

Lecture 1: Introduction to Program Analysis

17-355/17-665/17-819: Program Analysis
Claire Le Goues and Fraser Brown
Jan 17, 2023

* Course materials developed with Jonathan Aldrich and Rohan Padyhe



S3D

Software and Societal
Systems Department

Introductions



Prof. Claire Le Goues



Prof. Fraser Brown



TA Daniel Ramos

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 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 and declarative languages 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!

```
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
[Please fix](#) [Not useful](#)
Suggested fix attached: [show](#)

```
}

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

ErrorProne

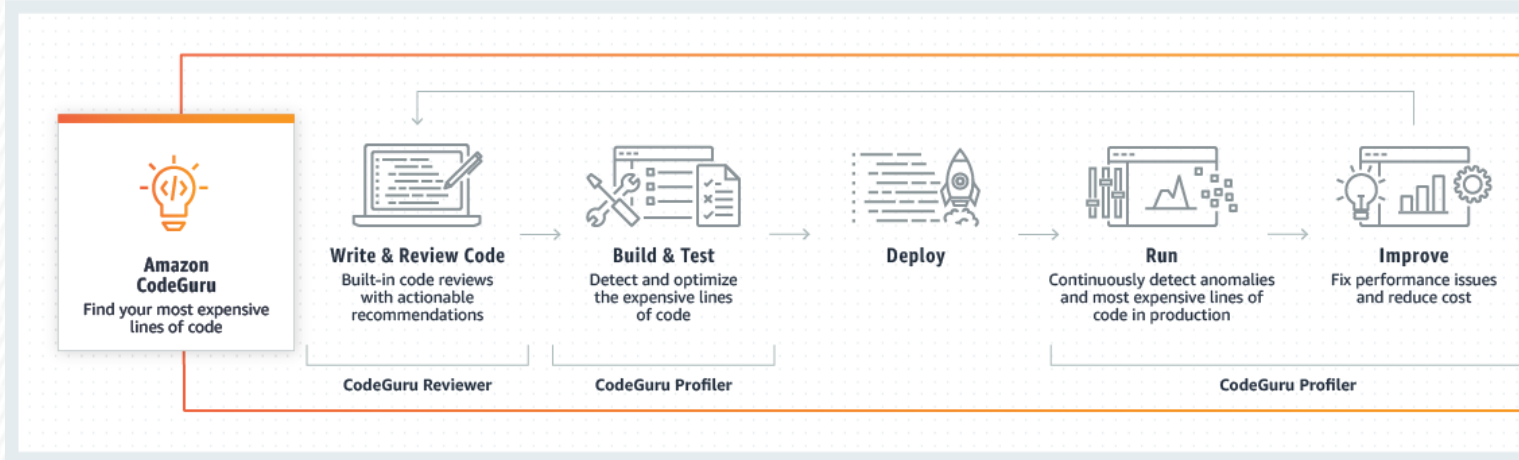
The screenshot shows the GitHub Marketplace interface. At the top, there's a search bar and navigation links for Pull requests, Issues, Marketplace, and Explore. The main content area is titled 'Marketplace / Search results' and shows a search for 'Code quality' with 245 results. A list of tools is displayed, including CodeScene, TestQuality, CodeFactor, Restyled.io, DeepScan, LGTM, Datree, Lucidchart Connector, DeepSource, Code Inspector, Codacov, codebeat, Codacy, Better Code Hub, Code Climate, Coveralls, Sider, Imgbot, and codelingo. Each tool entry includes a logo, name, and a brief description. The 'Code quality' category is selected, and the results are filtered accordingly. The page also includes a 'Filters' section on the left and a 'Also recommended for you' section at the bottom.



Advanced examples from industry

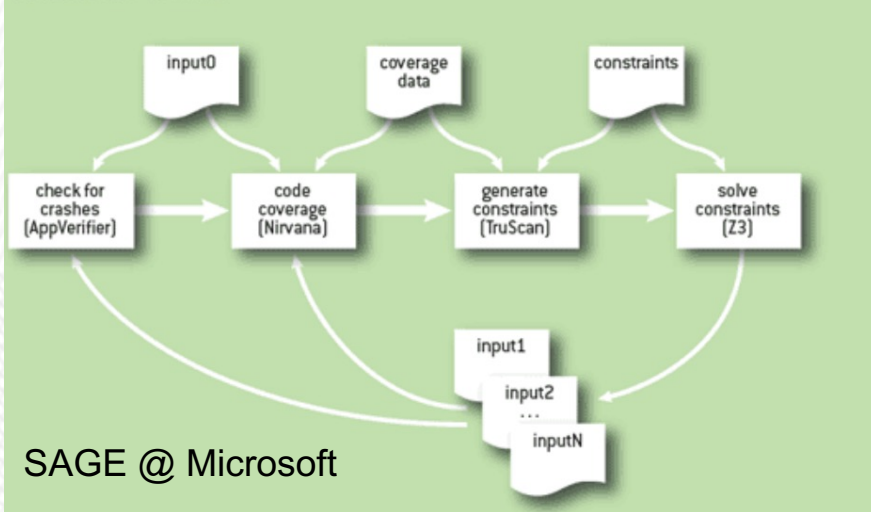
CodeGuru @ Amazon

GitHub CoPilot



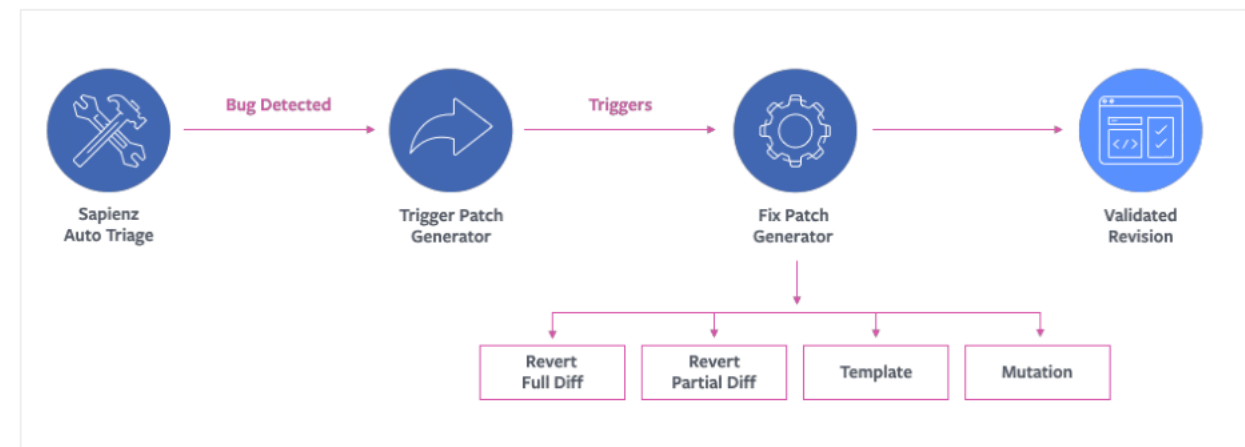
```
1 #!/usr/bin/env ts-node
2
3 import { fetch } from "fetch-h2";
4
5 // Determine whether the sentiment of text is positive
6 // Use a web service
7 async function isPositive(text: string): Promise<boolean> {
8   const response = await fetch("http://text-processing.com/api/sentiment/", {
9     method: "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



SAGE @ Microsoft

Sapienz and SapFix @ Facebook



Common types of issues found using automated program analysis

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

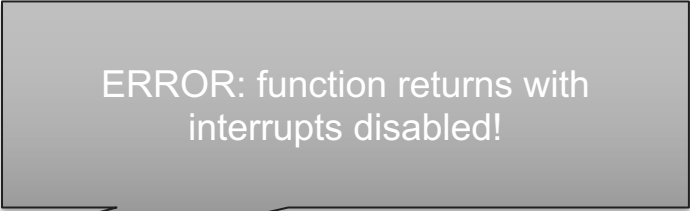
**IS THERE A BUG IN
THIS CODE?**


```
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 '00

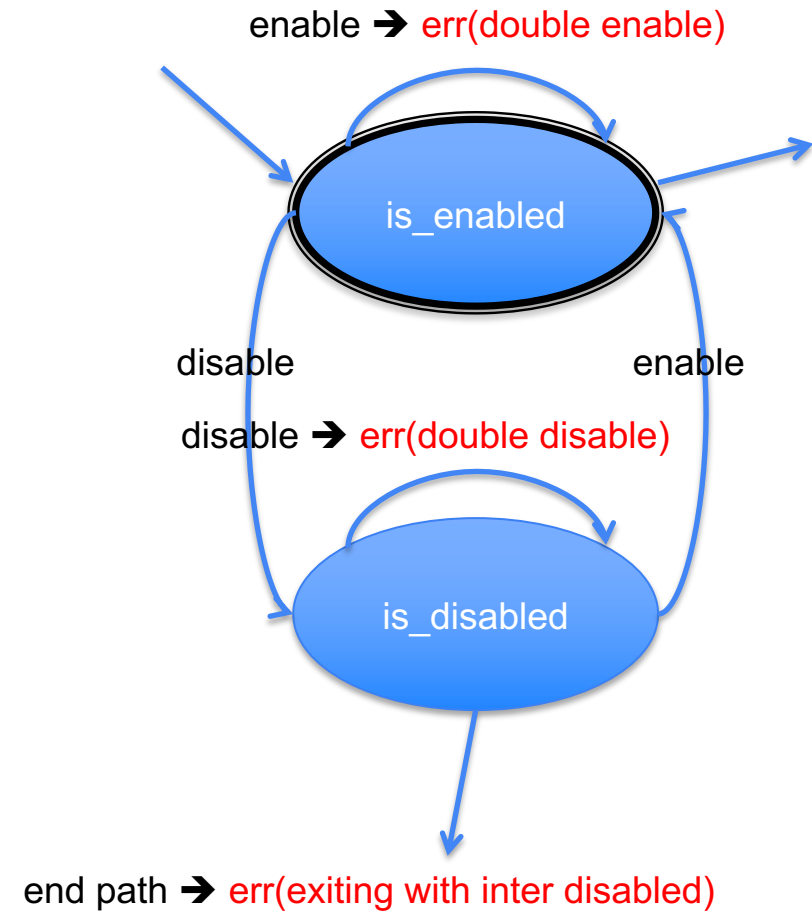
```
1. /* from Linux 2.3.99 drivers/block/raid5.c */
2. static struct buffer_head *
3. get_free_buffer(struct stripe_head * sh,
4.                 int b_size) {
5.     struct buffer_head *bh;
6.     unsigned long flags;
7.     save_flags(flags);
8.     cli(); // disables interrupts
9.     if ((bh = sh->buffer_pool) == 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 '00

```
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 '00

```
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 '00


```
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 '00

```
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 '00

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 the program without executing it.
 - ...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)

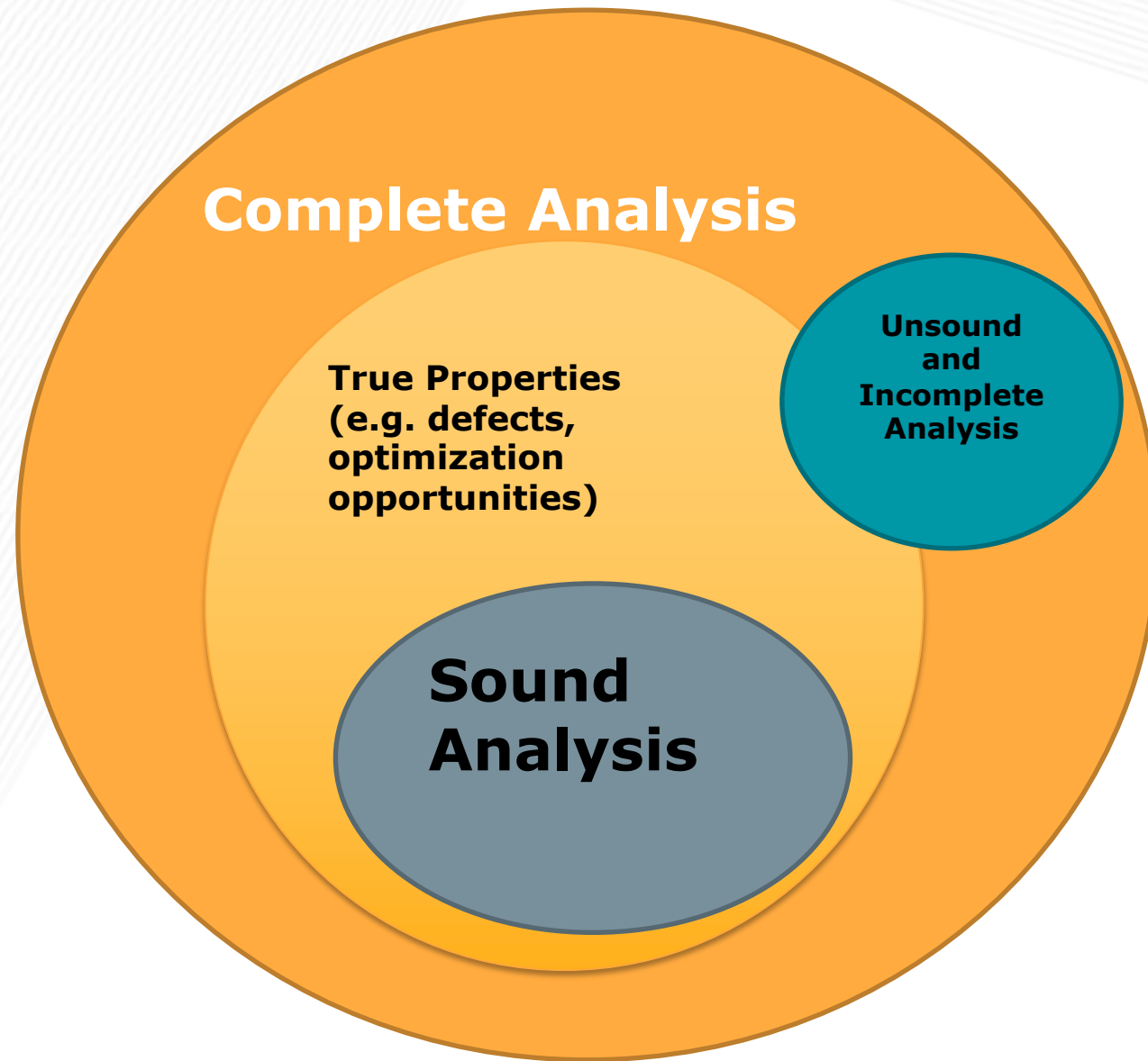
Over-approximate analysis:
 reports all potential defects
 -> no false negatives
 -> subject to false positives

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



In Defense of Soundness: A Manifesto

Ben Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Sam Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis
Microsoft Research, Samsung Research America, University of Athens, University of Waterloo, University of Alberta, University of Colorado Boulder, Tufts University, IIT Bombay, Aarhus University, Google

Static program analysis is a key component of many software development tools, including compilers, development environments, and verification tools. Practical applications of static analysis have grown in recent years to include tools by companies such as Coverity, Fortify, GrammaTech, IBM, and others. Analyses are often expected to be *sound* in that their result models all possible executions of the program under analysis. Soundness implies that the analysis computes an over-approximation in order to stay tractable; the analysis result will also model behaviors that do not actually occur in any program execution. The *precision* of an analysis is the degree to which it avoids such spurious results. Users expect analyses to be sound as a matter of course, and desire analyses to be as precise as possible, while being able to scale to large programs.

Soundness would seem essential for any kind of static program analysis. Soundness is also widely emphasized in the academic literature. Yet, in practice, soundness is commonly eschewed: we are not aware of a *single* realistic whole-program¹ analysis tool (e.g., tools widely used for bug detection, refactoring assistance, programming automation, etc.) that does not purposely make unsound choices. Similarly, virtually all published whole-program analyses are unsound and omit conservative handling of common language features when applied to *real programming languages*.

The typical reasons for such choices are engineering compromises: implementers of such tools are well aware of how they could handle complex language features soundly (e.g., by assuming that a complex language feature can exhibit *any* behavior), but do not do so because this would make the analysis *unscalable* or *imprecise* to the point of being useless. Therefore, the dominant practice is one of treating soundness as an engineering choice.

In all, we are faced with a paradox: on the one hand we have the ubiquity of unsoundness in any practical whole-program analysis tool that has a claim to precision and scalability; on the other, we have a research community that, outside a small group of experts, is oblivious to any unsoundness, let alone its preponderance in practice.

Our observation is that the paradox can be reconciled. The state of the art in realistic analyses exhibits consistent traits, while also integrating a sharp discontinuity. On the one hand, typical realistic analysis implementations have a *sound core*: most common language features are *over-approximated*, modeling all their possible behaviors. Every time there are multiple options (e.g., branches of a conditional statement, multiple data flows) the analysis models all of them. On the other hand, some specific language features, well known to experts in the area, are best *under-approximated*. Effectively, every analysis pretends that perfectly possible behaviors cannot happen. For instance, it is conventional for an otherwise sound static analysis to treat highly-dynamic language constructs, such as Java reflection or *eval* in JavaScript, under-approximately. A practical analysis, therefore, may pretend that *eval* does nothing, unless it can precisely resolve its string argument at compile time.

We introduce the term *soundy* for such analyses. The concept of *soundiness* attempts to capture the balance, prevalent in practice, of over-approximated handling of most language features, yet deliberately under-approximated handling of a feature subset well recognized by experts. Soundiness is in fact what is meant in many papers that claim to describe a sound analysis. A *soundy* analysis aims to be as sound as possible without excessively compromising precision and/or scalability.

Our message here is threefold:

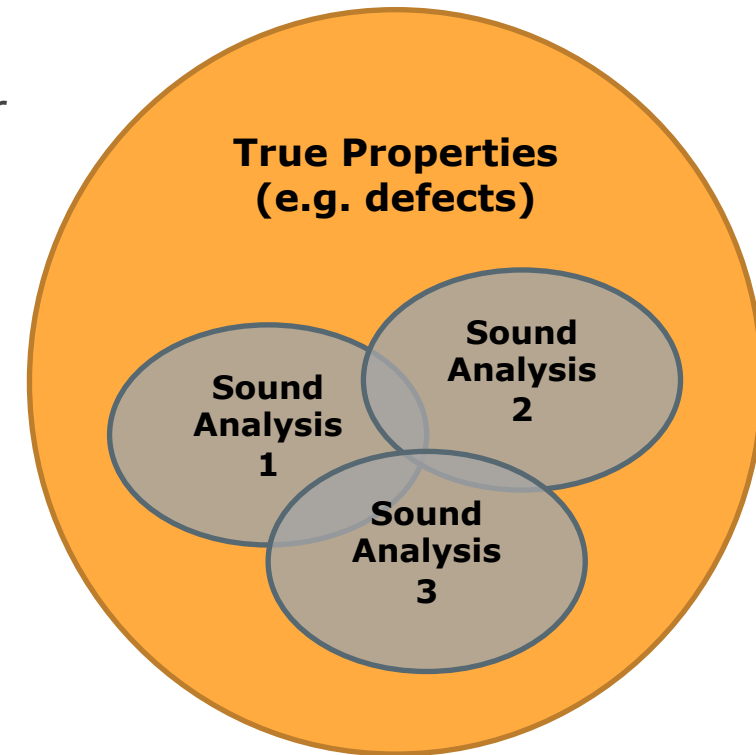
1. We bring forward the ubiquity of, and engineering need for, unsoundness in the static program analysis practice. For static analysis researchers, this may come as no surprise. For the rest of the community, which expects to use analyses as a black box, this unsoundness is less understood.

¹ We draw a distinction between whole program analyses, which need to model shared data, such as the heap, and modular analyses—e.g., type systems. Although this space is a continuum, the distinction is typically well-understood.



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 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!
- SAT/SMT solvers
- Symbolic execution: test all possible executions paths simultaneously.
 - Test generation
- Grey-box analysis for fuzz testing
- Program synthesis
- Program repair
- Real-world verification
- We will basically *not* cover types.

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 (T,Th, 11 AM ET – in WEH 2302).
 - Active learning exercise(s) in every class, so a laptop or phone is useful (a comment on that...)
 - Lecture notes for review --- get latest PDF from website
 - Some slides, much board, sometimes all board but slides released post facto with exercise answers. Check the website.
- Recitation 1x week (Fr, 10 AM ET – in BH A51).
 - Lab-like, very helpful for homework.
 - Be ready to work, bring a laptop
- Lecture recordings?
- Homework, two exams, project.
- There is an optional physical textbook. (“PPA”)
- We have lots of lecture notes that we release on the website.

Communication

- Course website: <https://cmu-program-analysis.github.io>
- We also use Canvas, Piazza, Gradescope (see website for links)
 - Canvas: In-class exercises, some assignments, grades tally
 - Gradescope: For written assignments
 - 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.
- Please use Piazza private messages for logistics or extensions questions; if you *must* email, ***email us both.***

“How do I get an A?”

- 10% in-class (lecture/recitation) participation and exercises
 - Check the syllabus for what do if you have to miss class.
- 40% homework assignments
 - Both written (proof-y) and coding (implementation-y).
 - First one (mostly coding) to be released by Friday!
- 30% two exams
- 20% final project
 - There will be some options here.
- No final exam; exam slot used for project presentations.
- We have late days and a late day policy; read the syllabus.
 - *tl;dr*: 3 late days per HW, with 5 total late days before penalties kick in

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 us)

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
 - Two exams to cover/integrate core material, but lower stakes per exam.
- 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.

Let's get started

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.

Our first representation: Abstract Syntax

- A tree representation of source code based on the language grammar.
- **Concrete syntax:** The rules by which programs can be expressed as strings of characters
 - E.g. "if (x * (a + b)) { foo(a); }"
 - Use finite automata and context-free grammars, automatic lexer/parser generators
- **Abstract syntax:** a subset of the parse tree of the program.
 - Only care about statements, expressions and their relationship with constituent operands.
 - Don't care about parenthesis, semicolons, keywords, etc.
- (The intuition is fine for this course; take compilers if you want to learn how to parse for real.)

The WHILE language – Example program

```
y := x;  
z := 1;  
if y > 0 then  
  while y > 1 do  
    z := z * y;  
    y := y - 1  
else  
  skip
```

- Sample program computes $z = x!$ using y as a temp variable.
- WHILE uses assignment statements, if-then-else, while loops.
- All vars are integers.
- Expressions only arithmetic (for vars) or relational (for conditions).
- No I/O statements. Inputs and outputs are implicit.
 - Later on, we may add extensions with explicit `read x` and `print x`.

WHILE abstract syntax

- Categories:
 - $S \in \mathbf{Stmt}$ statements
 - $a \in \mathbf{Aexp}$ arithmetic expressions
 - $x, y \in \mathbf{Var}$ variables
 - $n \in \mathbf{Num}$ number literals
 - $P \in \mathbf{BExp}$ boolean predicates
 - $l \in \mathbf{labels}$ statement addresses (line numbers)
- Syntax:
 - $S ::= x := a \mid \text{skip} \mid S_1 ; S_2$
 $\mid \text{if } P \text{ then } S_1 \text{ else } S_2 \mid \text{while } P \text{ do } S$
 - $a ::= x \mid n \mid a_1 \text{ op}_a a_2$
 - $\text{op}_a ::= + \mid - \mid * \mid / \mid \dots$
 - $P ::= \text{true} \mid \text{false} \mid \text{not } P \mid P_1 \text{ op}_b P_2 \mid a_1 \text{ op}_r a_2$
 - $\text{op}_b ::= \text{and} \mid \text{or} \mid \dots$
 - $\text{op}_r ::= < \mid \leq \mid = \mid > \mid \geq \mid \dots$

Concrete syntax is similar, but adds things like (parentheses) for disambiguation during parsing

Together: Building an AST

```
y := x;  
z := 1;  
if y > 0 then  
    while y > 1 do  
        z := z * y;  
        y := y - 1  
    else  
        skip
```


Ex 1: Building an AST for C code

```
void copy_bytes(char dest[], char source[], int n) {  
    for (int i = 0; i < n; ++i)  
        dest[i] = source[i];  
}
```

Our first static analysis: AST walking

- One way to find “bugs” is to walk the AST, looking for particular patterns.
 - Traverse the AST, look for nodes of a particular type
 - Check the neighborhood of the node for the pattern in question.

Example: shifting by more than 31 bits.

- Assume we want to find code patterns of the following form:
 $x \ll -3$
 $z \gg 35$
- For 32-bit integer vars, these operations may signal unintended typos, since it doesn't makes sense to shift by a number outside the range (0, 32).

Example: shifting by more than 31 bits.

```
For each instruction I in the program
  if I is a shift instruction
    if (type of I's left operand is int
        && I's right operand is a constant
        && value of constant < 0 or > 31)
      warn("Shifting by less than 0 or more
           than 31 is meaningless")
```


Our first static analysis: AST walking

- One way to find “bugs” is to walk the AST, looking for particular patterns.
 - Traverse the AST, look for nodes of a particular type
 - Check the neighborhood of the node for the pattern in question.
- Various frameworks, some more language-specific than others.
 - Tradeoffs between language agnosticism and semantic information available.
 - Consider “grep”: very language agnostic, not very smart.
 - Python’s “astor” package designed for Python ASTs. Clean API; highly specific.
- Classic architecture based on Visitor pattern:
 - class Visitor has a visitX method for each type of AST node X
 - Default Visitor code just descends the AST, visiting each node
 - To do something interesting for AST element of type X, override visitX
- **More recent approaches based on semantic search, declarative logic programming, or query languages.**