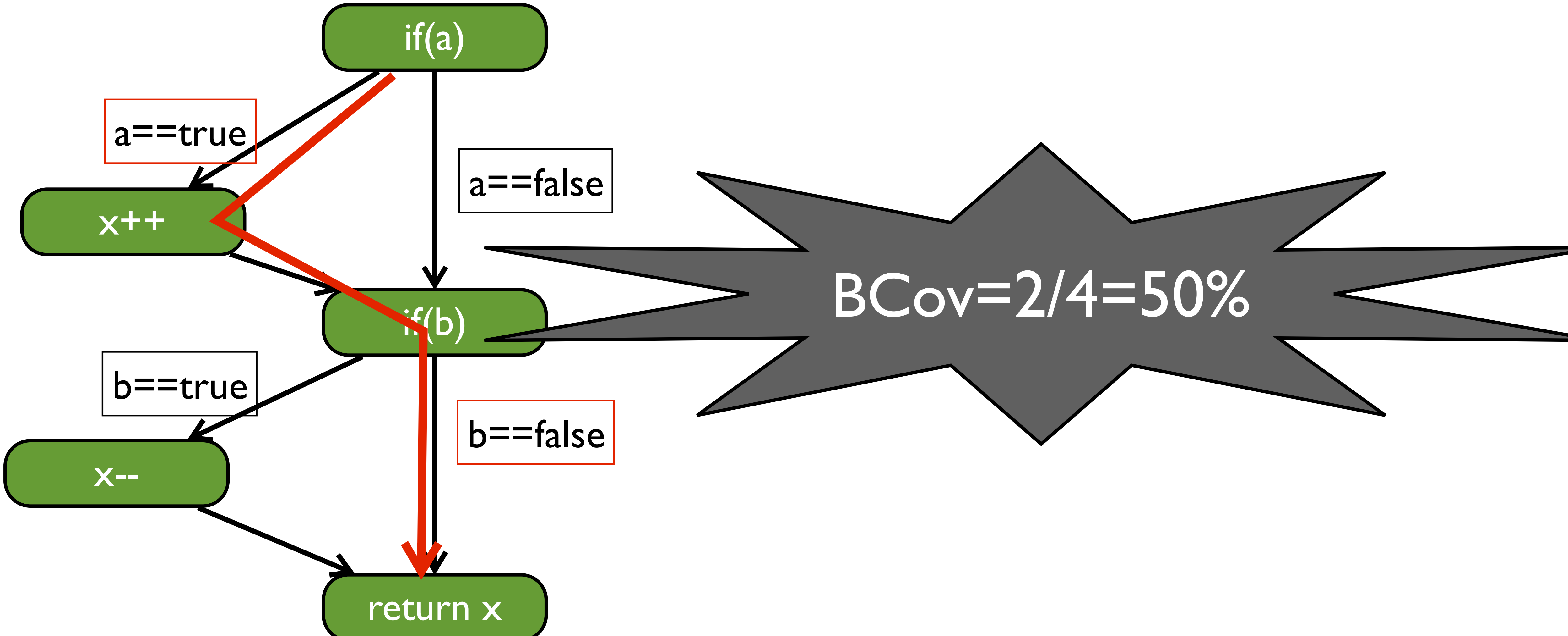
CFG-based Coverage: Branch Coverage

- The percentage of branches covered by the test
  - Consider both false and true branch for each conditional statement



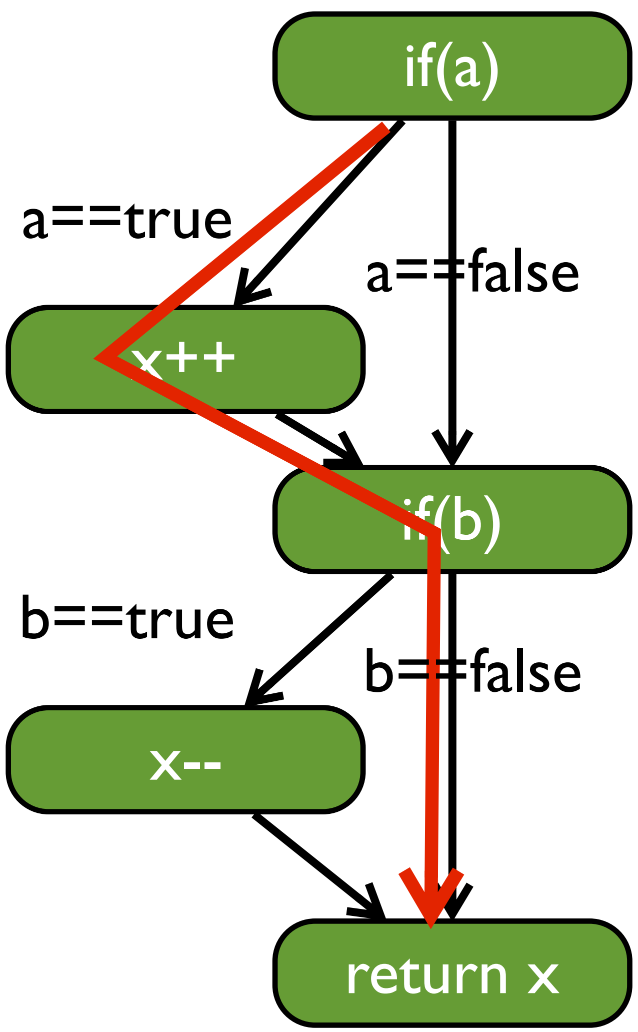
CFG-based Coverage: Branch Coverage

- The percentage of branches covered by the test
  - Consider both false and true branch for each conditional statement



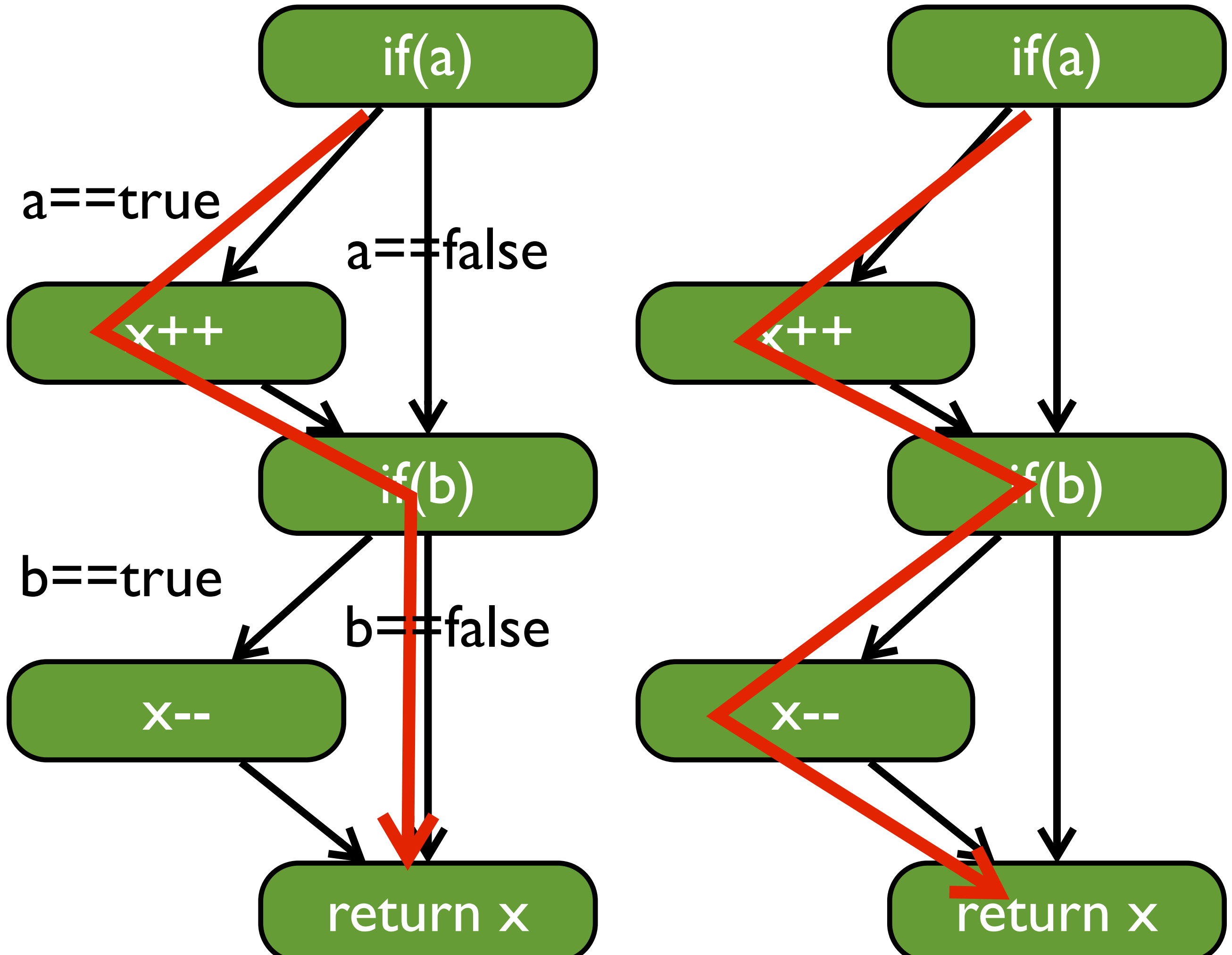
CFG-based Coverage: Path Coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



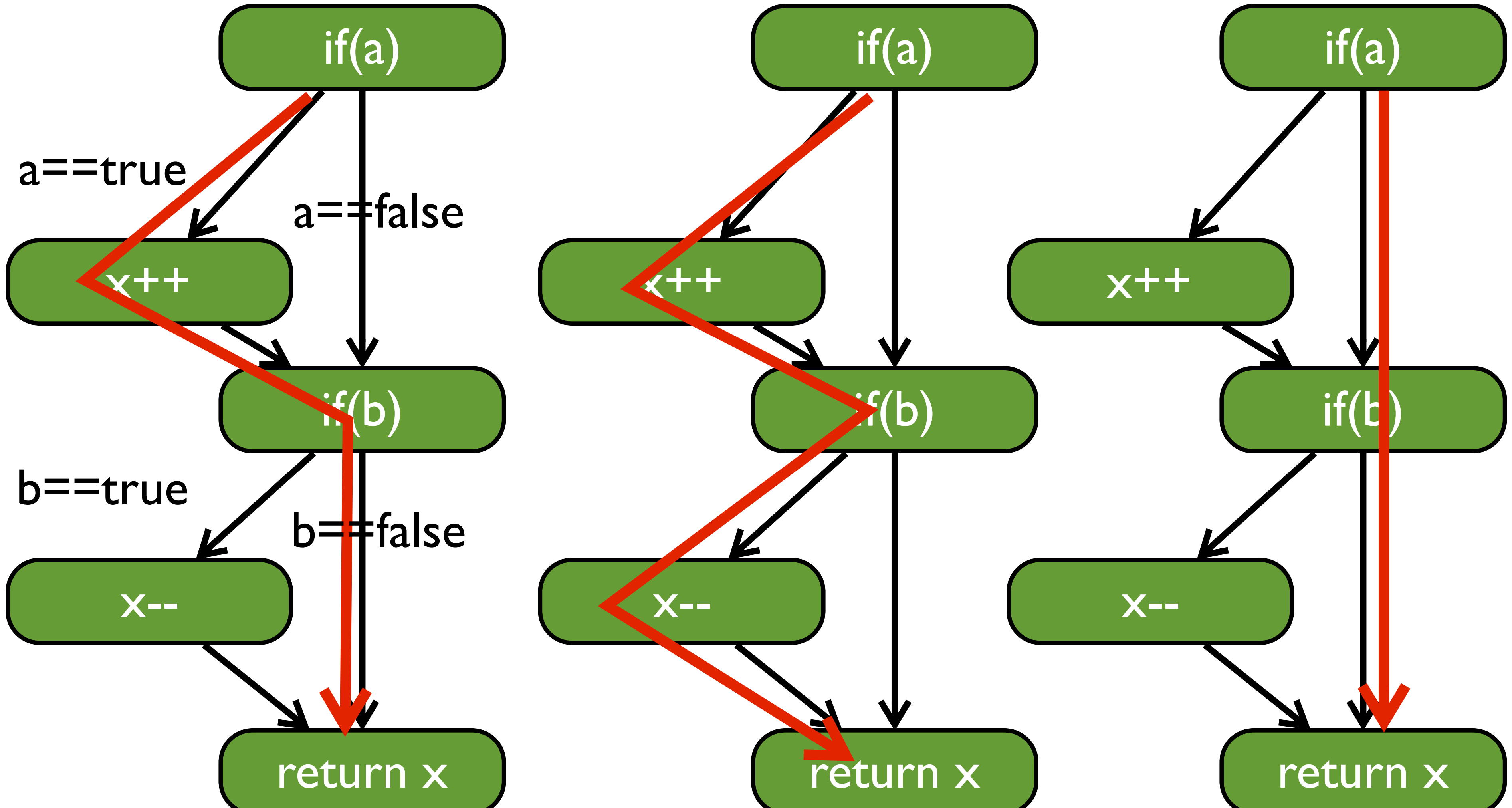
CFG-based Coverage: Path Coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



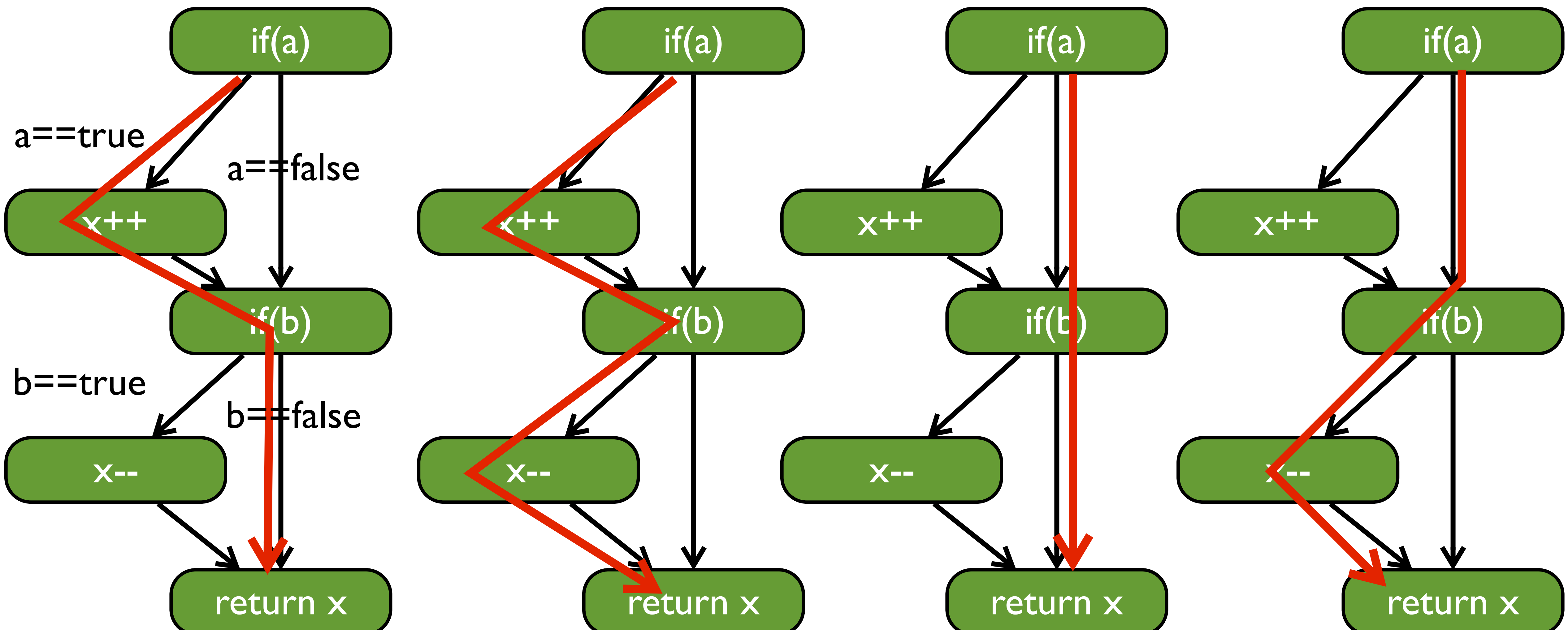
CFG-based Coverage: Path Coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



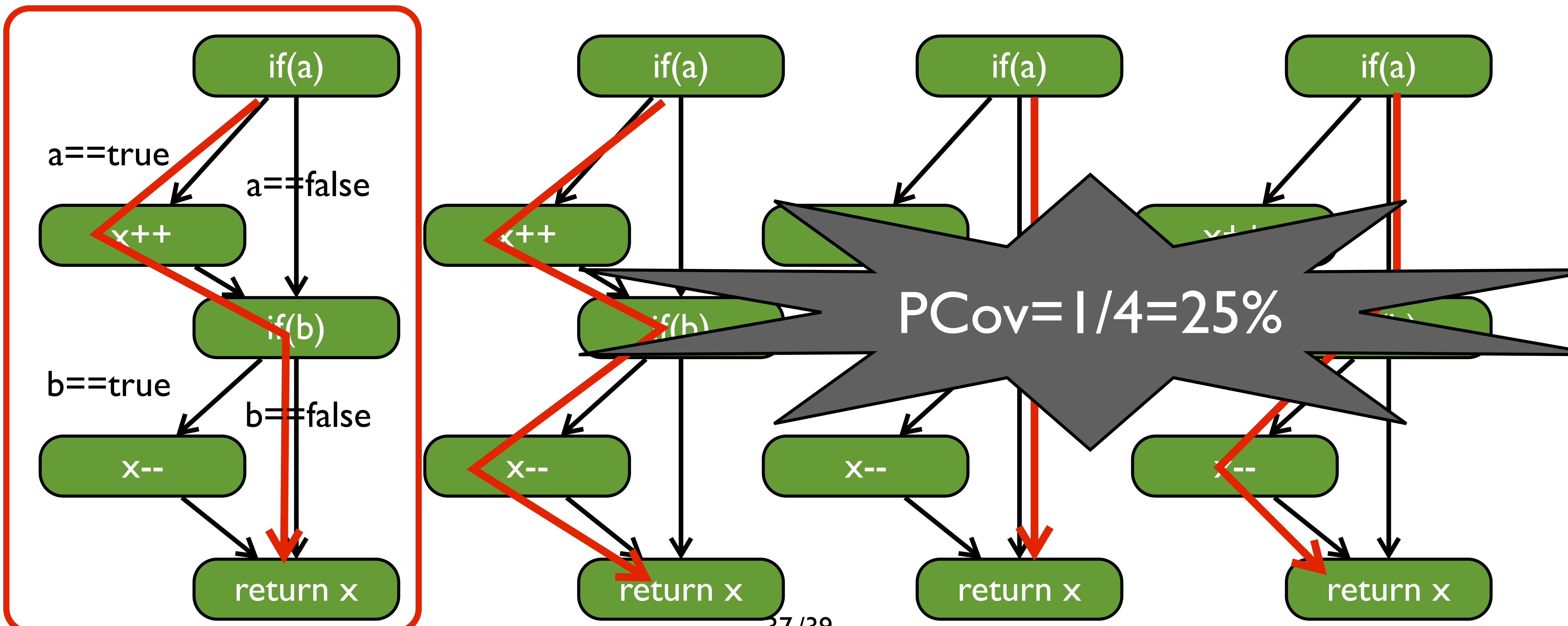
CFG-based Coverage: Path Coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



CFG-based Coverage: Path Coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



CFG-based Coverage: Comparison Summary

Path coverage
strictly subsumes branch coverage
strictly subsumes statement coverage

Hard to achieve:
p-coverage > b-coverage > s-coverage

CFG-based Coverage: Limitation

- 100% coverage of some aspect is never a guarantee of bug-free software

Test: assertEquals(1, sum(1,0))

```
public int sum(int x, int y){
        return x-y; //should be x+y
   }
```

Statement coverage: 100%

Branch coverage: 100%

Path coverage: 100%

Failed to detect the bug...

CE/CS/SE 3354
Software Engineering

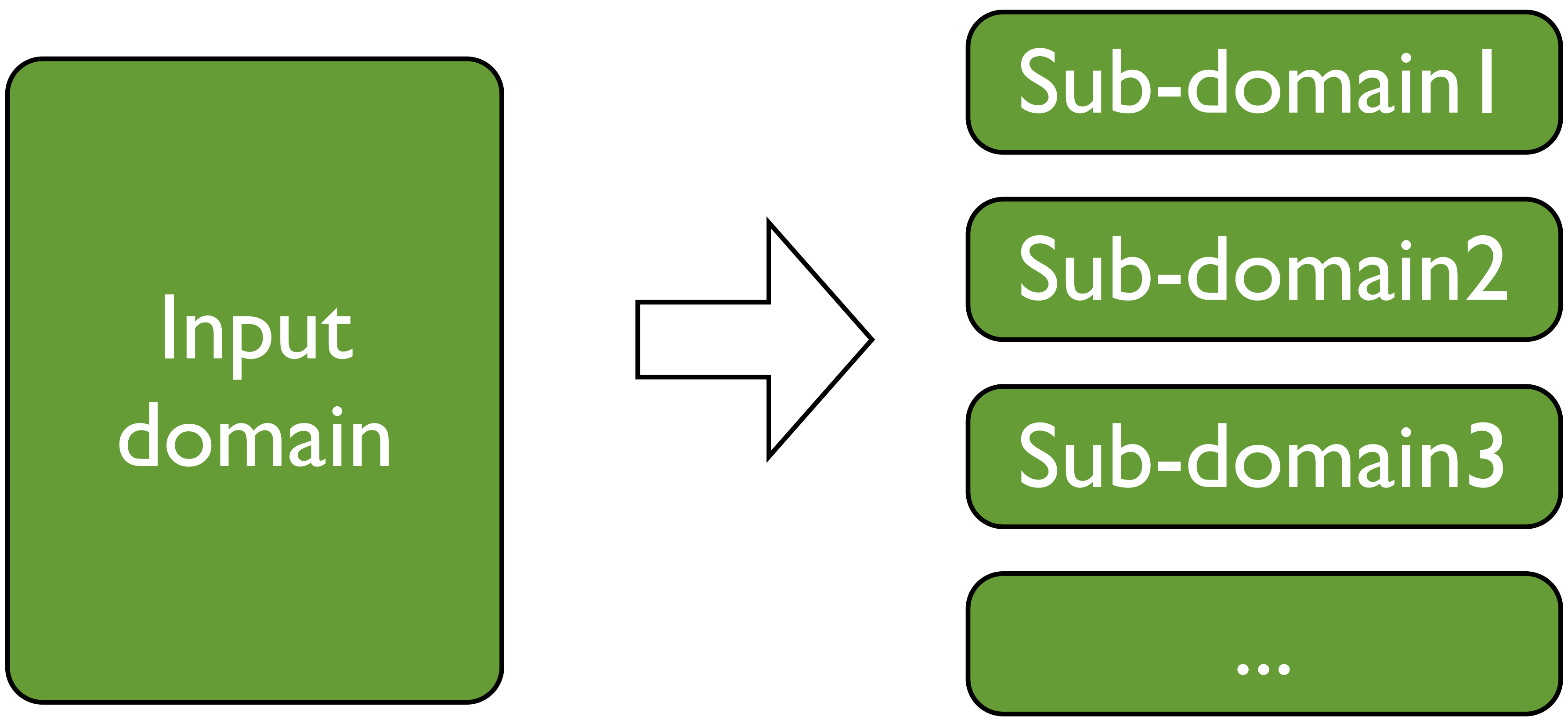
Automated Test Generation

Testing Methodologies

- Black-box (Functional) vs. White-box (Structural) testing
- Black-box testing: Generating test cases based on the functionality of the software
  - Internal structure of the program is hidden from the testing process
- White-box testing: Generating test cases based on the source-code structure of the program
  - Internal structure of the program is taken into account

Domain Testing

- Partition the input domain to equivalence classes
- For some requirements specifications it is possible to define equivalence classes in the input domain
- Choose one test case per equivalence class to test



Domain Testing: Example

- A factorial function specification:

If the input value n is less than 0 then error message must be printed. If $0<=n < 20$, then the exact value n! must be printed. If $20<=n<=200$, then an approximate value of n! must be printed in floating point format using some approximate numerical method. Finally, if $n > 200$, the input can be rejected by printing an error message.

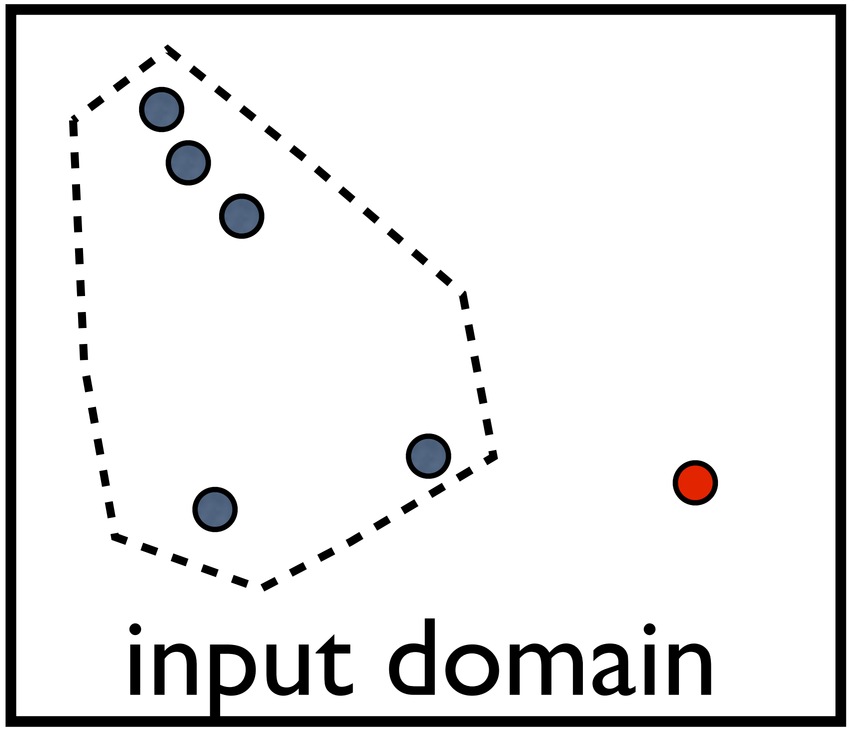
- Possible equivalence classes: D1 = {n<0}, D2 = {0<=n<20}, D3 = {20<=n<=200}, D4 = {n > 200}
- Choose one test case per equivalence class to test

Testing Boundary Conditions

- For the factorial example, ranges for variable n are:
  - [-∞, 0], [0,20], [20,200], [200, ∞]
- A possible test set:
  - {n = -5, n=0, n=11, n=20, n= 25, n=200, n= 3000}
- If we know the maximum and minimum values that n can take, we can also add those n=MIN, n=MAX to the test set

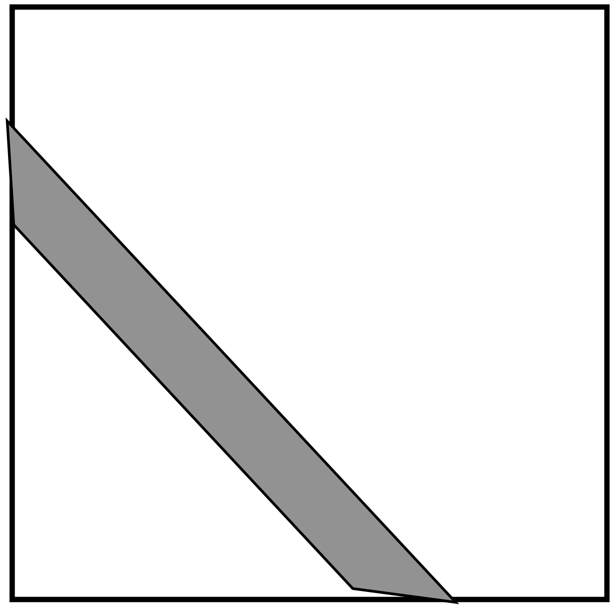
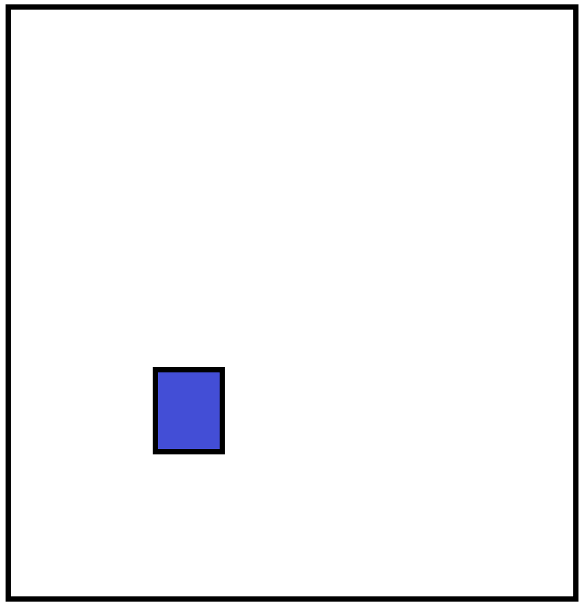
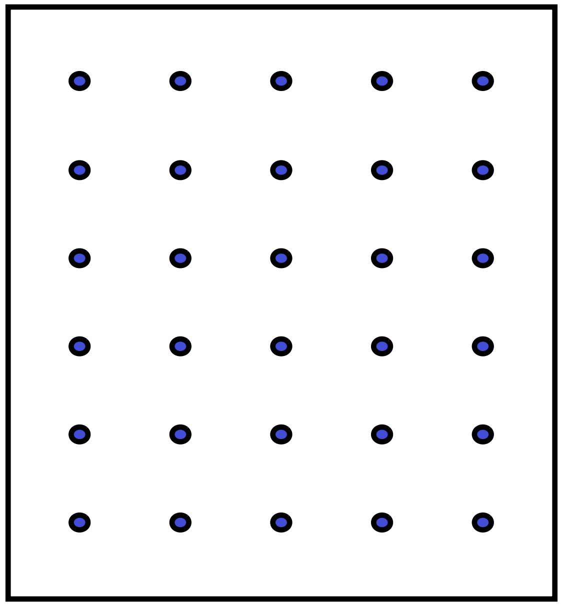
Random Testing

- Random Testing
  - selects tests from the entire input domain randomly and independently



- Advantages:
  - Intuitively simple
  - Allows statistical estimation of the software's reliability
- Disadvantages:
  - No guide towards failure-causing inputs

Types of Failure Patterns

Strip Pattern	Block Pattern	Point Pattern
		

CE/CS/SE 3354
Software Engineering

Dynamic Bug Detection

A representative technique: Tarantula

Statements	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`							0.5
`m = z;`							0.5
`if (y < z) {`							0.5
`if (x < y)`							0.63
`m = y;`							0
`else if (x < z)`							0.71
`m =` 							0.83
`} else {`							0
							0
							0
							0
							0
`return m;`							0.5
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\frac{1/1}{1/1+5/5}$$

$$\frac{1/1}{1/1+1/5}$$

Tarantula :

$$Suspicious(s) = \frac{\frac{fail(s)}{totalfail}}{\frac{fail(s)}{totalfail} + \frac{pass(s)}{totalpass}}$$

More Formulae

- Tarantula

$$Suspicious(s) = \frac{fail(s)/totalfail}{fail(s)/totalfail + pass(s)/totalpass}$$

- SBI

$$Suspicious(s) = \frac{fail(s)}{fail(s) + pass(s)}$$

- Jaccard

$$Suspicious(s) = \frac{fail(s)}{allfailed + pass(s)}$$

- Ochiai

$$Suspicious(s) = \frac{fail(s)}{\sqrt{allfailed * (pass(s) + fail(s))}}$$

Delta Debugging

- The problem definition
  - A program exhibit an error for an input
  - The input is a set of elements
  - E.g., a sequence of API calls, a text file, a serialized object, …
  - Problem:

    Find a smaller subset of the input that still cause the failure

Delta Debugging

- Algorithm
  - Split I to I1 and I2
  - Case I: I1 = ✗ and I2 = ✓

    Try I1
  - Case II: I1 = ✓ and I2 = ✗

    Try I2
  - Case III: I1 = ✗ and I2 = ✗

    Try both I1 and I2
  - Case IV: I1 = ✓ and I2 = ✓

    Handle interference for I1 and I2

UT DALLAS

CE/CS/SE 3354
Software Engineering

Software Security

Kinds of undesired behavior

- Stealing information: **~~confidentiality~~**
  - Corporate secrets (product plans, source code, …)
  - Personal information (credit card numbers, SSNs, …)
- Modifying information or functionality: **~~integrity~~**
  - Installing unwanted software (spyware, …)
  - Destroying records (accounts, logs, plans, …)
- Denying access: **~~availability~~**
  - Unable to purchase products
  - Unable to access banking information

Secure Software Development

- Consider security *throughout software lifecycle*
  - **Requirements**
  - **Design**
  - **Implementation**
  - **Testing**

- Corresponding activities
  - **Define** *security requirements, abuse cases,*
  - **Perform** *architectural risk analysis (threat modeling)* and *security-conscious design*
  - **Conduct** *code reviews, risk-based security testing,* and *penetration testing*

Making secure software

- **Flawed approach**: Design and build software, and ***ignore security at first***
  - Add security once the functional requirements are satisfied

- **Better approach**: ***Build security in*** from the start
  - Incorporate security-minded thinking into all phases of the development process

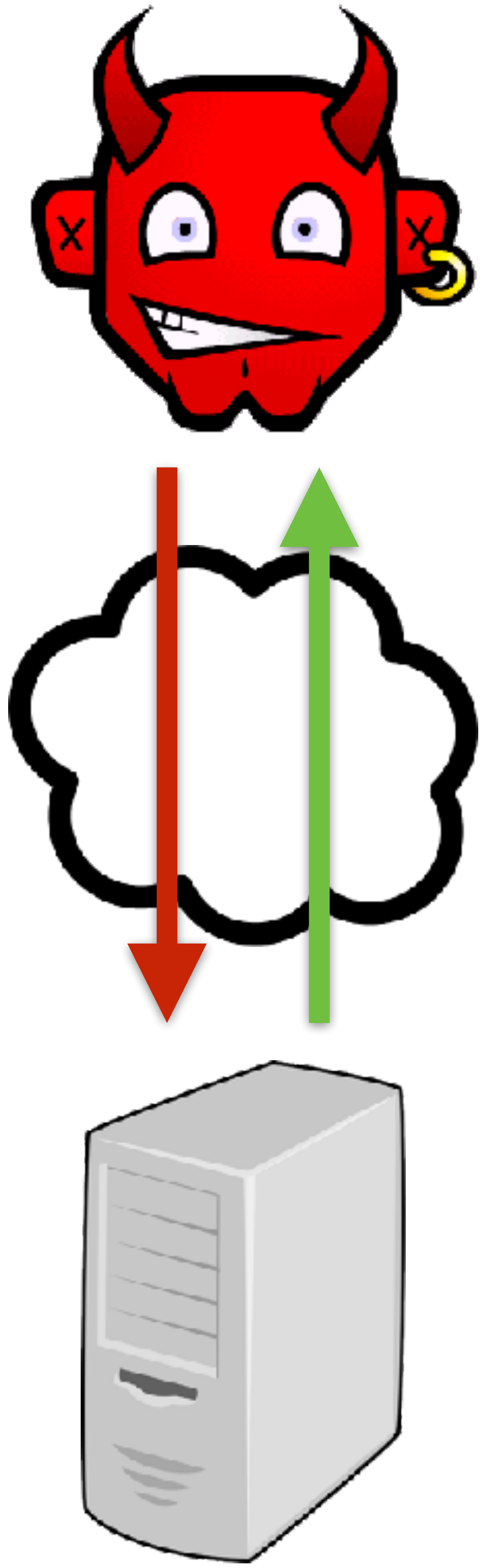
Threat Model

- The **threat model makes explicit the adversary's assumed powers**
  - Consequence: The threat model must match reality, otherwise the risk analysis of the system will be wrong
- The threat model is **critically important**
  - If you are not explicit about what the attacker can do, how can you assess whether your design will repel that attacker?
- This is part of **architectural risk analysis**

Example: **Network User**

- An (anonymous) user that can connect to a service via the network
- Can:
  - **measure** the size and timing of requests and responses
  - run **parallel sessions**
  - provide **malformed inputs**, **malformed messages**
  - **drop or send extra messages**

- **Example attacks**: SQL injection, XSS, buffer overrun, …

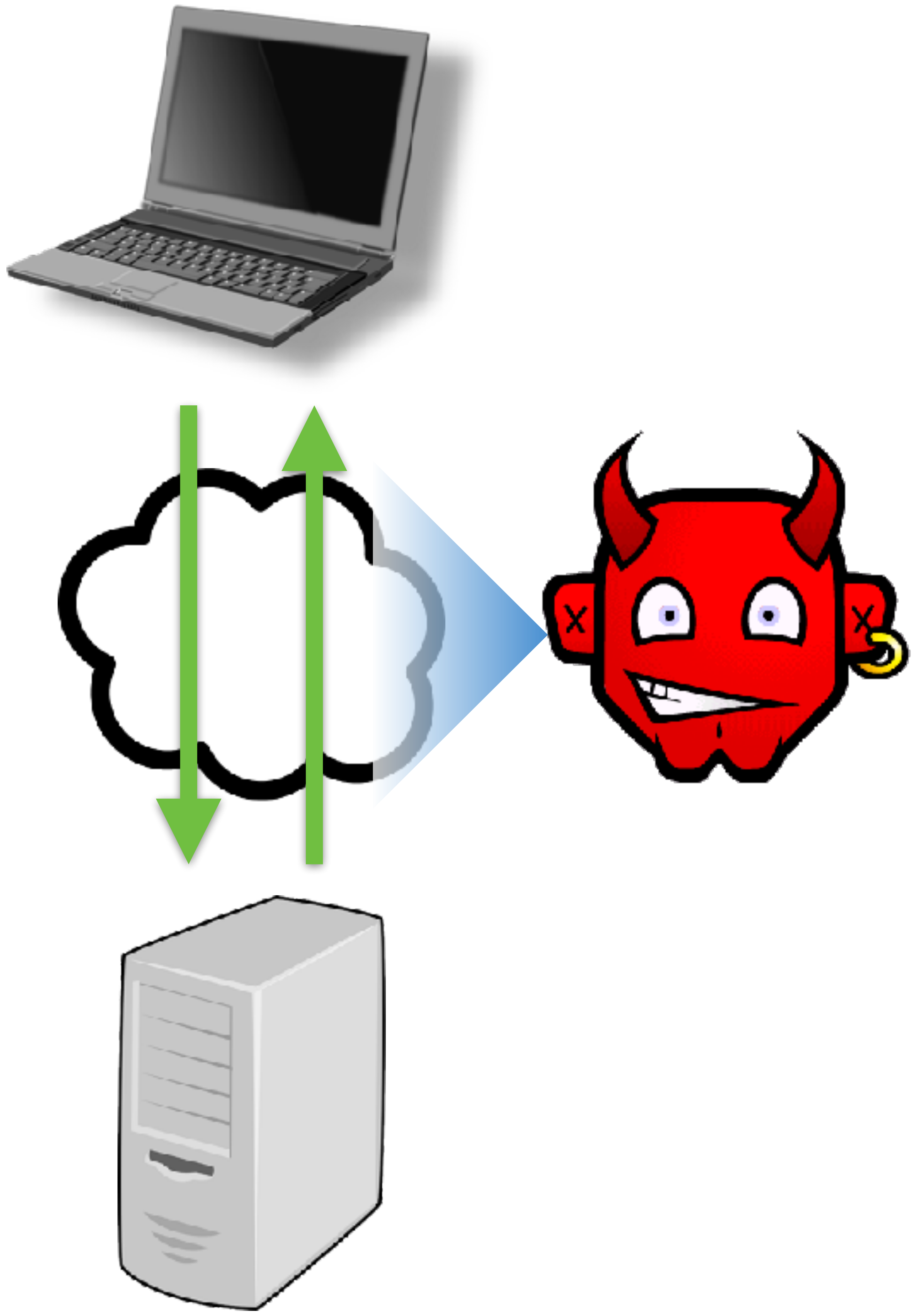


Example: **Snooping User**

- Internet user **on the same network** as other users of some service
  - For example, someone connected to an unencrypted Wi-Fi network at a coffee shop

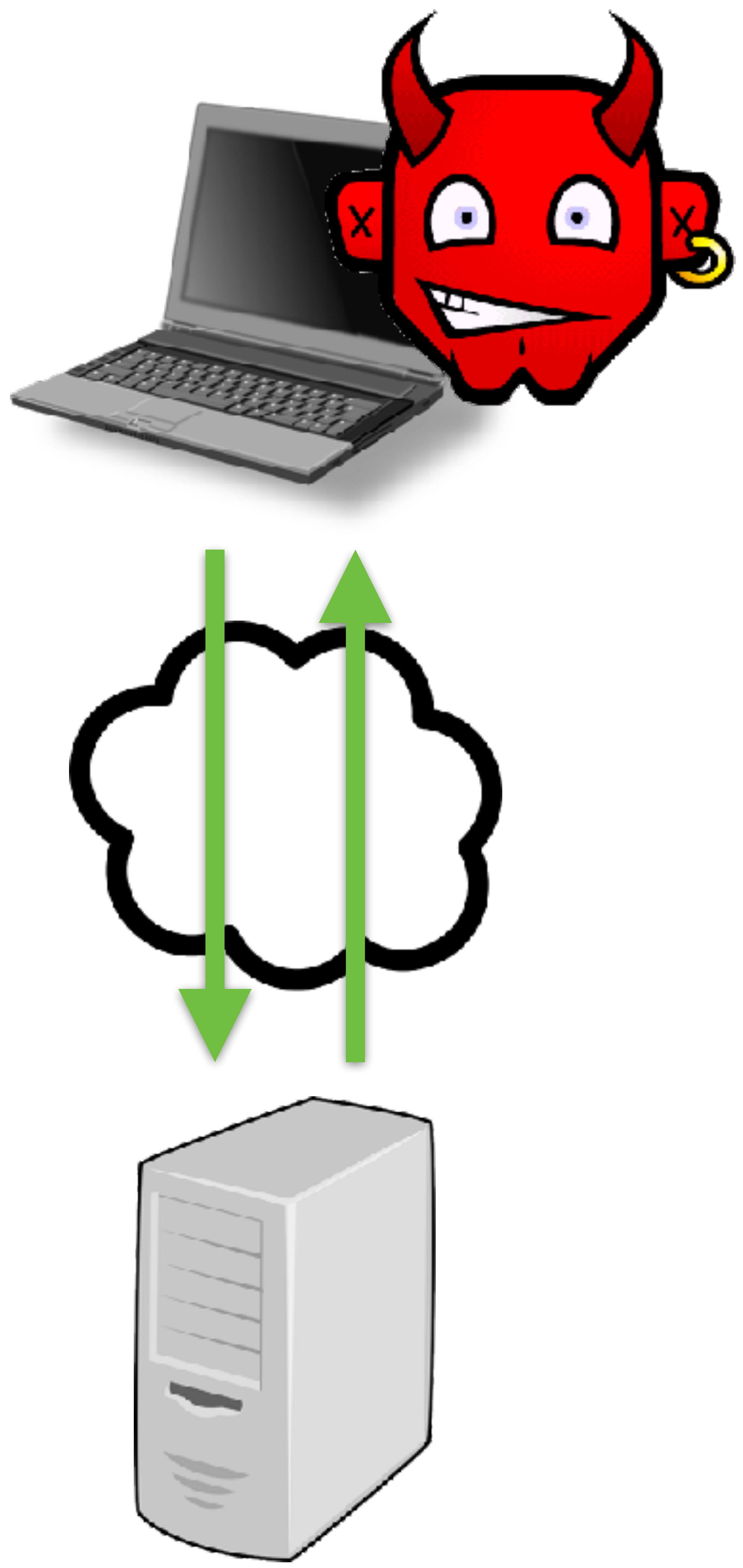
- Thus, can additionally
  - **Read/measure** others' **messages**,
  - **Intercept, duplicate,** and **modify messages**

- **Example attacks**: **Session hijacking** (and other data theft), **privacy-violating side-channel attack**, **denial of service**



Example: **Co-located User**

- Internet user **on the same machine** as other users of some service
  - E.g., **malware** installed on a user's laptop
- Thus, can additionally
  - **Read/write** user's **files** (e.g., cookies) and **memory**
  - **Snoop keypresses** and other events
  - Read/write the user's **display**
- **Example attacks**: **Password theft** (and other credentials/secrets)



Security Requirements

- **Software requirements** typically about **what** the **software should do**

- We also want to have **security requirements**
  - **Security-related goals** (or **policies**)
    - **Example**: One user's bank account balance should not be learned by, or modified by, another user, unless authorized
  - **Required mechanisms for enforcing them**
    - **Example**:
      1. Users identify themselves using passwords,
      2. Passwords must be "strong", and
      3. The password database is only accessible to login program.

Typical *Kinds* of Requirements

- **Policies**
  - **Confidentiality** (and Privacy)
  - **Integrity**
  - **Availability**

- Supporting **mechanisms**
  - **Authentication**
  - **Authorization**
  - **Auditability**

Abuse Cases

- Abuse cases illustrate security requirements
- Where use cases describe what a system *should* do, **abuse cases describe what it should *not* do**
- Example **use case**: The system shall allow bank managers to modify an account's interest rate
- Example **abuse case**: A user is able to spoof being a manager and thereby change the interest rate on an account

Secure Software Design

Design software architecture according to good **principles** and **rules**

Risk-based analysis of software architecture's design

Categories of Principles

- A **principle** is a high-level design goal with many possible manifestations
- **Prevention**
  - **Goal**: Eliminate software defects entirely
  - **Example**: Heartbleed bug would have been prevented by using a type-safe language, like Java
- **Mitigation**
  - **Goal**: Reduce the harm from exploitation of unknown defects
  - **Example**: Run each browser tab in a separate process, so exploitation of one tab does not yield access to data in another
- **Detection** (and **Recovery**)
  - **Goal**: Identify and understand an attack (and undo damage)
  - **Example**: Monitoring (e.g., expected invariants), snapshotting

The Principles

- **Favor simplicity**
  - Use fail-safe defaults
  - Do not expect expert users
- **Trust with reluctance**
  - Employ a small trusted computing base
  - Grant the least privilege possible
    - Promote privacy
    - Compartmentalize
- **Defend in Depth**
  - Use community resources - no security by obscurity
- **Monitor and trace**

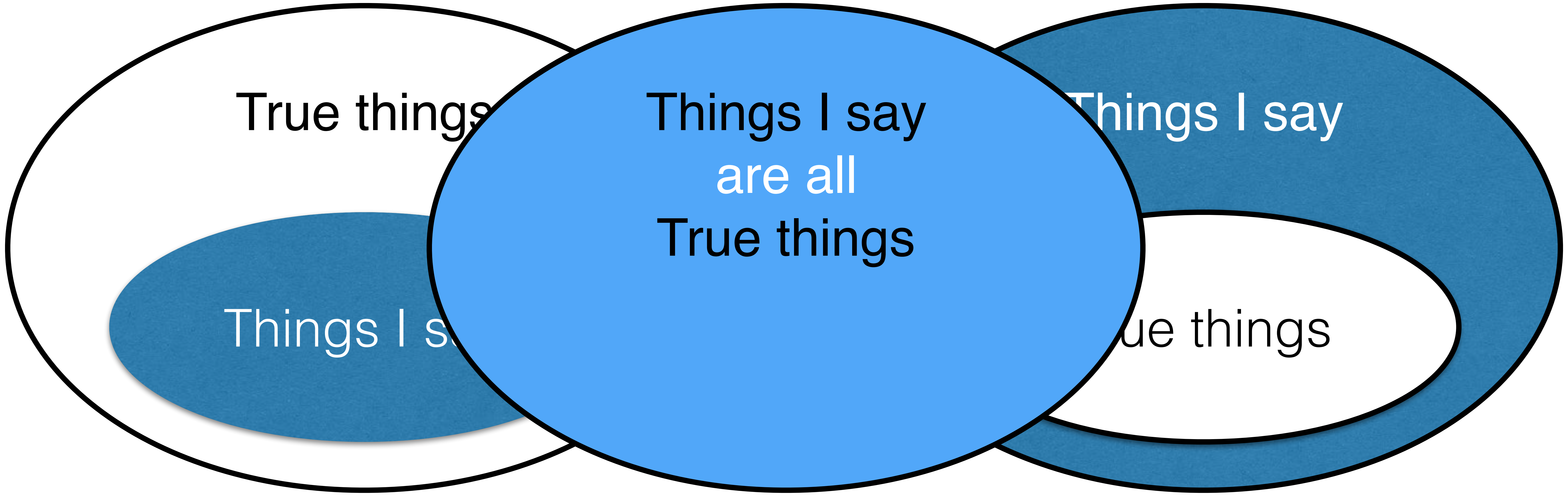
CE/CS/SE 3354
Software Engineering

Flow analysis for secure development
Penetration testing

Soundness Completeness

If analysis says that X is true, then X is true.

If X is true, then analysis says X is true.



Trivially Sound: Say nothing

Trivially Complete: Say everything

Sound and **Complete:**

Say exactly the set of true things

Flow Sensitivity

- Our analysis is **flow *in*sensitive**
  - Each variable has **one qualifier** which abstracts the taintedness of all values it ever contains

- A **flow sensitive analysis** would account for variables whose contents change
  - Allow each assigned use of a variable to have a different qualifier
    - E.g., α_1 is x's qualifier at line 1, but α_2 is the qualifier at line 2, where α_1 and α_2 can differ
  - Could implement this by transforming the program to assign to a variable at most once
    - Called **static single assignment (SSA)** form

Flow Sensitivity

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);
```

```
α char *name = fgets(…, network_fd);
β char *x1, γ *x2;
x1 = name;
x2 = "%s";
printf(x2);
```

$tainted \leq \alpha$

$\alpha \leq \beta$

$untainted \leq \gamma$

$\gamma \leq untainted$

No Alarm

Good solution exists:

$\gamma = untainted$

$\alpha = \beta = tainted$

Why *not* flow/path sensitivity?

- Flow sensitivity **adds precision**, and path sensitivity adds even more, which is *good*
- But both of these **make solving more difficult**
  - Flow sensitivity also *increases the number of nodes* in the constraint graph
  - Path sensitivity *requires more general solving procedures* to handle path conditions
- In short: **precision** (often) **trades off scalability**
  - Ultimately, limits the size of programs we can analyze

Context (In)sensitivity

- This is a problem of **context insensitivity**
  - All call sites are "conflated" in the graph
- **Context sensitivity** solves this problem by
  - **distinguishing call sites** in some way
    - We can give them a label ***i***, e.g., the line number in the program
  - **matching** up **calls** with the corresponding **returns**
    - Label call and return edges
    - Allow flows if the labels and **polarities** match
    - Use index **-i** for **argument passing**, i.e., $q_1 \leq_{-i} q_2$
    - Use index **+i** for **returned values**, i.e., $q_1 \leq_{+i} q_2$

Two Calls to Same Function

```
α char *a = fgets(…);
β char *b = id1(a);
ω char *c = id2("hi");
printf(c);
```

```
δ char *id(γ char *x) {
    return x;
}
```

tainted $\leq$ α

α $\leq_{-1}$ γ

γ $\leq$ δ

δ $\leq_{+1}$ β

untainted $\leq_{-2}$ γ

δ $\leq_{+2}$ ω

ω $\leq$ untainted

Indexes don't match up

Infeasible flow not allowed

No Alarm

Discussion

- **Context sensitivity** is a **tradeoff** again
  - *Precision* vs. *scalability*
    - $O(n)$ insensitive algorithm becomes $O(n^3)$ sensitive algorithm
  - But: sometimes *higher precision improves performance*
    - Eliminates infeasible paths from consideration (makes *n* smaller)

- Compromises possible
  - Only ***some* call sites treated sensitively**
    - Rest conflated
  - **Conflate *groups* of call sites**
    - Give them the same index
  - **Sensitivity only *up to a certain call depth***
    - Don’t do exact matching of edges beyond that depth

Implicit flows

```
void copy(tainted char *src,
          untainted char *dst,
          int len) {
  untainted int i, j;
  for (i = 0; i<len; i++) {
    for (j = 0; j<sizeof(char)*256; j++) {
      if (src[i] == (char)j)
        dst[i] = (char)j; //legal?
    }
  }
}
```

untainted char **untainted** char

Missed flow

Pen testing

- Pen testers employ **ingenuity** and **automated tools**
  - To rapidly explore a system's *attack surface*, looking for weaknesses to exploit
- Typically **carried out by a separate group** within, or outside, an organization, separate from developers
  - *Avoids tunnel vision*: Same reason doctors tend to not treat themselves or their own families
- Given **varied access to system internals**
  - From *no access*, like outside attacker, to *full access*, like a knowledgable insider

What is fuzzing?

- A kind of **random testing**
- **Goal**: make sure certain **bad things don't happen, no matter what**
  - **Crashes, thrown exceptions, non-termination**
  - All of these things can be the foundation of security vulnerabilities
- **Complements functional testing**
  - Test features (and lack of misfeatures) directly
  - Normal tests can be starting points for fuzz tests

Kinds of fuzzing

- **Black box**
  - The tool knows nothing about the program or its input
  - **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky
- **Grammar based**
  - The tool generates input informed by a grammar
  - **More work to use**, to produce the grammar, but **can go deeper** in the state space
- **White box**
  - The tool generates new inputs at least partially informed by the code of the program being fuzzed
  - Often **easy to use**, but **computationally expensive**