Foundations of Software Engineering

Static analysis (1/2)

Claire Le Goues

institute for SOFTWARE RESEARCH

Two fundamental concepts

- **Abstraction.**
  - Elide details of a specific implementation.
  - Capture semantically relevant details; ignore the rest.
- **Programs as data.**
  - Programs are just trees/graphs!
  - …and we know lots of ways to analyze trees/graphs, right?

Learning goals

- Give a one sentence definition of static analysis. Explain what types of bugs static analysis targets.
- Give an example of syntactic or structural static analysis.
- Construct basic control flow graphs for small examples by hand.
- Distinguish between control- and data-flow analyses; define and then step through on code examples simple control and data-flow analyses.

```
goto fail;
```

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
                                 SSLBuffer signedParams,
                                 uint8_t *signature,
                                 UInt16 signatureLen) {
     OSStatus err;
      .…
     if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
             goto fail;
     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
             goto fail;
             goto fail;
     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
             goto fail;
     …
fail:
     SSLFreeBuffer(&signedHashes);
     SSLFreeBuffer(&hashCtx);
     return err;
}
```

Is there a bug in this code?

```
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
     return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

ERROR: function returns with interrupts disabled!

With thanks to Jonathan Aldrich; example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

Could you have found them?

- How often would those bugs trigger?
- Driver bug:
  - What happens if you return from a driver with interrupts disabled?
  - Consider: that's one function
    - …in a 2000 LOC file
    - …in a module with 60,000 LOC
    - …IN THE LINUX KERNEL
- **Moral:** *Some defects are very difficult to find via testing, inspection.*

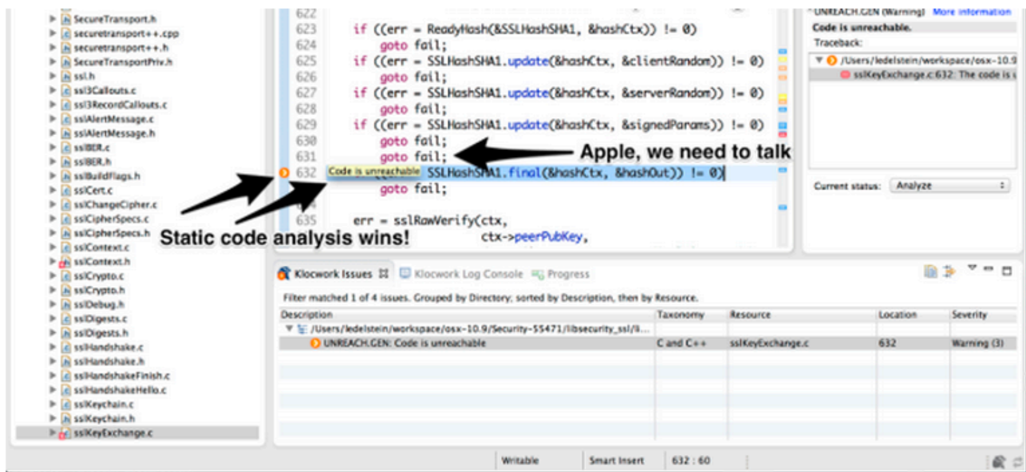
Klocwork: Our source code analyzer caught Apple's 'gotofail' bug

If Apple had used a third-party source code analyzer on its encryption library, it could have avoided the "gotofail" bug.

by Declan McCullagh | February 28, 2014 1:13 PM PST

Follow

57 223 23 5 More + Comments 25



Klocwork's Larry Edelstein sent us this screen snapshot, complete with the arrows, showing how the company's product would have nabbed the "goto fail" bug.

(Credit: Klocwork)

It was a single repeated line of code -- "goto fail" -- that left millions of Apple users vulnerable to Internet attacks until the company finally fixed it Tuesday.

Featured Posts

Google unveils Androi wearables
Internet & Media

Motorol powered
Internet

OK, Glas in my fa
Cutting E

Apple iP product
Apple

iPad wit comeba
Apple

Most Popular

Giant 3D house
6k Facel

Exclusiv Doesch
716 Twe

Google' four can
771 Goo

Connect With CNET

Facebook
Like Us

Google +

http://news.cnet.com/8301-1009_3-57619754-83/klocwork-our-source-code-analyzer-caught-apples-gotofail-bug/

institute for SOFTWARE RESEARCH

Defects of interest…

- Are on uncommon or difficult-to-force execution paths.
  - Which is why it's hard to find them via testing.
- Executing (or interpreting/otherwise analyzing) all paths concretely to find such defects is infeasible.
- **What we really want to do is check the entire possible state space of the program for particular properties.**

Defects Static Analysis can Catch

- **Defects that result from inconsistently following simple, mechanical design rules.**
  - **Security:** Buffer overruns, improperly validated input.
  - **Memory safety:** Null dereference, uninitialized data.
  - **Resource leaks:** Memory, OS resources.
  - **API Protocols:** Device drivers; real time libraries; GUI frameworks.
  - **Exceptions:** Arithmetic/library/user-defined
  - **Encapsulation:** Accessing internal data, calling private functions.
  - **Data races:** Two threads access the same data without synchronization

Key: check compliance to simple, mechanical design rules

What is Static Analysis?

- **Systematic** examination of an **abstraction** of program **state space**.
  – Does not execute code!
- **Abstraction:** produce a representation of a program that is simpler to analyze.
  – Results in fewer states to explore; makes difficult problems tractable.
- Check if a **particular property** holds over the entire state space:
  – Liveness: “something good eventually happens.”
  – Safety: “this bad thing can’t ever happen.”

The Bad News: Rice's Theorem

"Any nontrivial property about the language recognized by a Turing machine is undecidable.“

Henry Gordon Rice, 1953

Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)

	Error exists	No error exists
Error Reported	True positive (correct analysis result)	False positive
No Error Reported	False negative	True negative (correct analysis result)

Sound Analysis:

reports all defects
-> no false negatives
typically overapproximated

Complete Analysis:

every reported defect is an actual defect
-> no false positives
typically underapproximated

How does testing relate? And formal verification?

Syntactic Analysis

Find every occurrence of this pattern:

```
public foo() {
  …
   logger.debug(“We have ” + conn + “connections.”);
}
```

```
public foo() {
  …
  if (logger.inDebug()) {
    logger.debug(“We have ” + conn + “connections.”);
  }
}
```

grep "if \(logger\.inDebug" . -r

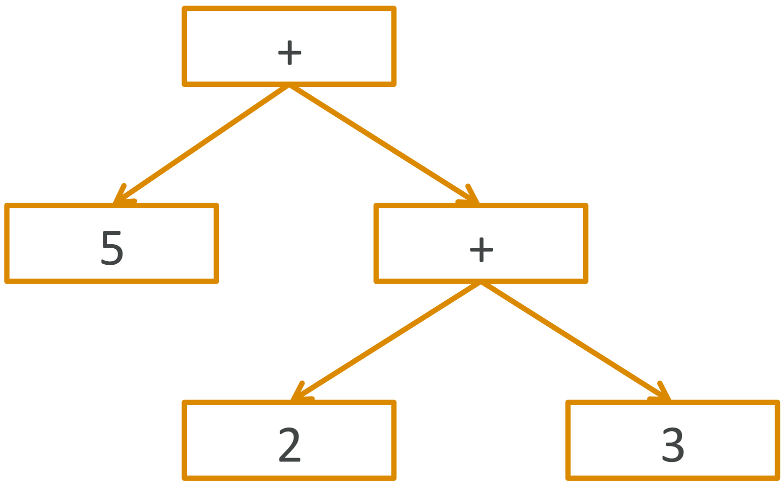
Abstraction?

```java
public void foo() {
    int a = computeSomething();

    if (a == "5")
        doMoreStuff();
}
```

Abstraction: abstract syntax tree

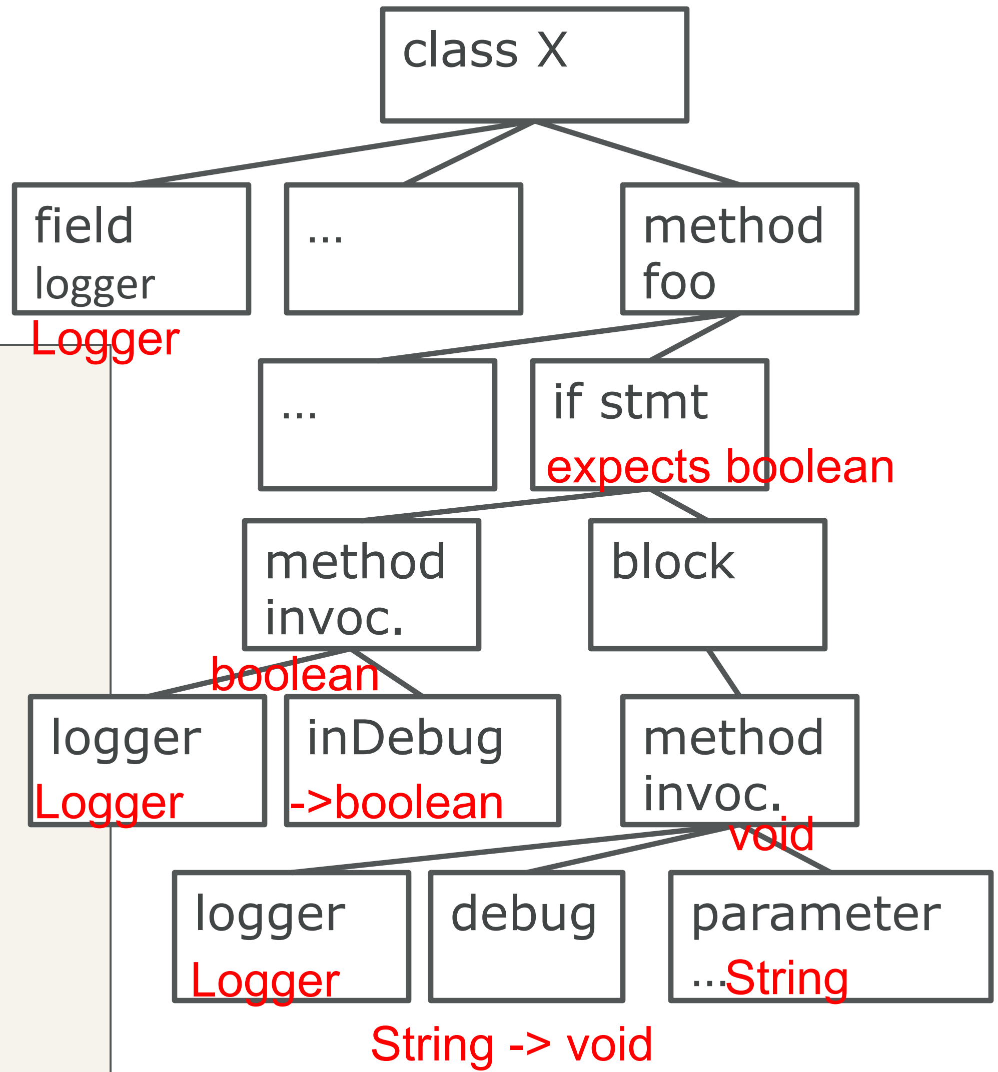
- Tree representation of the syntactic structure of source code.
  - Parsers convert concrete syntax into abstract syntax, and deal with resulting ambiguities.
- Records only the semantically relevant information.
  - Abstract: doesn’t represent every detail (like parentheses); these can be inferred from the structure.
- (How to build one? Take compilers!)

- Example: 5 + (2 + 3)

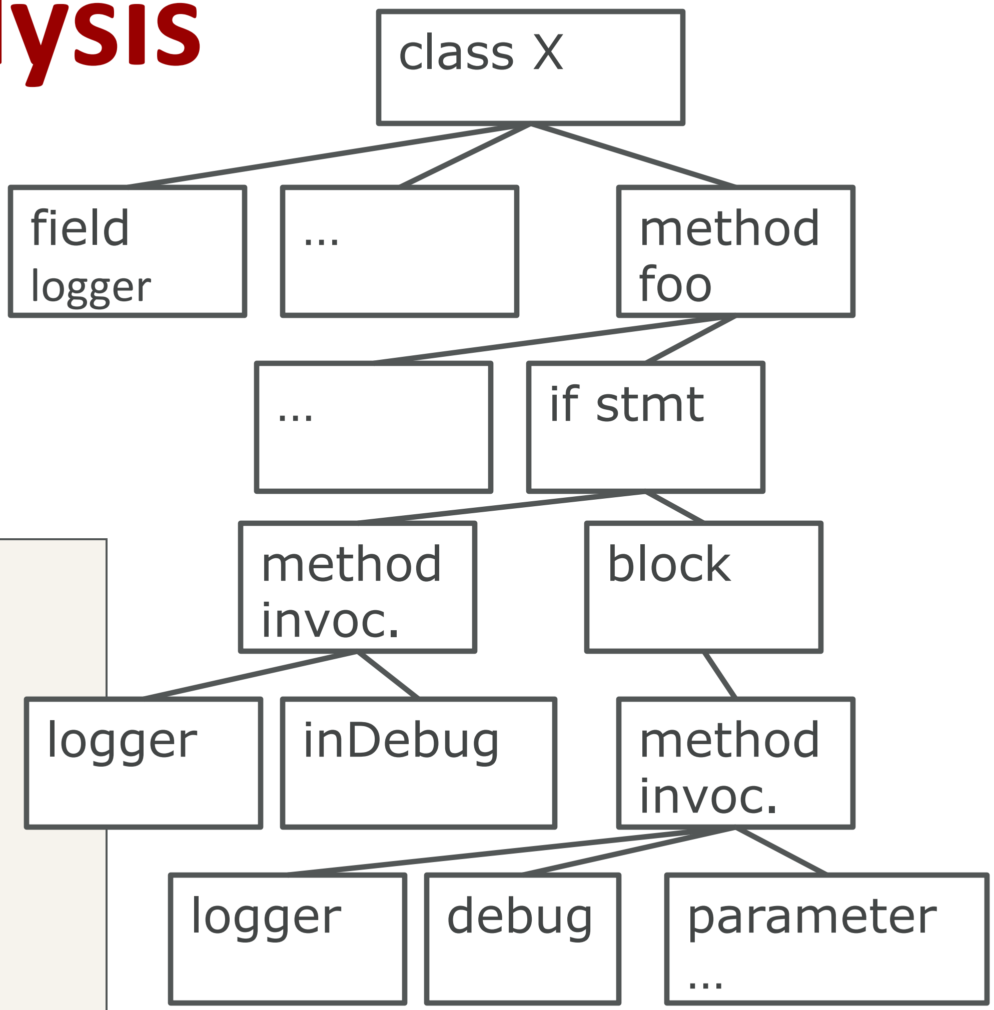


Type checking

```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug("We have " +
conn + "connections.");
    }
  }
}
class Logger {
   boolean inDebug() {…}
   void debug(String msg) {…}
}
```



Structural Analysis

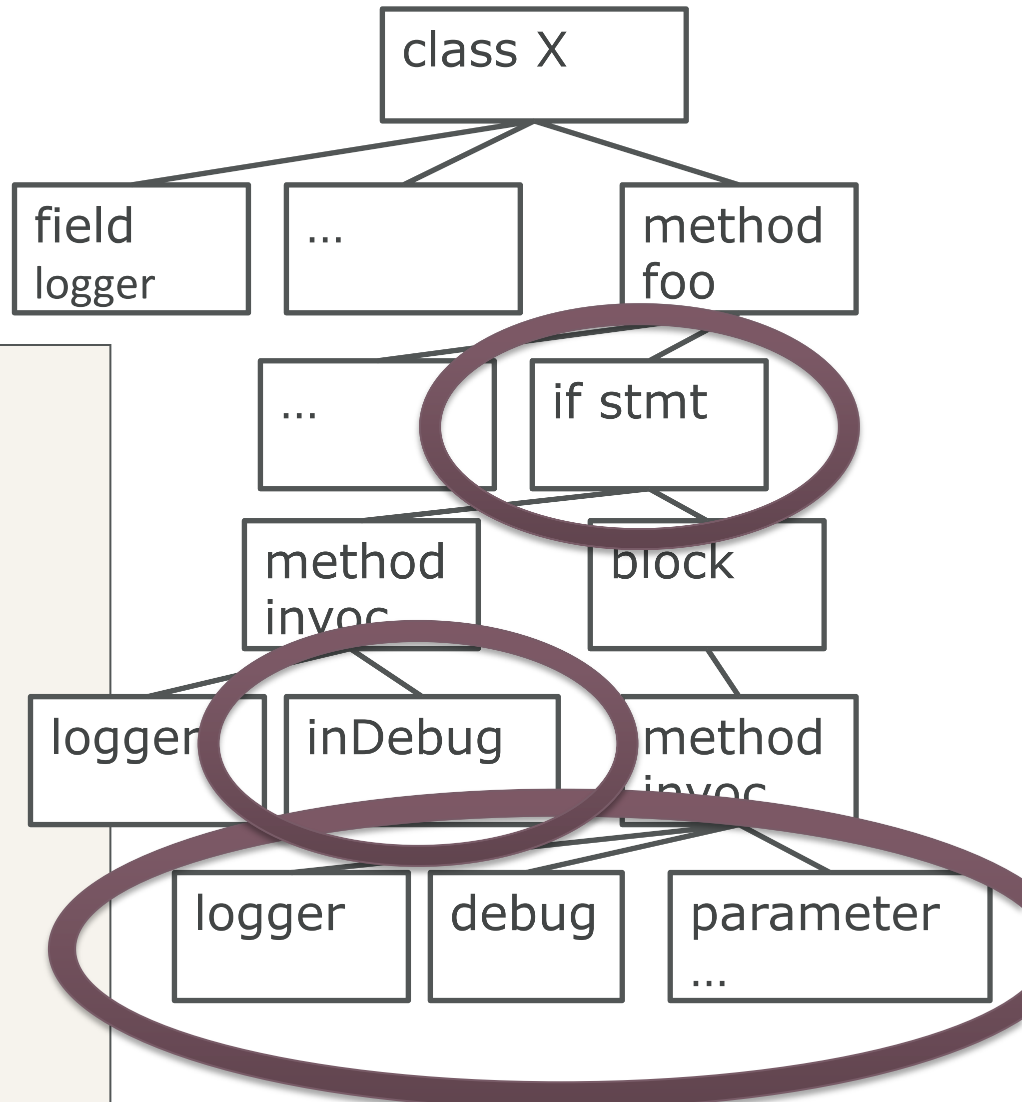



```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug("We have " +
conn + "connections.");
    }
  }
}
```

Abstract syntax tree walker

- Check that we don't create strings outside of a `Logger.inDebug` check
- Abstraction:
  - Look only for calls to `Logger.debug()`
  - Make sure they're all surrounded by if (`Logger.inDebug()`)
- Systematic: Checks all the code
- Known as an Abstract Syntax Tree (AST) walker
  - Treats the code as a structured tree
  - Ignores control flow, variable values, and the heap
  - Code style checkers work the same way

```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug(“We have ” +
conn + “connections.”);
    }
  }
}
class Logger {
   boolean inDebug() {…}
   void debug(String msg) {…}
}
```



Bug finding

```java
public Boolean decide() {
    if (computeSomething()==3)
        return Boolean.TRUE;
    if (computeSomething()==4)
        return false;
    return null;
}
```

overag History Bug Info Bug Expl

A.java: 69

Navigation

Bug: FBTest.decide() has Boolean return type and returns explicit null

A method that returns either Boolean.TRUE, Boolean.FALSE or null is an accident waiting to happen. This method can be invoked as though it returned a value of type boolean, and the compiler will insert automatic unboxing of the Boolean value. If a null value is returned, this will result in a NullPointerException.

Confidence: Normal, **Rank**: Troubling (14)
Pattern: NP_BOOLEAN_RETURN_NULL
Type: NP, **Category**: BAD_PRACTICE (Bad practice)

Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

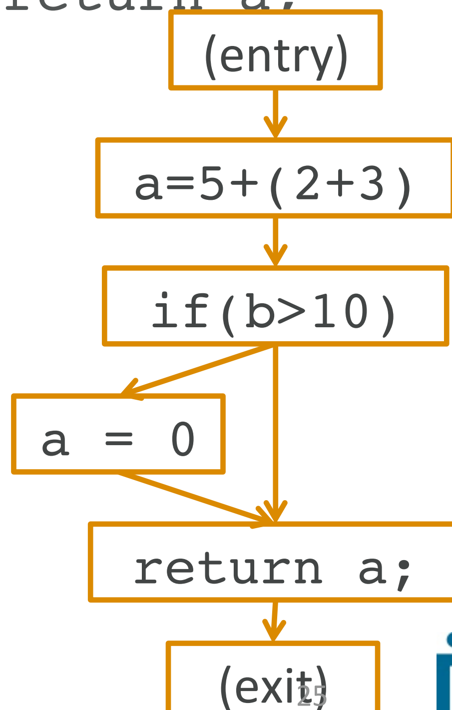
Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

Control flow graphs

- A tree/graph-based representation of the flow of control through the program.
  - Captures all possible execution paths.
- Each node is a basic block: no jumps in or out.
- Edges represent control flow options between nodes.
- Intra-procedural: within one function.
  - cf. inter-procedural

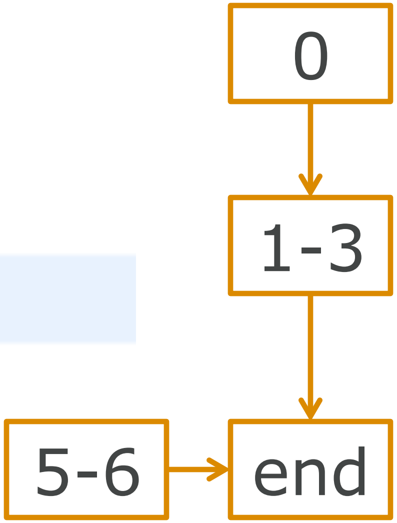
```
a = 5 + (2 + 3)
if (b > 10) {
   a = 0;
}
return a;
```



More on representation

- Basic definitions:
  - Nodes N – statements of program
  - Edges E – flow of control
  - pred(n) = set of all predecessors of n
  - succ(n) = set of all successors of n
  - Start node, set of final nodes (or one final node to which they all flow).
- **Program points**:
  - One program point before each node
  - One program point after each node
  - Join point: point with multiple predecessors
  - Split point: point with multiple successors

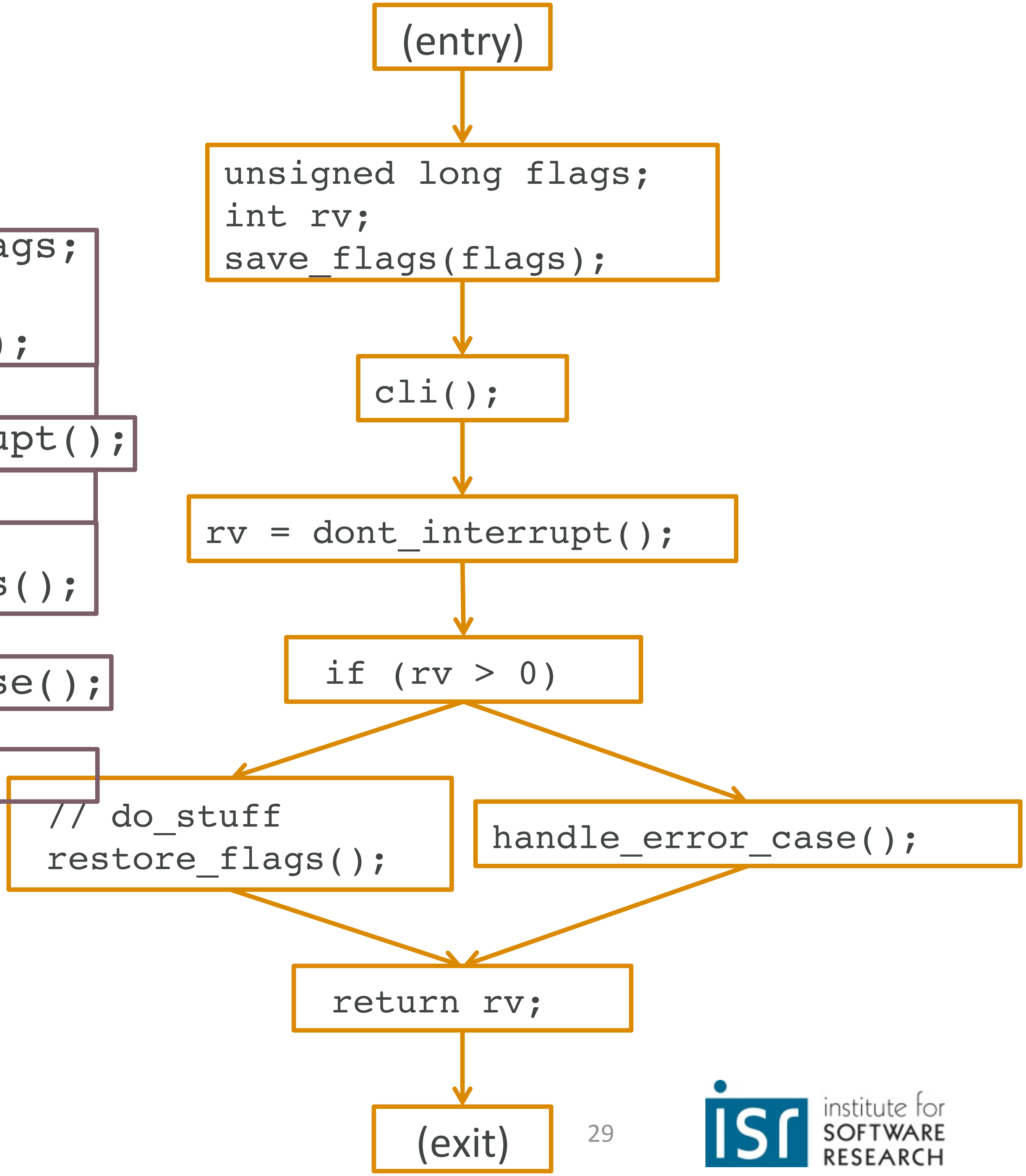
```
public int foo() {
    doStuff();

    return 3;

    doMoreStuff();
    return 4;
}
```



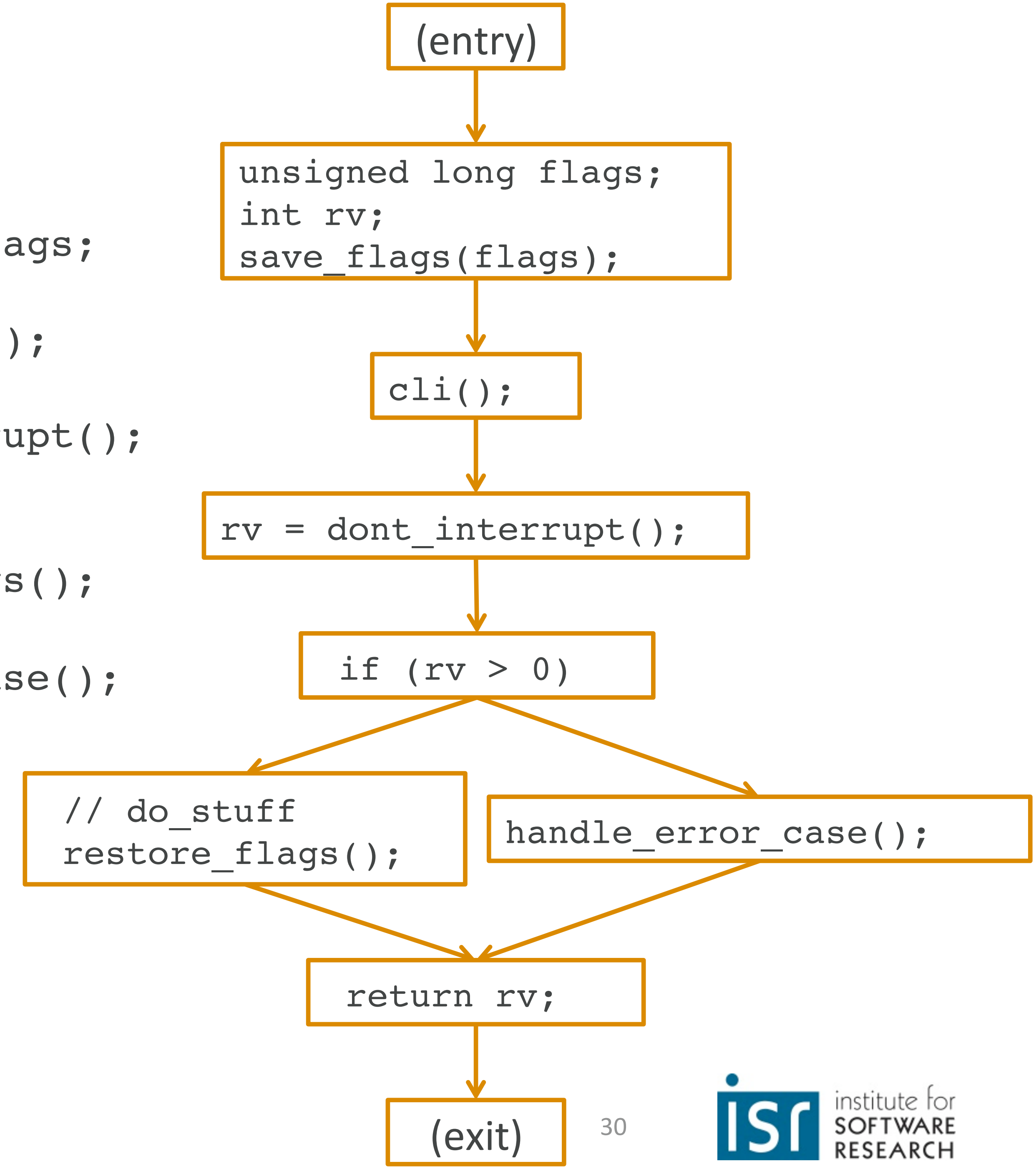
```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
     return NULL;
   sh->buffer_pool = bh -> b_next;
   bh->b_size = b_size;
   restore_flags(flags); // re-enables interrupts
   return bh;
}
```

With thanks to Jonathan Aldrich; example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

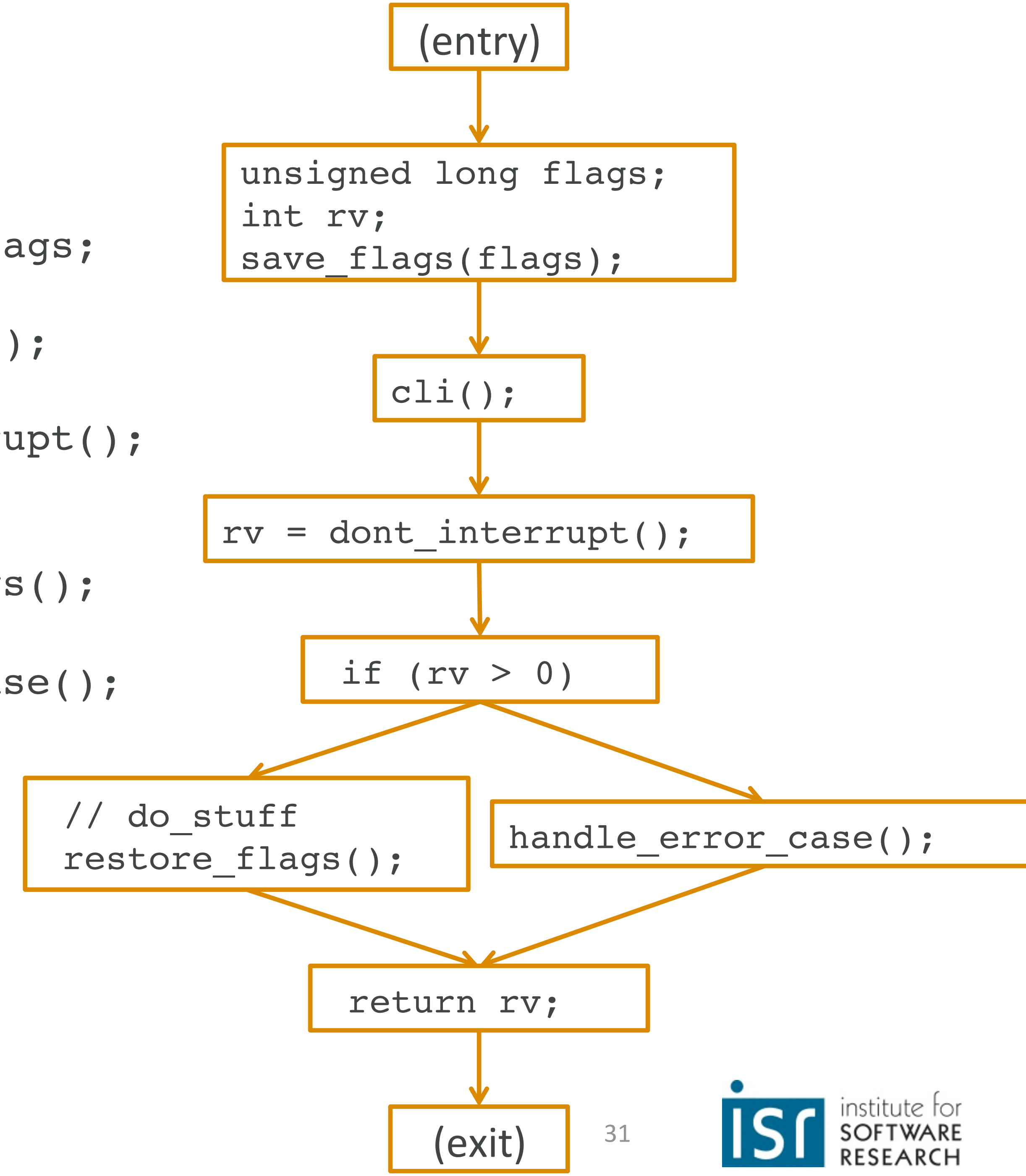
```
int foo() {
    unsigned long flags;
    int rv;
    save_flags(flags);
    cli();
    rv = dont_interrupt();
    if (rv > 0) {
       // do_stuff
       restore_flags();
    } else {
     handle_error_case();
    }
    return rv;
}
```



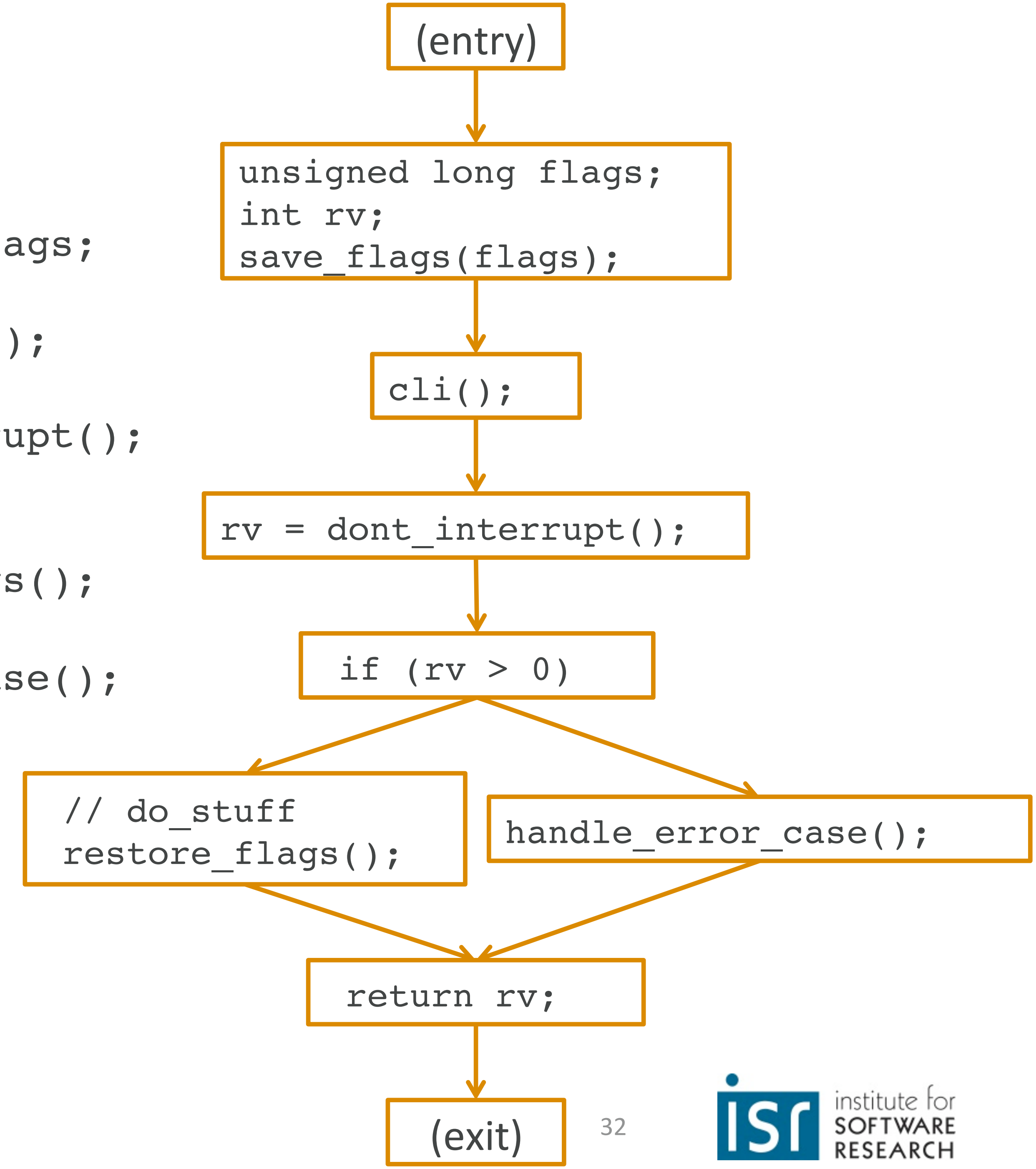
```
int foo() {
    unsigned long flags;
    int rv;
    save_flags(flags);
    cli();
    rv = dont_interrupt();
    if (rv > 0) {
        // do_stuff
         restore_flags();
    } else {
     handle_error_case();
    }
    return rv;
}
```



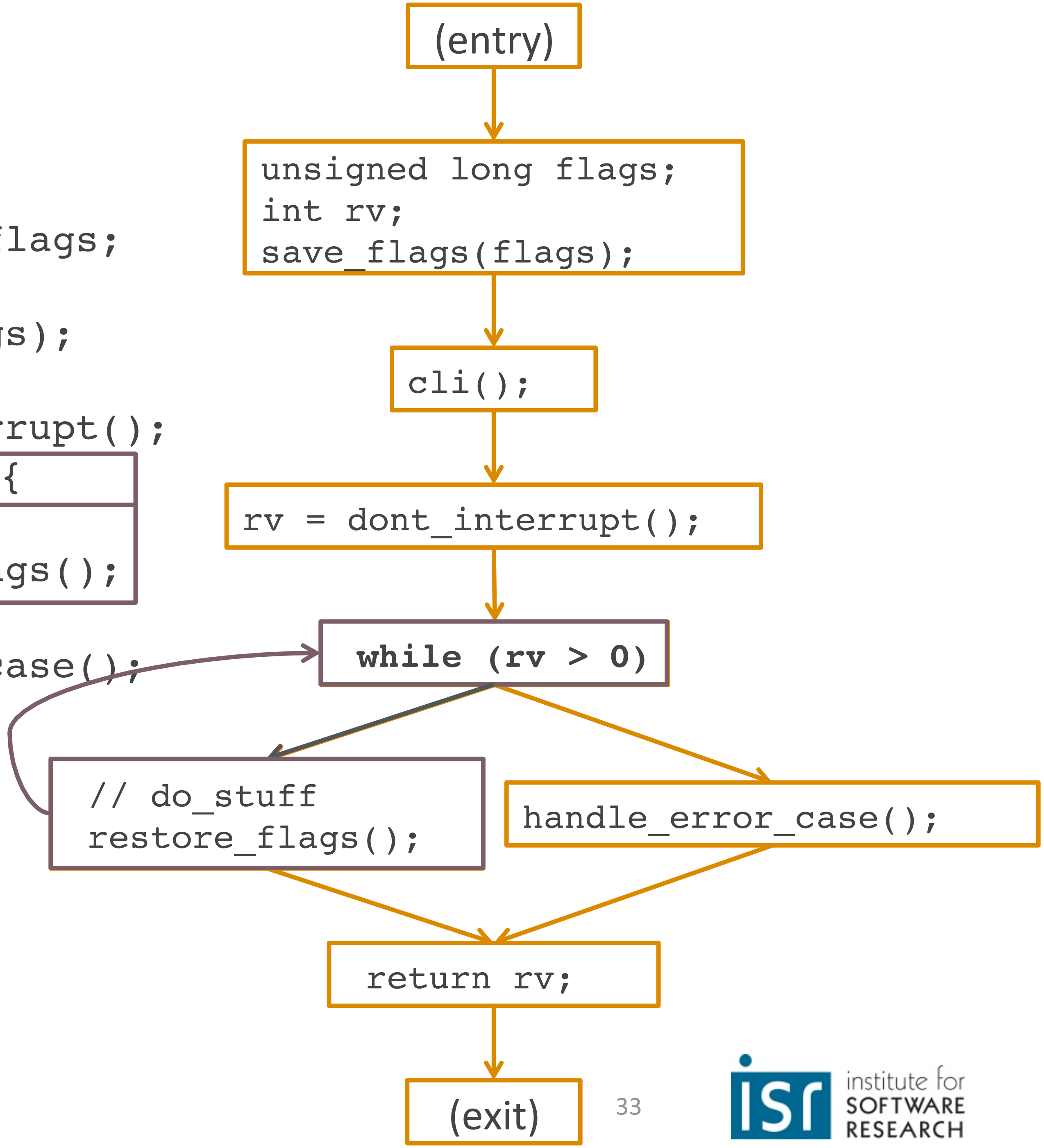
```
int foo() {
    unsigned long flags;
    int rv;
    save_flags(flags);
    cli();
    rv = dont_interrupt();
    while (rv > 0) {
       // do_stuff
        restore_flags();
    } else {
     handle_error_case();
    }
    return rv;
}
```



```
int foo() {
    unsigned long flags;
    int rv;
    save_flags(flags);
    cli();
    rv = dont_interrupt();
    while (rv > 0) {
        // do_stuff
         restore_flags();
    } else {
     handle_error_case();
    }
    return rv;
}
```



```
int foo() {
    unsigned long flags;
    int rv;
    save_flags(flags);
    cli();
    rv = dont_interrupt();
    while (rv > 0) {
        // do_stuff
         restore_flags();
    }
     handle_error_case();

    return rv;
}
```



institute for SOFTWARE RESEARCH

Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every program point.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

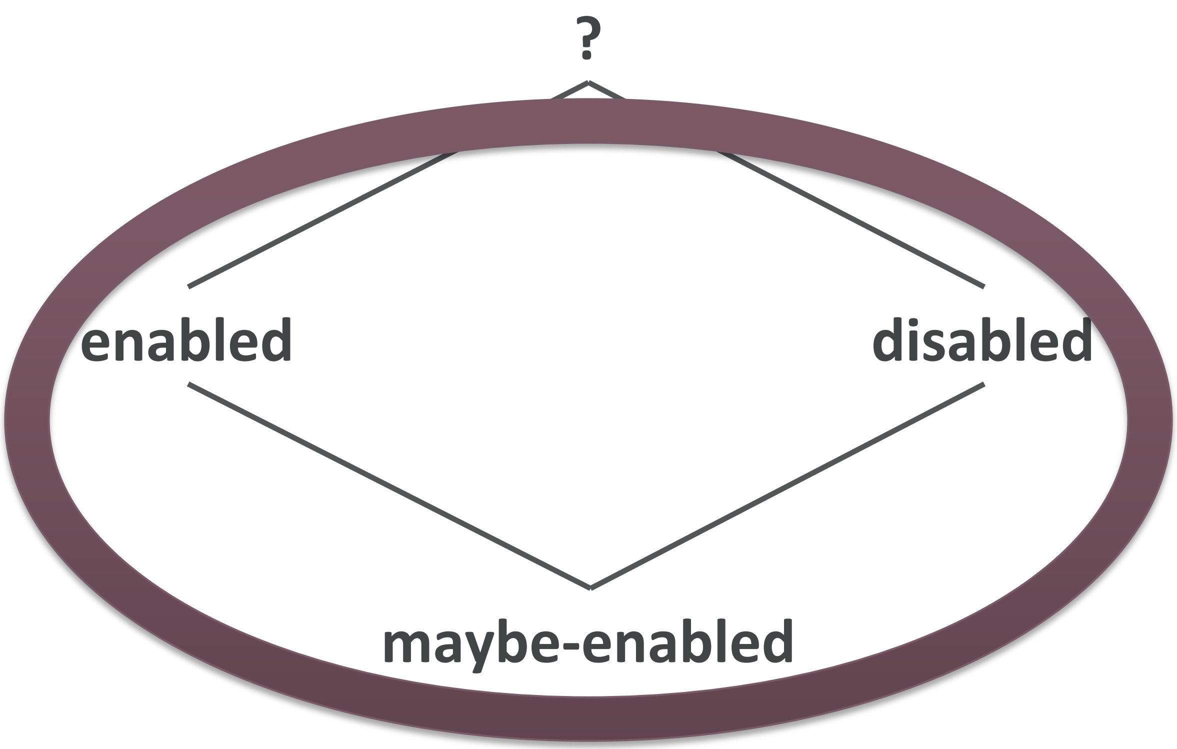
Abstract domain: lattices

- Lattice D = (S, r)
  - D is domain of program properties
  - S is a (possibly infinite) set of elements. Must contain unique *largest* (top) and *smallest* elements (bottom).
  - r is a binary relation over elements of S
- Required properties for r:
  - Is a partial order (reflexive, transitive, and anti-symmetric)
  - Every pair of elements has a unique **greatest lower bound** (*meet*) and a unique **least upper bound** (*join*)

Say wha?

- We are tracking all possible values related to a property of interest at every program point.
- Possible values---the information we're tracking---modeled as an element of the lattice that defines the domain.
- Use the lattice to compute information, by building constraints that describe how the information changes through the program:
  - **Transfer function:** Effect of instructions on state
  - **Meet/join:** effect of control flow

Example: interrupt checker



An interrupt checker

- **Abstraction**
  - Three abstract states: enabled, disabled, maybe-enabled
  - Warning if we can reach the end of the function with interrupts disabled.
- Transfer function:
  - If a basic block includes a call to `cli()`, then it moves the state of the analysis from **disabled** to **enabled**.
  - If a basic block includes a call to `restore_flags()`, then it moves the state of the analysis from **enabled** to **disabled**.

assume: pre-block program point: interrupts disabled

```
cli();
```

post-block program point: interrupts enabled assume: pre-block program point: interrupts enabled

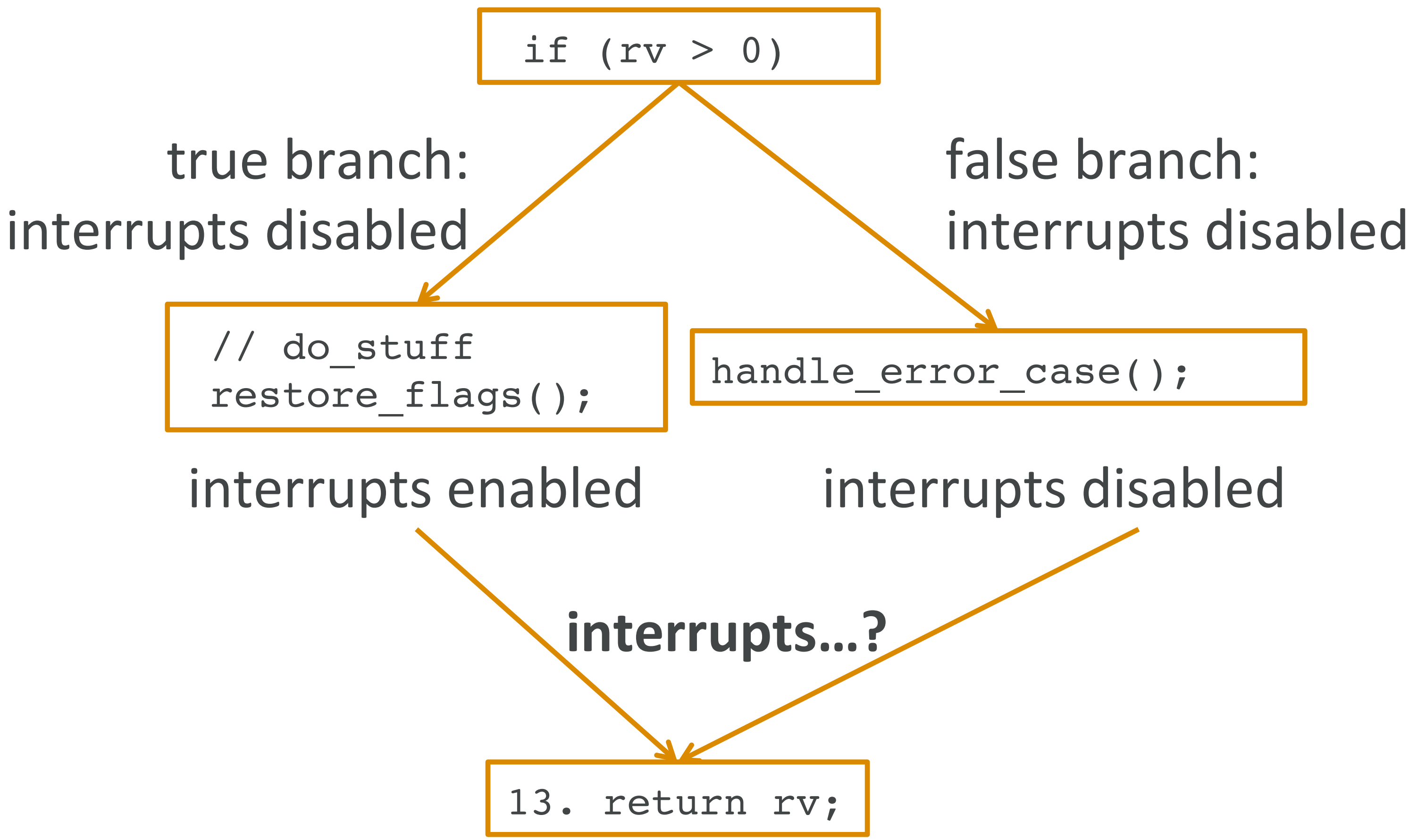
```
// do_stuff
restore_flags();
```

post-block program point: interrupts disabled

(Note that, in graphs, I leave out some intermediate program points when they're not interesting; you'll see what I mean in a second.)

Join

assume: pre-block program point: interrupts disabled



Join/branching

- What to do with information that comes to/from multiple previous states?
- When we get to a branch, what should we do?
  1. explore each path separately
     - Most exact information for each path
     - But—how many paths could there be?
     - Leads to state explosion, loops add an infinity problem. join paths back together
  2. Join!
     - Less exact, loses information (…Rice's theorem...)
     - But no state explosion, and terminates (more in a bit)
- Not just conditionals!
  – Loops, switch, and exceptions too!

Interrupt analysis: join function

- Abstraction
  - 3 states: enabled, disabled, maybe-enabled
  - Program counter
- **Join:** If at least one predecessor to a basic block has interrupts enabled and at least one has them disabled…

Join

- Join(enabled, enabled) → enabled
- Join(disabled, disabled) → disabled
- Join(disabled, enabled) → maybe-enabled
- Join(maybe-enabled, *) → maybe-enabled

Join: abstract!

assume: pre-block program point: interrupts disabled

```
if (rv > 0)
```

true branch:
interrupts disabled

false branch:
interrupts disabled

```
// do_stuff
restore_flags();
```

```
handle_error_case();
```

interrupts enabled

interrupts disabled

Join(enabled, disabled) → maybe enabled

```
13. return rv;
```

(Note: this is where information gets "lost.")

institute for SOFTWARE RESEARCH

Control/Dataflow analysis

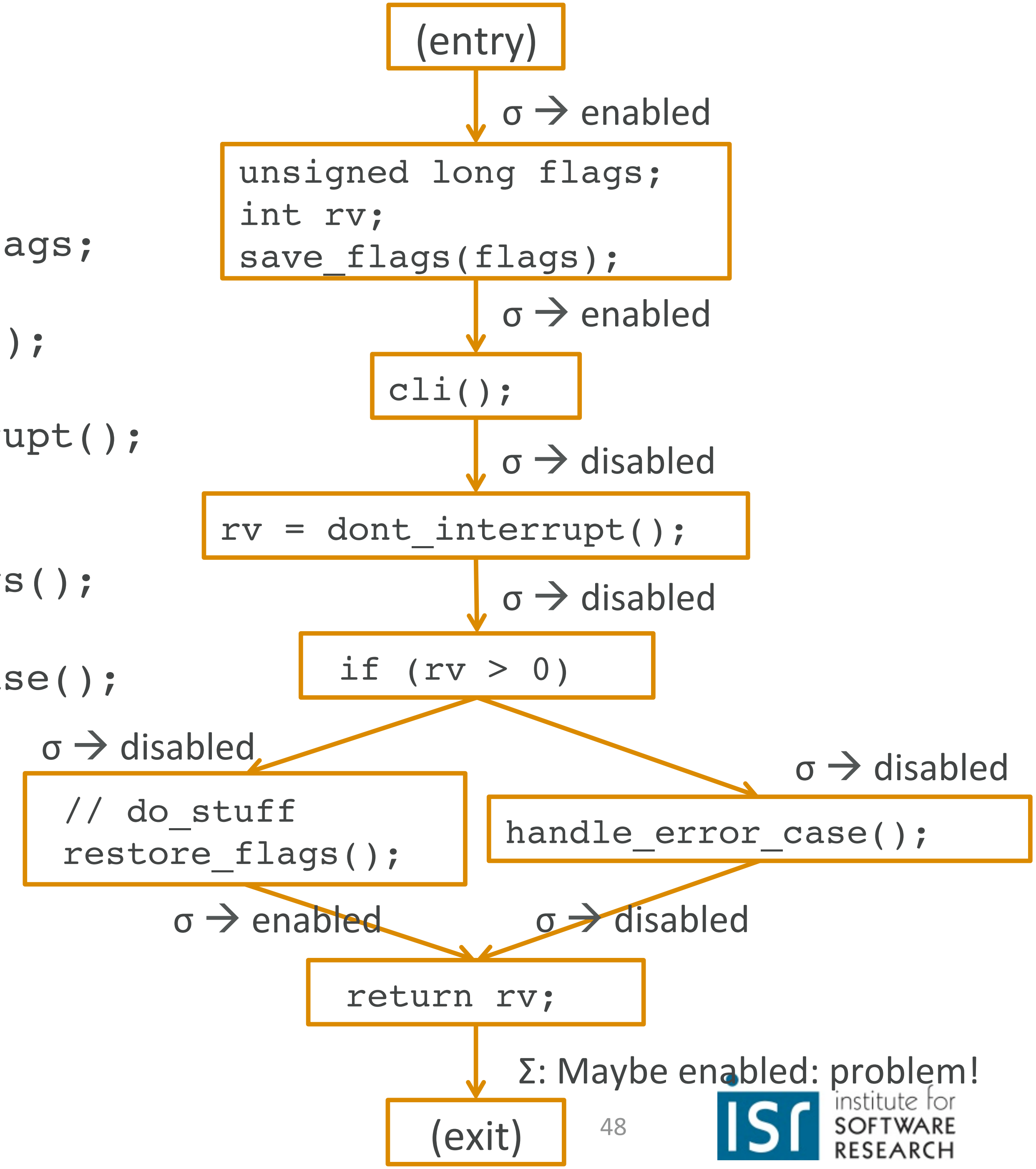
- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every program point.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

institute for SOFTWARE RESEARCH

Reasoning about a CFG

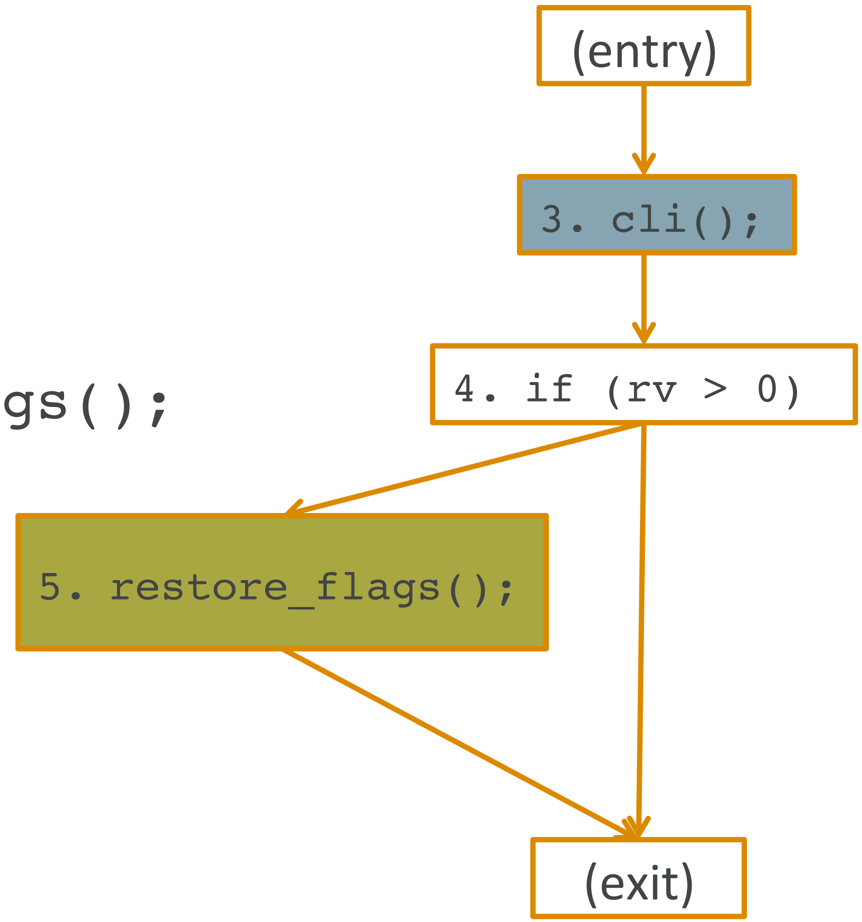
- Analysis updates state at *program points:* points between nodes.
- For each node:
  – determine state on entry by examining/combining state from predecessors.
  – evaluate state on exit of node based on effect of the operations (*transfer)*.
- *Iterate through successors and over entire graph until the state at each program point stops changing.*
- **Output: state at each program point**

```
int foo() {
    unsigned long flags;
    int rv;
    save_flags(flags);
    cli();
    rv = dont_interrupt();
    if (rv > 0) {
       // do_stuff
        restore_flags();
    } else {
     handle_error_case();
    }
    return rv;
}
```



Abstraction

```
1.  void foo() {
2.      …
3.      cli();
4.    if (a) {
5.        restore_flags();
6.      }
7.  }
```



Data- vs. control-flow

- Dataflow: tracks abstract values for each of (some subset of) the variables in a program.
- Control flow: tracks state global to the function in question.

Zero/Null-pointer Analysis

- Could a variable x ever be 0?
  - (what kinds of errors could this check for?)
- Original domain: N maps every variable to an integer.
- Abstraction: every variable is non zero (NZ), zero(Z), or maybe zero (MZ)

Zero analysis transfer

- What operations are relevant?

Zero analysis join

- Join(zero, zero) → zero
- Join(not-zero, not-zero) → not-zero
- Join(zero, not-zero) → maybe-zero
- Join(maybe-zero, *) → maybe-zero

Example

- Consider the following program:

```
x = 10;
y = x;
z = 0;
while (y > -1) {
   x = x/y;
   y = y-1;
   z = 5;
}
```

- Use **zero analysis** to determine if y could be zero at the division.

Learning goals

- Give a one sentence definition of static analysis. Explain what types of bugs static analysis targets.
- Give two examples of syntactic or structural static analysis.
- Construct basic control flow graphs for small examples by hand.
- Distinguish between control- and data-flow analyses; define and then step through on code examples simple control and data-flow analyses.