

# Lecture 1: Introduction to Program Analysis

17-355/17-665/17-819: Program Analysis

Rohan Padhye

Aug 26, 2025

\* Course materials developed with Jonathan Aldrich and Claire Le Goues

# Introductions



Prof. Rohan Padhye



TA Vasu Vikram

# My Background

- Involved with program analysis for ~12 years.
- PhD from UC Berkeley, Masters from IIT Bombay (India)
  - Contributed to research on fuzz testing, static interprocedural analysis, performance analysis, etc.
- Worked at IBM Research, Microsoft Research, Samsung Research America, and Amazon Web Services
  - Developed tools for improving developer productivity, finding input-validation software bugs, identifying security vulnerabilities in mobile systems, discovering concurrency issues in distributed systems, fuzzing cloud-hosted databases etc.
- Advising PhD students in CMU's Software Engineering program
  - I lead the Program Analysis, Software Testing, and Applications research group (PASTA lab)



IBM Research



Microsoft Research



# Learning objectives

- Provide a high level definition of program analysis and give examples of why it is useful.
- Sketch the explanation for why all analyses must be approximate.
- Understand the course mechanics and be motivated to read the syllabus.
- Describe the function of an AST and outline the principles behind AST walkers for simple bug-finding analyses.
- Recognize the basic WHILE demonstration language and translate between WHILE and While3Addr.

# What is this course about?

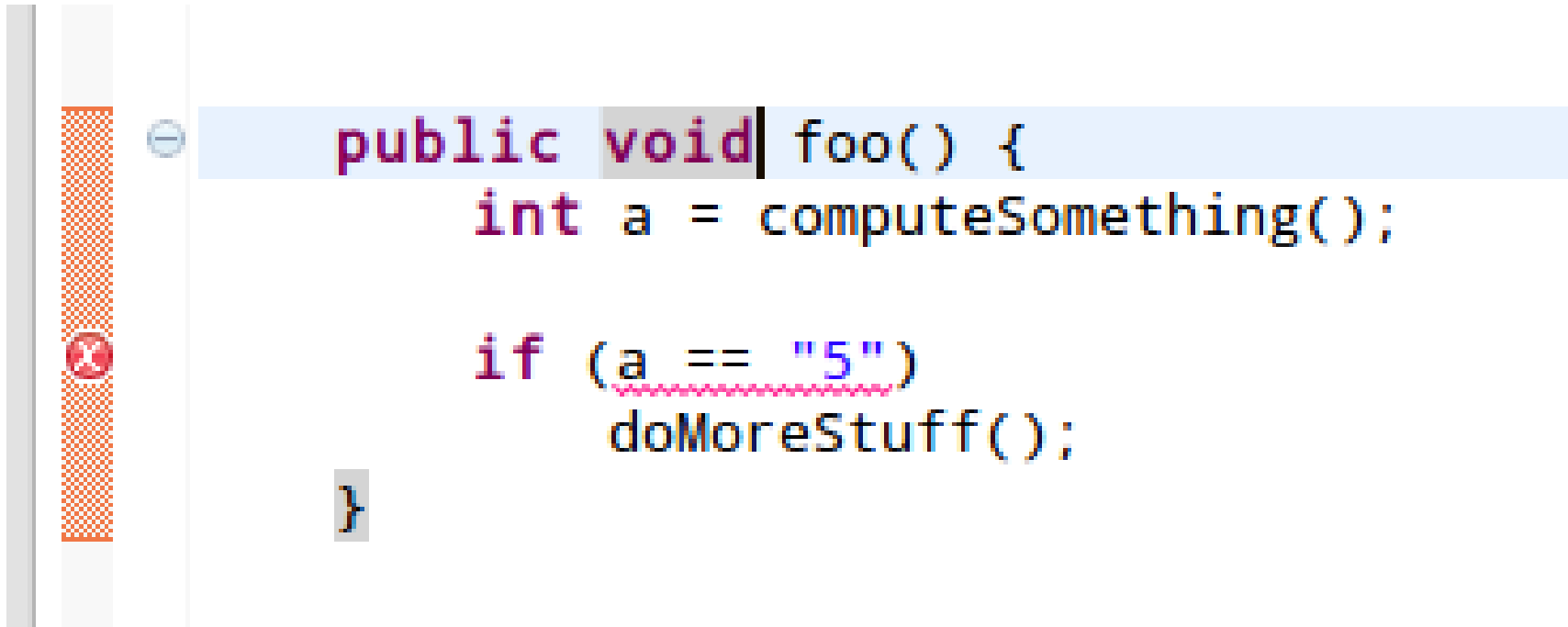
- Program analysis is the systematic examination of a program to determine its properties.
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- We will learn:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

# Why might you care?

Program analysis, and the skills that underlie it, have implications for:

- Automatic bug finding
- Language design and implementation (compilers, VMs)
- Program transformation (refactoring, optimization, repair)
- Program synthesis

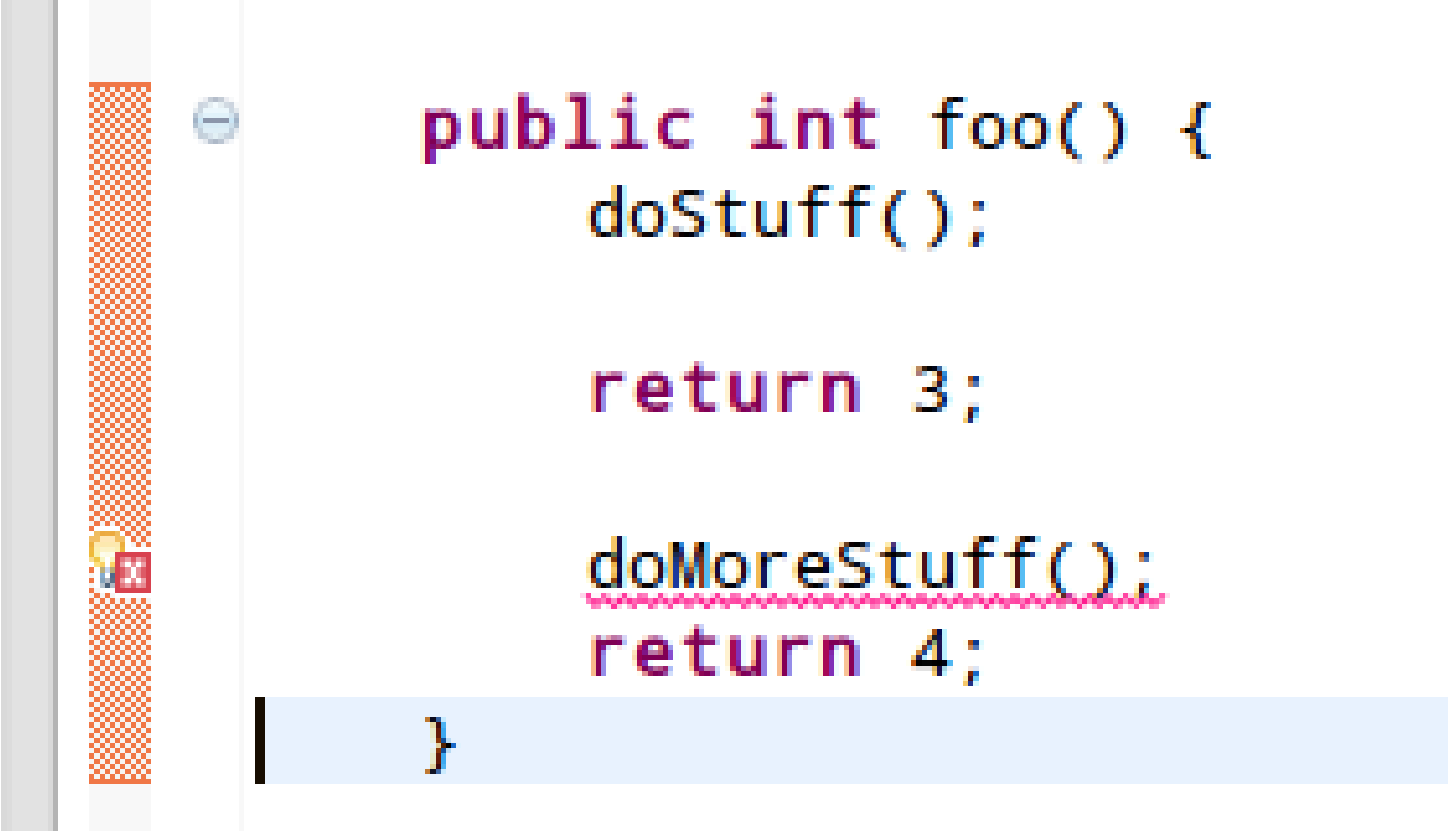
# You've seen it before



A screenshot of a code editor interface. On the left, there is a vertical scrollbar and a line of code is highlighted in blue. The code is a Java method definition. The first line is `public void foo() {`. The second line is `int a = computeSomething();`. The third line is `if (a == "5")`, where the string `"5"` is underlined with a red squiggly line, indicating a type mismatch error. The fourth line is `doMoreStuff();`. The fifth line is `}`. The error icon is a red circle with a white 'x' inside, located to the left of the `if` statement.

```
public void foo() {  
    int a = computeSomething();  
  
    if (a == "5")  
        doMoreStuff();  
}
```

# You've seen it before



```
public int foo() {  
    doStuff();  
  
    return 3;  
  
    doMoreStuff();  
    return 4;  
}
```



# Lots of tools available

## Lint

```
//depot/google3/java/com/google/devtools/staticanalysis/Test.java
package com.google.devtools.staticanalysis;

public class Test {
    public boolean foo() {
        return getString() == "foo".toString();
    }

    public String getString() {
        return new String("foo");
    }
}
```

**Apply** **Cancel**

```
package com.google.devtools.staticanalysis;
import java.util.Objects;

public class Test {
    public boolean foo() {
        return Objects.equals(getString(), "foo".toString());
    }

    public String getString() {
        return new String("foo");
    }
}
```

**Lint** Missing a Javadoc comment.  
Java  
1:02 AM, Aug 21  
[Please fix](#) [Not useful](#)

```
public boolean foo() {
    return getString() == "foo".toString();
}
```

**ErrorProne** String comparison using reference equality instead of value equality  
StringEquality  
1:03 AM, Aug 21  
(see <http://code.google.com/p/error-prone/wiki/StringEquality>)  
[Please fix](#) [Not useful](#)  
Suggested fix attached: [show](#)

```
}

public String getString() {
    return new String("foo");
}
```

ErrorProne

github.com/marketplace?category=code-quality

Search or jump to...

Pull requests Issues Marketplace Explore

Marketplace / Search results

Types

Apps

Actions

Categories

API management

Chat

**Code quality**

Code review

Continuous integration

Dependency management

Deployment

IDEs

Learning

Localization

Mobile

Monitoring

Project management

Publishing

Recently added

Security

Support

Testing

Utilities

Filters

Verification

Verified

Unverified

Your items

Purchases

Also recommended for you

245 results filtered by Code quality

**Code quality**

Automate your code review with style, quality, security, and test-coverage checks when you need them.

**CodeScene** The analysis tool to identify and prioritize technical debt and evaluate your organizational efficiency

**TestQuality** Modern, powerful, test plan management

**CodeFactor** Automated code review for GitHub

**Restyled.io** Restyle Pull Requests as they're opened

**DeepScan** Advanced static analysis for automatically finding runtime errors in JavaScript code

**LGTM** Find and prevent zero-days and other critical bugs, with customizable alerts and automated code review

**Datree** Policy enforcement solution for confident and compliant code

**Lucidchart Connector** Insert a public link to a Lucidchart diagram so team members can quickly understand an issue or pull request

**DeepSource** Discover bug risks, anti-patterns and security vulnerabilities before they end up in production. For Python and Go

**Code Inspector** Code Quality, Code Reviews and Technical Debt evaluation made easy

**Codecov** Group, merge and compare coverage reports

**codebeat** Code review expert on demand. Automated for mobile and web

**Codacy** Automated code reviews to help developers ship better software, faster

**Better Code Hub** A Benchmarked Definition of Done for Code Quality

**Code Climate** Automated code review for technical debt and test coverage

**Coveralls** Ensure that new code is fully covered, and see coverage trends emerge. Works with any CI service

**Sider** Automatically analyze pull request against custom per-project rulesets and best practices

**Imgbot** A GitHub app that optimizes your images

**codelingo** Your Code, Your Rules - Automate code reviews with your own best practices

**Check TODO** Checks for any added or modified TODO items in a Pull Request

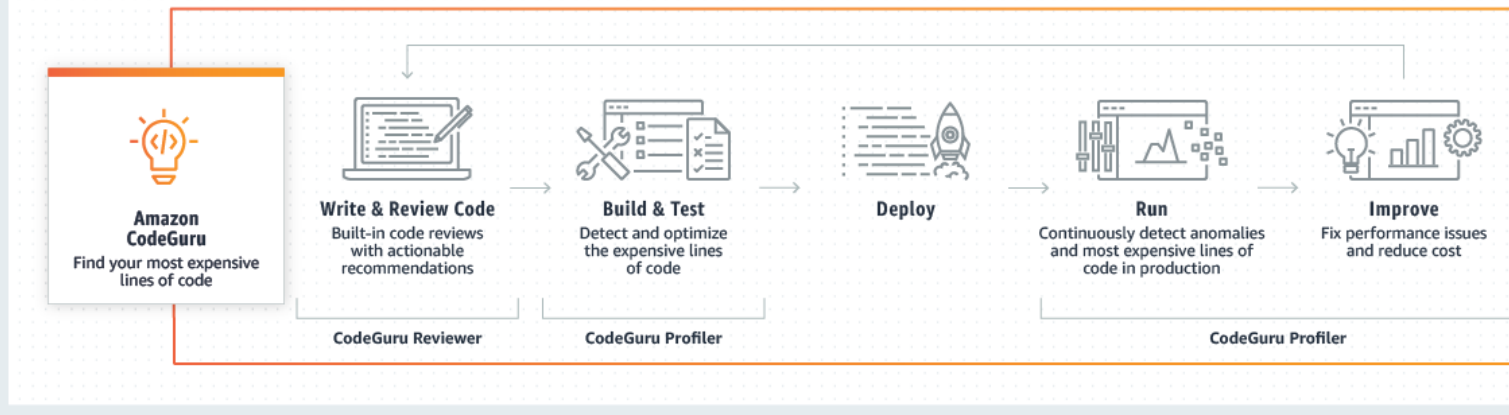
[Previous](#) [Next](#)

<https://github.com/marketplace?category=api-management>

# Advanced examples from industry

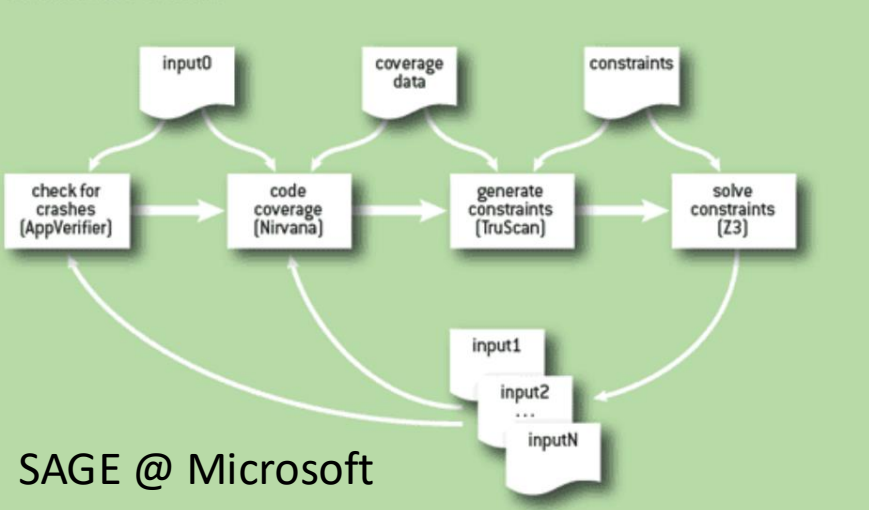
GitHub CoPilot

## CodeGuru @ Amazon

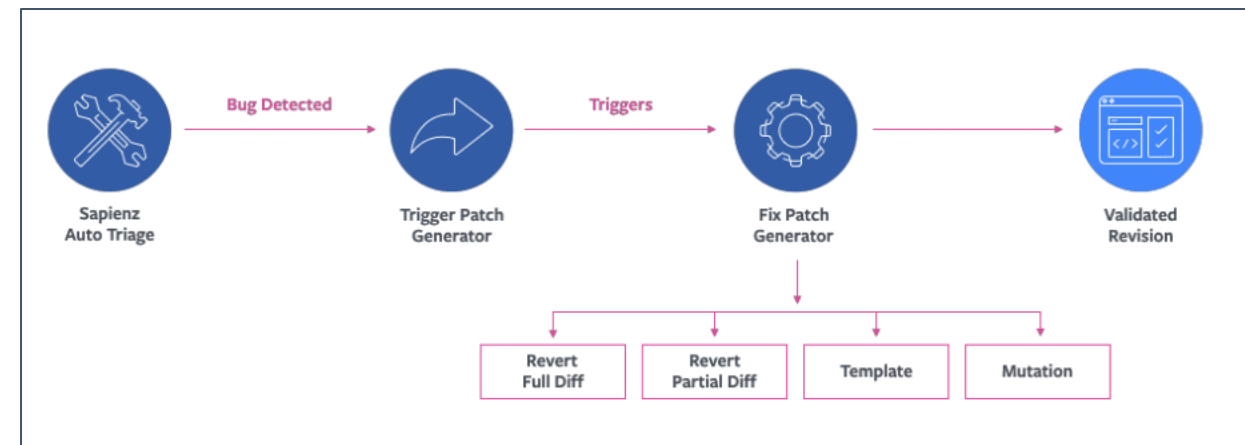


```
1 #!/usr/bin/env ts-node
2
3 import { fetch } from "fetch-h2";
4
5 // Determine whether the sentiment of text is positive
6 // Use a web service
7 async function isPositive(text: string): Promise<boolean> {
8   const response = await fetch('http://text-processing.com/api/sentiment/', {
9     method: "POST",
10    body: 'text=${text}',
11    headers: {
12      "Content-Type": "application/x-www-form-urlencoded",
13    },
14  });
15  const json = await response.json();
16  return json.label === "pos";
17 }
```

## Architecture of SAGE



## Sapienz and SapFix @ Facebook



# What kinds of issues does program analysis help find?

Defects that result from inconsistently following simple 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

Is there a bug in this code?

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.    struct buffer_head *bh;
6.    unsigned long flags;
7.    save_flags(flags);
8.    cli(); // disables interrupts
9.    if ((bh = sh->buffer_pool) == NULL)
10.        return NULL;
11.    sh->buffer_pool = bh -> b_next;
12.    bh->b_size = b_size;
13.    restore_flags(flags); // re-enables interrupts
14.    return bh;
15.}
```

Part of the spec:  
Interrupts should not be  
disabled upon function return

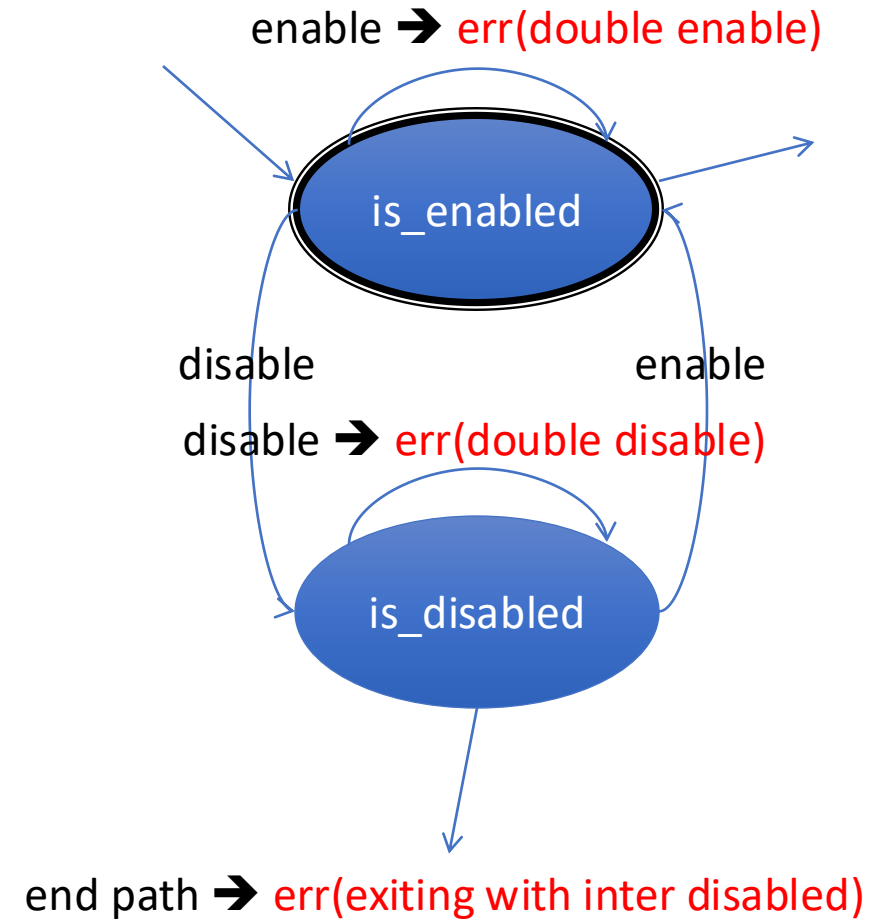
Example from Engler et al., *Checking system rules Using  
System-Specific, Programmer-Written Compiler  
Extensions*, OSDI '000

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.    struct buffer_head *bh;
6.    unsigned long flags;
7.    save_flags(flags);
8.    cli(); // disables interrupts
9.    if ((bh = sh->buffer_pool) == NULL)
10.        return NULL;
11.    sh->buffer_pool = bh -> b_next;
12.    bh->b_size = b_size;
13.    restore_flags(flags); // re-enables interrupts
14.    return bh;
15.}
```

ERROR: function returns with  
interrupts disabled!

Example from Engler et al., *Checking system rules Using  
System-Specific, Programmer-Written Compiler  
Extensions*, OSDI '000

# Abstract Model



```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.    struct buffer_head *bh;
6.    unsigned long flags;
7.    save_flags(flags);
8.    cli(); // disables interrupts
9.    if ((bh = sh->buffer_pool) == NULL)
10.        return NULL;
11.    sh->buffer_pool = bh -> b_next;
12.    bh->b_size = b_size;
13.    restore_flags(flags); // re-enables interrupts
14.    return bh;
15.}
```



Initial state: is\_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000



```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.    struct buffer_head *bh;
6.    unsigned long flags;
7.    save_flags(flags);
8.    cli(); // disables interrupts
9.    if ((bh = sh->buffer_pool) == NULL)
10.        return NULL;
11.    sh->buffer_pool = bh -> b_next;
12.    bh->b_size = b_size;
13.    restore_flags(flags); // re-enables interrupts
14.    return bh;
15.}
```



Transition to: is\_disabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.               int b_size) {
5.    struct buffer_head *bh;
6.    unsigned long flags;
7.    save_flags(flags);
8.    cli(); // disables interrupts
9.    if ((bh = sh->buffer_pool) == NULL)
10.        return NULL;
11.    sh->buffer_pool = bh -> b_next;
12.    bh->b_size = b_size;
13.    restore_flags(flags); // re-enables interrupts
14.    return bh;
15.}
```



Final state: is\_disabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```

1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.    struct buffer_head *bh;
6.    unsigned long flags;
7.    save_flags(flags);
8.    cli(); // disables interrupts
9.    if ((bh = sh->buffer_pool) == NULL)
10.        return NULL;
11.    sh->buffer_pool = bh -> b_next;
12.    bh->b_size = b_size;
13.    restore_flags(flags); // re-enables interrupts
14.    return bh;
15.}

```

Transition to: is\_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.    struct buffer_head *bh;
6.    unsigned long flags;
7.    save_flags(flags);
8.    cli(); // disables interrupts
9.    if ((bh = sh->buffer_pool) == NULL)
10.        return NULL;
11.    sh->buffer_pool = bh -> b_next;
12.    bh->b_size = b_size;
13.    restore_flags(flags); // re-enables interrupts
14.    return bh;
15.}
```



Final state: is\_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

# Behavior of interest...

- Is on uncommon execution paths.
  - Hard to exercise when testing.
- Executing (or analyzing) all paths is infeasible
- **Instead: (abstractly) check the entire possible state space of the program.**

# What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- We will learn:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

# What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- Principal techniques:
  - **Dynamic:**
    - **Testing:** Direct execution of code on test data in a controlled environment.
    - **Analysis:** Tools extracting data from test runs.
  - **Static:**
    - **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
    - **Analysis:** Tools reasoning about program behavior without executing it.
    - **Inference:** Statistical models of code (e.g., AI / ML)
  - ...and their combination.

# What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- Principal techniques:
  - **Dynamic:**
    - **Testing:** Direct execution of code on test data in a controlled environment.
    - **Analysis:** Tools extracting data from test runs.
  - **Static:**
    - **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
    - **Analysis:** Tools reasoning about program behavior without executing it.
    - **Inference:** Statistical models of code (e.g., AI / ML)
  - ...and their combination.



# The Bad News: Rice's Theorem

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

Henry Gordon Rice, 1953

# Proof by contradiction (sketch)

Assume that you have a function that can determine if a program  $p$  has some nontrivial property (like `divides_by_zero`):

```
1.  int silly(program p, input i) {  
2.      p(i);  
3.      return 5/0;  
4.  }  
5.  bool halts(program p, input i) {  
6.      return divides_by_zero(`silly(p,i)`);  
7.  }
```

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

Over-approximate analysis:

reports all potential defects

-> no false negatives

-> subject to false positives

Under-approximate analysis:

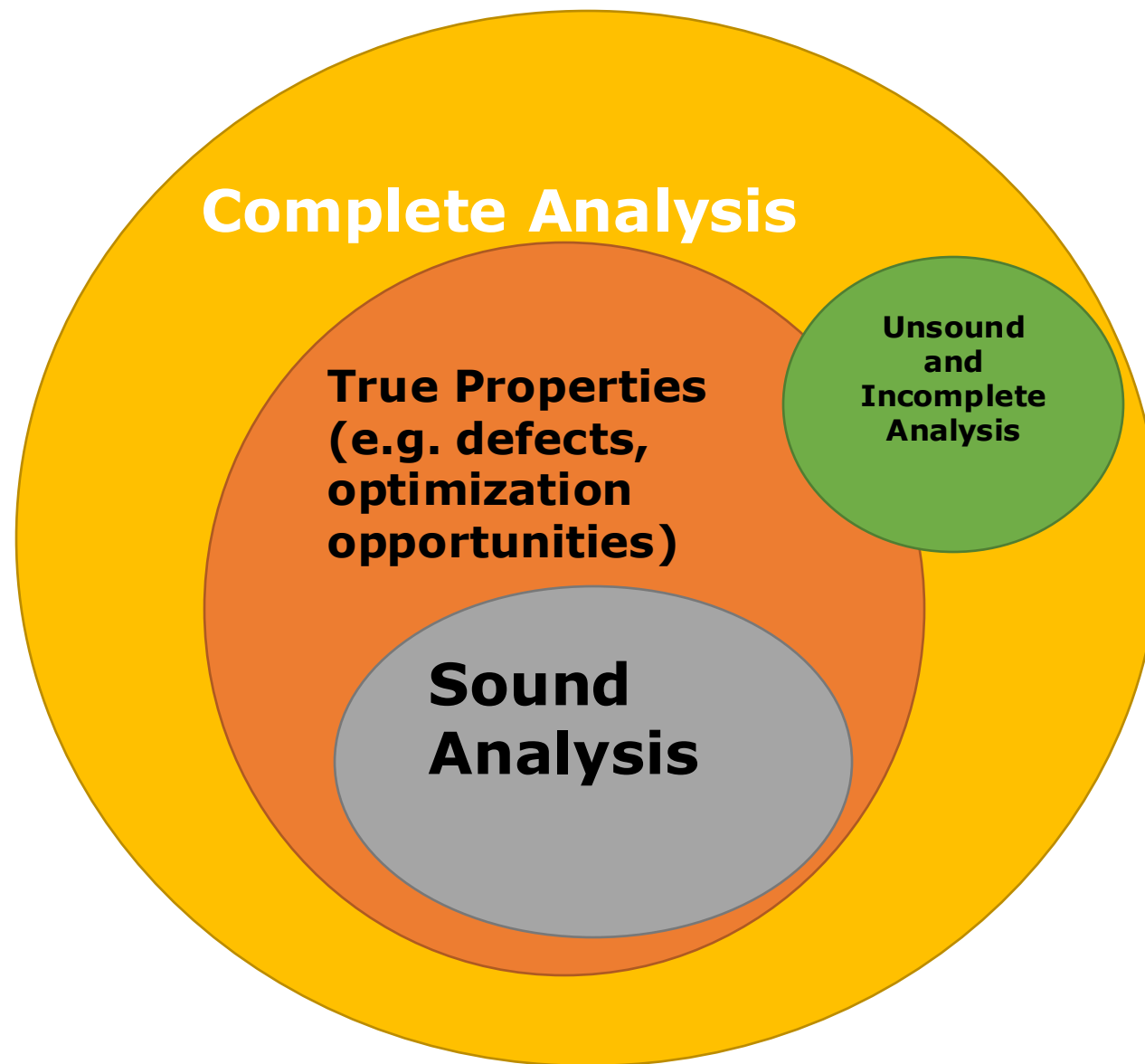
every reported defect is an actual defect

-> no false positives

-> subject to false negatives

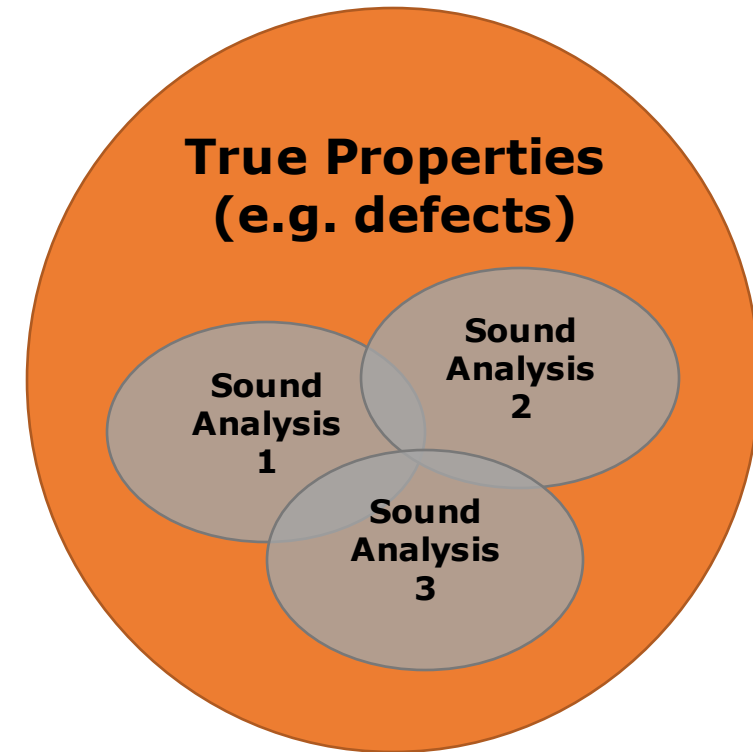
# Soundness and Completeness

- An analysis is “sound” if every claim it makes is true
- An analysis is “complete” if it makes every true claim
- Soundness/Completeness correspond to under/over-approximation depending on context.
  - E.g. compilers and verification tools treat “soundness” as over-approximation since they make claims over all possible inputs
  - E.g. code quality tools often treat “sound” analyses as under-approximation because they make claims about existence of bugs



# Soundness and Completeness Tradeoffs

- Sound + Complete is impossible in general (Rice's theorem)
- Most practical tools attempt to be either sound or complete for some specific application, using approximation
- Multiple classes of sound/complete techniques may exist, with trade-offs for accuracy and performance.
- Program analysis is a rich field because of the constant and never-ending battle to balance these trade-offs with ever-increasing software complexity



# Course overview

# Course topics

- Program representation
- Abstract interpretation: Use abstraction to reason about possible program behavior.
  - Operational semantics.
  - Dataflow Analysis
  - Termination, complexity
  - Widening, collecting
  - Interprocedural analysis
  - Pointer analysis
  - Control flow analysis
- Hoare-style verification: Make logical arguments about program behavior.
  - Axiomatic semantics
- Model checking (briefly) : reason about all possible program states.
  - Take 15-414 if you want the full treatment!
- Symbolic execution: test all possible executions paths simultaneously.
  - Concolic execution / test generation
- Grey-box analysis for fuzz testing
  - Take 17-712 if you want the full treatment!
- Dynamic analysis for race detection
- Program synthesis
- Program repair
- We will *not* cover types.
- We *may* touch on recent AI-based methods



# What to expect

- Beautiful and elegant theory (15-251 is a soft pre-req)
  - Mostly discrete mathematics, symbolic reasoning, inductive proofs
  - This is traditionally a “white-board” course [may not always use slides]
- Build awesome tools
  - Engineering of program analyses, compilers, and bug finding tools make great use of many fundamental ideas from computer science and software engineering
- New way to think about programs (15-150 or 15-214 soft pre-reqs)
  - Representations, control/data-flow, input state space
- Appreciate the limits and achievements in the space
  - What tools are *impossible* to build?
  - What tools are *impressive* that they exist at all?
  - When is it appropriate to use a particular analysis tool versus another?
  - How to interpret the results of a program analysis tool?

# Fundamental concepts

- Abstraction
  - Elide details of a specific implementation.
  - Capture semantically relevant details; ignore the rest.
- The importance of semantics.
  - We prove things about analyses with respect to the semantics of the underlying language.
- Program proofs as inductive invariants.
- Implementation
  - You do not understand analysis until you have written several.

# Course mechanics

# When/what

- Lectures 2x week (Tue, Th in GHC 4102).
  - Active learning exercise(s) in every class – bring pen & paper!!!
  - Lecture notes for review — get latest PDF from website
- Recitation 1x week (Fridays in WEH 5320).
  - Lab-like, very helpful for homework.
  - Be ready to work
- Homework, midterm exam, project.
- There is an optional physical textbook. (“PPA”)

# Communication

- Course website: <https://cmu-program-analysis.github.io>
- We also use Piazza, Gradescope (see website for links)
  - Gradescope: For homework assignments and exams (grading) <https://www.gradescope.com/courses/1103580> (entry code **J6EK22**)
  - Piazza: Please use public posts for any course related questions as much as possible, unless the matter is sensitive. Feel free to respond to other posts and engage in discussion.
- We have office hours! Or, by appointment.

# “How do I get an A?”

- 10% in-class participation and exercises
- 40% homework assignments
  - Both written (proof-y) and coding (implementation-y).
  - First one (mostly coding) to be released by Friday!
- 30% across two midterm exams
- 20% final project
  - There will be some options here.
- No final exam; exam slot used for project presentations.
- We have late days and an absence policy; read the syllabus.
  - *tl;dr* — 6 free excuses across the whole semester, for any reason

# Slight variations in expectations

- If you're taking the undergraduate version of the course (17-355)
  - Recitation attendance is expected and part of participation grade.
- If you're taking the graduate version of the course (17-665/819)
  - Recitation attendance is encouraged.
  - Higher bar for final course project.
    - Master's students: Expected to engage with large codebases (either frameworks or targets)
    - PhD students: Expected to engage with research questions
- You are welcome to move up your expectations to be assessed differently (email me)

# CMU can be a pretty intense place.

- A 12-credit course is expected to take ~12 hours a week.
- We aim to provide a rigorous but tractable course.
  - More frequent assignments rather than big monoliths
  - Midterm exam to cover core material from first half of course
- Please let us know how much time the class is *actually taking*.
  - We have no way of knowing if you have three midterms in one week.
  - Sometimes, we misjudge assignment difficulty.
- If it's 2 am and you're panicking...put the homework down, send us an email, and go to bed.