Lecture 1: Introduction to Program Analysis

17-355/17-665/17-819: Program Analysis
Rohan Padhye
Jan 18, 2022

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

Prof. Rohan Padhye

TA Bella Laybourn
My Background

• Involved with program analysis for ~10 years.
• PhD from UC Berkeley, Masters from IIT Bombay (India)
• Worked at IBM Research, Microsoft Research, and Samsung Research America
• Advising PhD students at CMU's Institute for Software Research
• Developed tools for improving developer productivity, finding input-validation software bugs, identifying security vulnerabilities in mobile systems, discovering concurrency issues in distributed systems, etc.
• Contributed to research on fuzz testing, static interprocedural analysis, dynamic performance analysis, etc.
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 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:
  o Precise program representations
  o Tractable, systematic ways to reason over those representations.

• We will learn:
  o How to unambiguously define the meaning of a program, and a programming language.
  o How to prove theorems about the behavior of particular programs.
  o 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

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

    if (a == "5")
        doMoreStuff();
}
```
You’ve seen it before

```java
public int foo() {
  doStuff();
  return 3;
  doMoreStuff();
  return 4;
}
```
Lots of tools available

Lint

```java
package com.google.devtools.statisticanalysis;

public class Test {
    public boolean foo() {
        return getString().equals("foo");
    }
}
```

ErrorProne

```java
package com.google.devtools.statisticanalysis;

public class Test {
    public boolean foo() {
        return getString().equals("foo");
    }
}
```

https://github.com/marketplace?category=code-quality
Advanced examples from industry

**CodeGuru @ Amazon**

- **Write & Review Code**: Built-in code reviews with actionable recommendations
- **Build & Test**: Detect and optimize the expensive lines of code
- **Deploy**: Continuously detect anomalies and most expensive lines of code in production
- **Improve**: Fix performance issues and reduce cost

**Architecture of SAGE**

- Input
- Code coverage
- Generate constraints
- Solve constraints

**SAGE @ Microsoft**

**Sapienz and SapFix @ Facebook**

- Bug Detected
- Triggers
- Fix Patch Generator
- Validated Revision

GitHub CoPilot
Common types of issues found using automated program analysis

• Defects that result from inconsistently following simple design rules.
  o **Security:** Buffer overruns, improperly validated input.
  o **Memory safety:** Null dereference, uninitialized data.
  o **Resource leaks:** Memory, OS resources.
  o **API Protocols:** Device drivers; real time libraries; GUI frameworks.
  o **Exceptions:** Arithmetic/library/user-defined
  o **Encapsulation:** Accessing internal data, calling private functions.
  o **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?
/* 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;

}
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.}

Example from Engler et al., Checking system rules Using System-Specific, Programmer-Written Compiler Extensions, OSDI '000
(c) 2022 J. Aldrich, C. Le Goues, R. Padhye
Abstract Model

- `is_enabled`
  - Enable: `err(double enable)`
  - Disable: `err(double disable)`

- `is_disabled`
  - End path: `err(exiting with inter disabled)`
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;
}
/* 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;
}
/* 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;
}
/* 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;
}
/* 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;
}
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:
  o Precise program representations
  o Tractable, systematic ways to reason over those representations.

• We will learn:
  o How to unambiguously define the meaning of a program, and a programming language.
  o How to prove theorems about the behavior of particular programs.
  o 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:
  o **Dynamic:**
    ▪ **Testing:** Direct execution of code on test data in a controlled environment.
    ▪ **Analysis:** Tools extracting data from test runs.
  o **Static:**
    ▪ **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
    ▪ **Analysis:** Tools reasoning about the program without executing it.
  o ...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 \texttt{divides\_by\_zero}):

1. \texttt{int silly(program p, input i) \{}
2. \texttt{p(i);}  
3. \texttt{return 5/0;}  
4. \texttt{\}}
5. \texttt{bool halts(program p, input i) \{}
6. \texttt{return divides\_by\_zero(`silly(p,i)`);}  
7. \texttt{\}}
<table>
<thead>
<tr>
<th></th>
<th>Error exists</th>
<th>No error exists</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error Reported</strong></td>
<td>True positive</td>
<td>False positive</td>
</tr>
<tr>
<td></td>
<td>(correct analysis result)</td>
<td></td>
</tr>
<tr>
<td><strong>No Error Reported</strong></td>
<td>False negative</td>
<td>True negative</td>
</tr>
<tr>
<td></td>
<td>(correct analysis result)</td>
<td></td>
</tr>
</tbody>
</table>

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.
Complete Analysis

True Properties (e.g. defects, optimization opportunities)

Sound Analysis

Unsound and Incomplete Analysis
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.
  - Concolic execution
  - Test generation
- Grey-box analysis for fuzz testing
- Dynamic analysis for race detection
- Program synthesis
- Program repair
- We will basically not cover types.
Fundamental concepts

• Abstraction
  o Elide details of a specific implementation.
  o Capture semantically relevant details; ignore the rest.

• The importance of semantics.
  o We prove things about analyses with respect to the semantics of the underlying language.

• Program proofs as inductive invariants.

• Implementation
  o You do not understand analysis until you have written several.
Course mechanics
What to expect

• Beautiful and elegant theory (15-251 is a soft pre-req)
  o Mostly discrete mathematics, symbolic reasoning, inductive proofs
  o This is traditionally a “white-board” course [using slides while we're on Zoom]

• Build awesome tools
  o 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)
  o Representations, control/data-flow, input state space

• Appreciate the limits and achievements in the space
  o What tools are impossible to build?
  o What tools are impressive that they exist at all?
  o When is it appropriate to use a particular analysis tool versus another?
  o How to interpret the results of a program analysis tool?
When/what

• Lectures 2x week (T,Th – in GHC 4101 from Feb 1; now on Zoom).
  o Active learning exercise(s) in every class
  o Lecture notes for review --- get latest PDF from website
• Recitation 1x week (Fr – in MI 348 from Feb 1; now on Zoom).
  o Lab-like, very helpful for homework.
  o 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 Canvas, Piazza, Gradescope (see website for links)
  o Canvas: In-class exercises, some assignments, Zoom links, grades tally
  o Gradescope: For written assignments
  o 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
- 50% homework assignments
  - Both written (proof-y) and coding (implementation-y).
  - First one (mostly coding) to be released by Friday!
- 20% midterm exam
- 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)
  o Recitation attendance is expected and part of participation grade.

• If you’re taking the graduate version of the course (17-665/819)
  o Recitation attendance is encouraged.
  o 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.
  o More frequent assignments rather than big monoliths
  o Midterm exam to cover core material from first half of course
• Please let us know how much time the class is actually taking.
  o We have no way of knowing if you have three midterms in one week.
  o 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 use extensions with explicit `read x` and `print x`. 

(c) 2022 J. Aldrich, C. Le Goues, R. Padhye
**WHILE abstract syntax**

- **Categories:**
  - \( S \in \text{Stmt} \) — statements
  - \( a \in \text{Aexp} \) — arithmetic expressions
  - \( x, y \in \text{Var} \) — variables
  - \( n \in \text{Num} \) — number literals
  - \( P \in \text{BExp} \) — boolean predicates
  - \( l \in \text{labels} \) — statement addresses (line numbers)

- **Syntax:**
  - \( S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \)
    - \( \text{if} \ P \text{ then } S_1 \text{ else } S_2 \mid \text{while} \ P \text{ do } S \)
  - \( a ::= x \mid n \mid a_1 \text{ op } a_2 \)
  - \( \text{op}_a ::= + \mid - \mid * \mid / \mid \ldots \)
  - \( P ::= \text{true} \mid \text{false} \mid \text{not} \ P \mid P_1 \text{ op }_b P_2 \mid a1 \text{ op }_r a2 \)
  - \( \text{op}_b ::= \text{and} \mid \text{or} \mid \ldots \)
  - \( \text{op}_r ::= < \mid \leq \mid = \mid > \mid \geq \mid \ldots \)

Concrete syntax is similar, but adds things like (parentheses) for disambiguation during parsing.
Exercise: Building an AST

\[ \begin{align*}
y & := x; \\
z & := 1; \\
\text{if } y > 0 \text{ then} & \\
& \quad \text{while } y > 1 \text{ do} \\
& \quad \quad z := z \times y; \\
& \quad \quad y := y - 1 \\
\text{else} & \\
& \quad \text{skip}
\end{align*} \]
Ex 1: Building an AST for C code

```c
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.
  o Traverse the AST, look for nodes of a particular type
  o Check the neighborhood of the node for the pattern in question.
  o Basically, a glorified “grep” that knows about the syntax but not semantics of a language.
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 make 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.

- One common 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

- Other more recent approaches based on semantic search, declarative logic programming, or query languages.
CodeQL

- A language for querying code. Developed by GitHub.
- Supports many common languages.
- Library of common programming patterns and optimizations.

### Inefficient empty string test

**Name:** inefficient empty string test  
**Description:** Checking a string for equality with an empty string is inefficient.  
**ID:** java/inefficient-empty-string-test  
**Kind:** problem  
**Severity:** recommendation  
**Precision:** high

**Query: InefficientEmptyStringTest.ql**

When checking whether a string $s$ is empty, perhaps the most obvious solution is to write something like $s.equals("\")$ (or "".equals(s)). However, this actually carries a fairly significant overhead, because String.equals performs a number of type tests and conversions before starting to compare the content of the strings.

**Recommendation**

The preferred way of checking whether a string $s$ is empty is to check if its length is equal to zero. Thus, the condition is $s.length() == 0$. The `length` method is implemented as a simple field access, and so should be noticeably faster than calling `equals`.

Note that in Java 8 and later, the `String` class has an `isEmpty` method that checks whether a string is empty. If the codebase does not need to support Java 5, it may be better to use that method instead.

Example: Java string compare with ""

```java
// Inefficient version
class InefficientDBClient {
    public void connect(String user, String pw) {
        if (user.equals("") || ".equals(pw))
            throw new RuntimeException();
...
    }
}

// More efficient version
class EfficientDBClient {
    public void connect(String user, String pw) {
        if (user.length() == 0 || (pw != null && pw.length() == 0))
            throw new RuntimeException();
...
    }
}
```
CodeQL query for empty string comparison

```sql
/**
 * @name Inefficient empty string test
 * @description Checking a string for equality with an empty string is Inefficient.
 * @kind problem
 * @problem.severity recommendation
 * @precision high
 * @id java/inefficient-empty-string-test
 * @tags efficiency
 * maintainability
 */

import java

from MethodAccess mc
where
  mc.getQualifier().getType() instanceof TypeString and
  mc.getMethod().hasName("equals") and
  (mc.getArgument(0).StringValue.getRepresentedString() = "" or
   mc.getQualifier().StringValue.getRepresentedString() = "")
select mc, "Inefficient comparison to empty string, check for zero length instead."
```
Ex 2: String concatenation in a loop

• Write pseudocode for a simple syntactic analysis that warns when string concatenation occurs in a loop
  o Why? In Java and .NET it may be more efficient to use a StringBuffer
  o Assume any appropriate AST elements
For next time

• Get on Piazza and Canvas
• Read lecture notes and the course syllabus
• Homework 1 will be released later this week, and is due next Thursday.