

CIS 547  
Software Analysis

# Introduction to Software Analysis

Mayur Naik



```
s.close()
for i in range(1, 1000):
    attack()

import os
print os.getcwd()
print id = os.fork()
def attack():
    #pid = os.fork()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((sys.argv[1], sys.argv[2]))
    print ">> GET /" + sys.argv[1]
    s.send("GET /" + sys.argv[1])
    s.send("Host: " + sys.argv[2])
    s.close()
for i in range(1, 1000):
```

Welcome to Software Analysis!


In this course, we will study the theory and practice of software analysis, which lies at the heart of many software development processes such as diagnosing bugs, testing, debugging, and more.

What this class won't do is teach you basic concepts of programming. Instead, through a mix of basic and advanced exercises and examples, you will learn techniques and tools to enhance your existing programming skills and build better software.

CIS 547 - Software Analysis

# LESSON

## Course Overview and Introduction



Property of Penn Engineering | 2

CIS 547 - Software Analysis

SEGMENT

Why Take this Course?

Penn Engineering

Property of Penn Engineering | 3

CIS 547 - Software Analysis

Why Take This Course?

- Learn modern methods for improving software quality
  - reliability, security, performance, etc.
- Become a better software developer/tester
- Build specialized tools for software diagnosis and testing
- For the war stories

Penn Engineering

Property of Penn Engineering | 4

So why should you take this course?

Bill Gates once said and I quote "We have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization."

In this course, you will learn modern methods for improving software quality in a broad sense, encompassing reliability, security, and performance.

This will enable you to become a better and more productive software developer or tester, as the aspects that we will address in this course, such as software testing and debugging, comprise over 50% of the cost of software development.

You will also be able to implement these methods in specialized tools for software diagnosis and testing tasks. An example task is systematically testing an Android application in various end-user scenarios.

But let's face it: you're really here for the war stories.



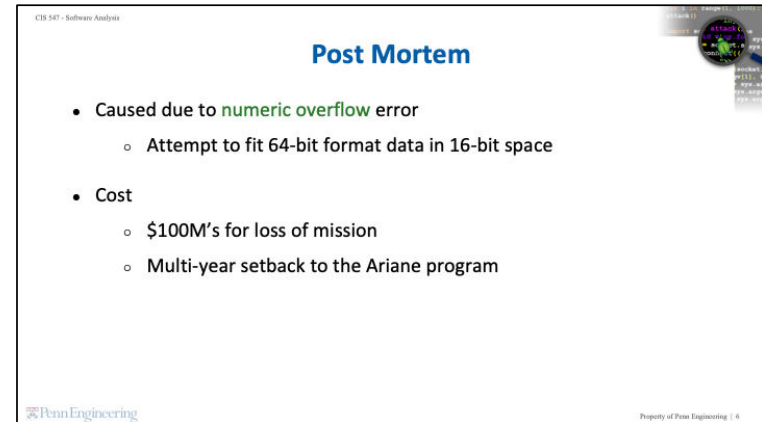
The Ariane Rocket Disaster of 1996 is a war story of epic proportions.

Here is a video of the maiden launch of the Ariane Rocket in 1996 by the European Space Agency.

[https://www.youtube.com/watch?v=PK\\_yguLapgA&t=63](https://www.youtube.com/watch?v=PK_yguLapgA&t=63)

Roughly 40 seconds after the launch, the rocket reaches an altitude of two and a half miles. But then it abruptly changes course and triggers a self-destruct mechanism, destroying its payload of expensive scientific satellites.

So why did this happen, and what was the aftermath of this disaster? Let's take a look.



The cause of the disaster was diagnosed to be a kind of programming error called a numeric overflow error, in a program running on the Ariane rocket's onboard computer.

The error resulted from an attempt during takeoff to convert one piece of data -- the sideways velocity of the rocket -- from a 64-bit format to a 16-bit format. The number was too big to fit and resulted in an overflow. This error was misinterpreted by the rocket's onboard computer as a signal to change the course of the rocket.

This failure translated into millions of dollars in lost assets and several years of setbacks for the Ariane Program. The methods that we will learn in this course could have prevented this error.

To read more about this disaster access the link provided in the lecture handout. <http://www.around.com/ariane.html>

Now let's look at another problem that is more earthly and affects everyday users of software.

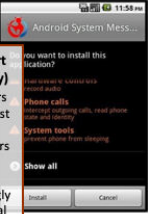
CSB 547 - Software Analysis

## Security Vulnerabilities

- Exploits of errors in programs
- Widespread problem
  - Moonlight Maze (1998)
  - Code Red (2001)
  - Titan Rain (2003)
  - Stuxnet (2010)
  - Heartbleed (2014)
- And getting worse ...

**2011 Mobile Threat Report (Lookout™ Mobile Security)**

- 0.5-1 million Android users affected by malware in first half of 2011
- 3 out of 10 Android owners likely to face web-based threat each year
- Attackers using increasingly sophisticated ways to steal data and money



Property of Penn Engineering | 7

While the Ariane disaster was a consequence of a programming error, at least the damage was an unintended consequence.

On the other hand, we have security vulnerabilities, wherein malicious hackers can exploit these errors in everyday mobile and web applications to compromise the security of the underlying systems and data.

This is a widespread problem, and it has been since the early days of the Internet. Several examples of programming bugs leading to security vulnerabilities you may have heard of include:

- Moonlight Maze, which probed American computer systems for at least two years since 1998,
- Code Red, which affected hundreds of thousands of Microsoft web servers in 2001,
- Titan Rain, a series of coordinated attacks on American computer systems for three years since 2003,
- Stuxnet, a computer worm that shut down Iranian nuclear facilities in 2010, and
- Heartbleed, a security bug that allowed to steal information protected using the popular OpenSSL cryptographic software library.

And the problem has only gotten worse with the advent of smartphones; now you too can make yourself vulnerable to programming disasters simply by installing an app.

CSB 547 - Software Analysis

## READING

### From Software Bugs to Security Vulnerabilities

Property of Penn Engineering | 8

## LESSON

### Introduction to Software Analysis

## SEGMENT

### What is Program Analysis? Overview of Dynamic, Static, and Hybrid Approaches

CS 547 - Software Analysis

## What is Program Analysis?

- Body of work to discover useful facts about programs
- Broadly classified into three kinds:
  - **Dynamic (execution-time)**
  - **Static (compile-time)**
  - **Hybrid (combines dynamic and static)**

Penn Engineering

Property of Penn Engineering | 11

Program analysis is the process of automatically discovering useful facts about programs. An example of a useful fact is a programming error. We saw an example of a programming error that was responsible for the Ariane disaster, and others that underlie security vulnerabilities.

Program analysis as a whole can be broadly classified into three kinds of analyses: dynamic, static, and hybrid.

Dynamic analysis is the class of run-time analyses. These analyses discover information by running the program and observing its behavior.

Static analysis is the class of compile-time analyses. These analyses discover information by inspecting the source code or binary code of the program.

Hybrid analyses combine aspects of both dynamic and static analyses, by combining runtime and compile-time information in interesting ways.

Let's take a closer look at dynamic and static analyses.

CS 547 - Software Analysis

## Dynamic Program Analysis

- Infer facts of the program by monitoring its runs
- Examples:
 

<p>Array bound checking Purify</p>	<p>Memory leak detection Valgrind</p>
<p>Datarace detection Eraser</p>	<p>Finding likely invariants Daikon</p>

Penn Engineering

Property of Penn Engineering | 12

Dynamic program analysis infers facts about a program by monitoring its runs. Here are four examples of well-known dynamic analysis tools.

Purify is a dynamic analysis tool for checking memory accesses, such as array bounds, in C and C++ programs.

Valgrind is a dynamic analysis tool for detecting memory leaks in x86 binary programs. A memory leak occurs when a program fails to release memory that it no longer needs.

Eraser is a dynamic analysis tool for detecting data races in concurrent programs. A data race is a condition in which two threads in a concurrent program attempt to simultaneously access the same memory location, and at least one of those accesses is a write. Data races typically indicate programming errors, as the order in which the accesses in a data race occur can produce different results from run to run.

Finally, Daikon is a dynamic analysis tool for finding likely invariants. An invariant is a program fact that is true in every run of the program.

CIS 547 - Software Analysis

## Static Analysis

- Infer facts of the program by inspecting its source (or binary) code
- Examples:
 

Suspicious error patterns Lint, FindBugs, Coverity	Checking API usage rules Microsoft SLAM
Memory leak detection Facebook Infer	Verifying invariants ESC/Java

Penn Engineering Property of Penn Engineering | 13

Static program analysis infers facts about a program by inspecting its code. Here are four examples of well-known static analysis tools.

Tools such as Lint, FindBugs, and Coverity inspect the source code of C++ or Java programs for suspicious error patterns.

SLAM is a tool from Microsoft that checks whether C programs respect API usage rules. This tool is used by Windows developers to check whether device drivers use the API of the Windows kernel correctly.

Facebook Infer is a static analysis tool developed by Facebook for detecting memory leaks in Android applications.

Finally, ESC/Java is a tool for specifying and verifying invariants in Java programs.

We will look at an example of an invariant next.

CIS 547 - Software Analysis

## SEGMENT

### Discovering Invariants by Dynamic and Static Analysis

Penn Engineering Property of Penn Engineering | 14

CS 547 - Software Analysis

## Program Invariants

An invariant at the end of the program is  $(z == c)$ , for some constant  $c$ . What is  $c$ ?

```

int p(int x) { return x * x; }
void main() {
  int z;
  if (getc() == 'a')
    z = p(6) + 6;
  else
    z = p(-7) - 7;
}

```

z = ?

Penn Engineering Property of Penn Engineering | 15

Let's do the following exercise to illustrate a concrete example of a useful program fact, namely, a program invariant.

Consider the following program which reads a character from the input using function `getc()`. If the input is the character 'a', it takes the true branch, otherwise it takes the false branch.

Recall that an invariant is a program fact that is true in every run of the program.

An invariant at the end of this example program is  $(z == c)$  for some constant  $c$ . What is  $c$ ? Let's figure it out.

CS 547 - Software Analysis

## Program Invariants

An invariant at the end of the program is  $(z == c)$ , for some constant  $c$ . What is  $c$ ?

Disaster averted!

```

int p(int x) { return x * x; }
void main() {
  int z;
  if (getc() == 'a')
    z = p(6) + 6;
  else
    z = p(-7) - 7;
  if (z != 42)
    disaster();
}

```

z = 42

Penn Engineering Property of Penn Engineering | 16

The value of  $c$  is 42. To see why, we need to reason about only two cases over all runs of this program.

In the runs where the true branch is taken, the value of  $z$  is  $p(6) + 6$ , which is  $6*6 + 6$ , which is  $36 + 6$ , which is 42.

In the runs where the false branch is taken, the value of  $z$  is  $p(-7) - 7$ , which is  $(-7)*(-7) - 7$ , which is  $49 - 7$ , which is 42 again.

Thus, the value of  $c$  is 42. We have thus shown that  $(z == 42)$  is a program invariant at the exit of this program.

Now let us slightly change this program to call `disaster` whenever the value of  $z$  is not equal to 42.

Then, notice that the invariant we just discovered is a useful fact for proving that this program can never call `disaster`!



CS 547 - Software Analysis

## Discovering Invariants By Dynamic Analysis

$(z == 42)$  *might be an invariant*

$(z == 30)$  *is definitely not an invariant*

```

int p(int x) { return x * x; }

void main() {
  int z;
  if (getc() == 'a')
    z = p(6) + 6;
  else
    z = p(-7) - 7;
  if (z != 42)
    disaster();
}

```

**z = 42**

Penn Engineering Property of Penn Engineering | 17

Now let's see how the different kinds of program analyses fare at discovering program invariants.

Let's first consider dynamic analysis. For simplicity, the shown program has only two paths. But in general, programs have loops or recursion, which can lead to arbitrarily many paths. Since dynamic analysis discovers information by running the program a finite number of times, it cannot in general discover information that requires observing an unbounded number of paths. As a result, a dynamic analysis tool like Daikon can at best detect *\*likely\** invariants.

From any run of the shown program, Daikon can at best conclude that  $(z == 42)$  is a *\*likely\** invariant. It cannot prove that  $z$  will always be 42, and that the call to `disaster` can never happen.

This is not to say that dynamic analysis is useless. For one, the information that  $z$  might be 42 could be a useful fact. More importantly, Daikon can conclusively rule out entire classes of invariants even by observing a single run.

For instance, from any run of this example program, Daikon can conclude that  $(z == c)$  is definitely not an invariant for any  $c$  other than 42, such as 30.

On the other hand, to conclusively determine that  $(z == 42)$  is an invariant, and therefore showing that the program will never call `disaster`, we need static analysis.

CS 547 - Software Analysis

## Discovering Invariants By Static Analysis

$(z == 42)$  *is definitely an invariant*

$(z == 30)$  *is definitely not an invariant*

```

int p(int x) { return x * x; }

void main() {
  int z;
  if (getc() == 'a')
    z = p(6) + 6;
  else
    z = p(-7) - 7;
  if (z != 42)
    disaster();
}

```

**z = 42**

Penn Engineering Property of Penn Engineering | 18

Now let's consider how static analysis works on this example.

Static analysis can conclusively say that  $(z == 42)$  is an invariant by inspecting the source code of the program. The reasoning it applies is similar to what we ourselves used in the quiz. Recall that we too inspected the source code to determine that the constant  $c$  has value 42.

Static analysis can therefore show at compile-time that the program will never call `disaster` at run-time.

You should now be able to see how the Ariane disaster could have been averted using static analysis.

Next, we will delve deeper into how static analysis discovers invariants of the form  $(z == 42)$  even for programs that have an unbounded number of paths.

## SEGMENT

## Anatomy of a Static Analysis

## Example Static Analysis Problem

Find variables that have a constant value at a given program point

```
void main() {
  z = 3;
  while (true) {
    if (x == 1)
      y = 7;
    else
      y = z + 4;
    assert(y == 7);
  }
}
```

Consider the problem of proving assertions of the form “variable == constant”, such as “y == 7” in this program.

We can pose this question in terms of a classic static analysis problem.

This problem aims to find variables that have a constant value at a given program point.

We will explain step-by-step how a static analysis discovers that variable y has the value 7 at the end of each iteration of the loop in this program.

CS 547 - Software Analysis

## Terminology

- Control-flow graph
- Abstract vs. concrete states
- Termination
- Completeness
- Soundness

Property of Penn Engineering | 21

Let's first introduce common terminology. Static analysis typically operates on a suitable intermediate representation of the program. One such representation shown here is a control-flow graph. It is a directed graph that summarizes the flow of control in all possible runs of the program. Each node in the graph corresponds to a unique statement in the program, and each edge outgoing from a node denotes a possible successor of that node in some execution.

To achieve its stated goal, our static analysis tracks the values of the three variables in this program,  $x$ ,  $y$ , and  $z$ , at each program point. This is called an abstract state, in contrast to a concrete state which tracks the actual values in a particular run. Since static analysis does not run the program, it does not operate directly over concrete states. Instead, it operates over abstract states, each of which summarizes a set of concrete states. This is called the abstract semantics as opposed to the concrete semantics.

As a result of this summarization, the analysis may fail to accurately represent the value of a variable in an abstract state, which we denote using a question mark. While this ensures the termination of the analysis even for programs with an unbounded number of paths, it can also lead the analysis to miss variables that have a constant value at a given program point. For this reason, we say that the analysis sacrifices completeness. Conversely, whenever the analysis concludes that a variable has a constant value at a given program point, this conclusion is indeed correct in all runs of the program. For this reason, we say that the analysis is sound.

CS 547 - Software Analysis

## Example Abstract Domain

Value is **unknown** to the analysis

Value is **undefined** by the analysis

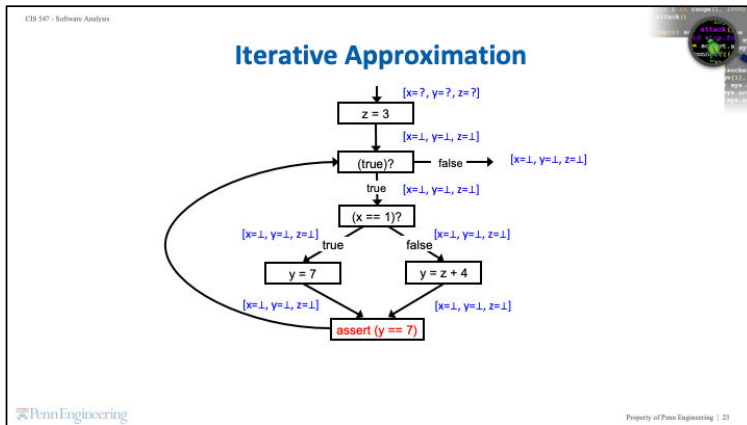
Property of Penn Engineering | 22

Designing a static analysis is an art: while the concrete semantics is dictated by the programming language at hand, there is no single best choice of abstract semantics. As we shall see later in this lesson, different choices yield different analysis results, and the right choice is dictated by the consumer of the analysis. Here, we will take a look at a particular abstract semantics for our problem of finding variables that have a constant value at a given program point.

The first step in designing an abstract semantics is to design an abstract domain. This abstract domain shows the possible abstract values that each integer-typed variable in the program can take. Besides all possible constant values, we also include two special values: 'top', denoted by question mark, to denote that the value is unknown to the analysis, and

'bottom' to denote that the value is undefined by the analysis.

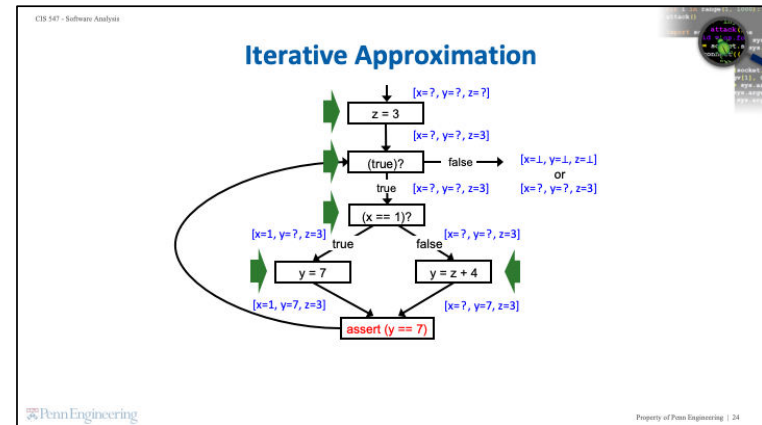
The abstract values are ordered in a structure called a lattice which dictates the possible orders in which a variable's abstract value might change as the analysis proceeds. Let's make things more clear by looking at an example.



We will use a common static analysis method called iterative approximation.

The analysis begins with abstract value 'top' for the three variables at the entry of the program and the abstract value 'bottom' at all other program points. This is the initial state of the analysis and captures the intuition that it has not yet visited any program point except the entry point. Furthermore, it captures the intuition that, at the entry point, variables x, y, and z are uninitialized and could therefore take arbitrary concrete values.

The term "program point" is ambiguous and representation dependent. It is unclear whether we mean before, after, or during a statement is executed. We will discuss this more formally later in the course. For the purpose of this module, we will consider a program point as an edge in the control flow graph.



At each step of the iterative approximation process, the analysis updates its knowledge about the values of the three variables at each program point. The analysis does this update based upon the information that it has inferred at the immediate predecessors of that program point.

For instance, after the statement that assigns 3 to z, the analysis knows that the value of z is the constant 3.

Now, the analysis reaches the conditional check in the loop "true?". In this case, our conditional expression is simple and it is easy to see that the true branch will always be taken, and the analysis updates its knowledge of the values of x, y, and z in that branch accordingly.

We call the false branch "infeasible" as the program will never take that path. In general, however, conditionals are not always this simple. Thus, the analysis designer has yet another choice to make. That is, should our analysis attempt to interpret conditionals? If the analysis were to not interpret conditionals, then it would have to assume that the false branch is feasible, and update the abstract state at the exit of the program accordingly, to reflect its knowledge that the values of x and y are unknown, and z is 3.

Another interesting update occurs in the true branch of the condition that checks whether the value of x is 1. Assuming our analysis attempts to interpret conditionals,

it knows that the value of  $x$  must be the constant 1 in the true branch. However, notice that in the false branch, the analysis does not know whether  $x$  has a constant value in all runs of this program. So it continues to indicate that the value of  $x$  is unknown. Regardless of whether our analysis attempts to interpret conditionals, this is the most precise abstract representation that we can have for  $x$ . This is because  $x \neq 1$  cannot be represented by our abstract domain. All we know is that  $x$  is some value other than 1, so we must represent it as unknown.

Similarly, after the statement that assigns 7 to  $y$ , the analysis knows that the value of  $y$  is the constant 7. The analysis has thus concluded that, every time this program point is reached in any run,  $x$  has value 1,  $y$  has value 7, and  $z$  has value 3.

Now let's look at the statement that assigns the expression  $z + 4$  to  $y$ . Since the analysis had previously discovered that the value of  $z$  before this statement is the constant 3, it can conclude that the value of  $y$  after this statement must be  $3 + 4$ , which is 7.

At this point, the analysis has concluded that, at each immediate predecessor of the assertion, the value of  $y$  is 7. It thereby concludes that the value of  $y$  in the assertion must be 7, and therefore that the assertion is valid.

The term iterative approximation implies that in general, the analysis might need to visit the same program point multiple times. This is because of the presence of loops, which can require the analysis to update facts that were previously inferred by the analysis at the same program point. We will emphasize this aspect in the following example.

CIS 547 - Software Analysis

### Another Example: Iterative Approximation

Fill in the value of variable  $b$  that the analysis infers at:

- 1)
- 2)
- 3)

1) the loop header  
2) entry of loop body  
3) exit of loop body

Enter "?" if a definite value cannot be inferred.

Penn Engineering Property of Penn Engineering | 25

Consider the following program. The analysis begins with the abstract value 'top' for variable  $b$  at the start of this program and the abstract value 'bottom' everywhere else. In each of the three boxes shown, let's determine the value of variable  $b$  that the analysis infers at the corresponding program point after completing its analysis. We will call these program points the loop header, the entry of the loop body, and the exit of the loop body, respectively. Be sure to take looping into consideration when performing your iterative analysis.

CS 547 - Software Analysis

### Another Example: Iterative Approximation

Fill in the value of variable  $b$  that the analysis infers at:

- 1) the loop header
- 2) entry of loop body
- 3) exit of loop body

Enter "?" if a definite value cannot be inferred.

Penn Engineering Property of Penn Engineering | 26

The value of  $b$  in the first box is 1. This is because immediately after the assignment of 1 to  $b$ , our static analysis knows that the value of  $b$  is 1.

As the analysis proceeds, it discovers that the value of  $b$  at the entry of the loop body is still 1.

Similarly, it discovers that the value of  $b$  at the exit of the loop body is 2. But the analysis is not done yet. It must analyze the loop again to ensure that these values are indeed sound.

The analysis revisits the entry of the loop body. This time, it notices that the value of  $b$  can be 1 or 2. So it updates the value of  $b$  at the entry of the loop body to unknown. Continuing further, the analysis updates the value of  $b$  at the exit of the loop body to unknown as well.

Due to these updates, the analysis analyzes the loop yet again. But this time, it concludes that the values of  $b$  at the entry and exit of the loop body have saturated. Therefore, the correct value of  $b$  in the 2nd and 3rd boxes is the unknown value.

Combining multiple abstract states from different program paths is called **merging**. Again, the analysis writer must make a decision in defining the combination operator. In this example, the combination operator is defined as a form of conjunction. That is,

for  $b$  to be equal to 1 in our abstract state, it **must** be equal to 1 in ALL program paths. Thus, when we merge the states from the first and second iterations of our loop,  $b$  is unknown.

However, consider an analysis in which we are trying to prove that  $b$  **may** be equal to 1 on some program path. In this case, we would want to define our combination operator as a form of disjunction. If  $b$  is equal to 1 on at least one program path, it should be equal to 1 in our abstract state. Again, this choice is made by the analysis developer by considering tradeoffs and the needs of the analysis user. We will look at the primary tradeoffs and consumers of program analysis later in this module.

CS 547 - Software Analysis

## Recap: Parts of a Static Analysis

- **Program representation**
  - e.g. control-flow graph, AST, or bytecode
- **Abstract domain**
  - e.g. how to approximate program values
- **Transfer functions**
  - e.g. assignments, conditionals, merge points, ...
- **Fixed-point computation algorithm**
  - e.g. iterative approximation

Penn Engineering Property of Penn Engineering | 27

In summary, a static analysis consists of the following components:

First we must specify the **program representation**. In order to analyze a program, we must be able to represent it precisely. Choices in program representation range broadly with different levels of abstraction. Common representation choices include control-flow graphs, Abstract Syntax Trees (ASTs), and bytecode.

Next, we must specify the **abstract domain**. Static analysis does not operate on actual program values, so we must represent concrete states with an approximate, abstract domain. In our example, we chose single constant values, in combination with 'top' and 'bottom' to comprise our abstract domain.

A **transfer function** specifies how to calculate the abstract state given a program statement. For example, in our iterative analysis, given the statement "x=1", the transfer function would set x to 1 in our abstract state. The transfer functions also specify how to combine information at control-flow merge points.

Lastly, the designer must specify a **fixed-point computation algorithm** that invokes the transfer functions of individual program statements to analyze the program. The algorithm computes a "fixed-point" meaning that it terminates when the abstract states are no longer changing.

At each step, the analysis designer has many choices. The correct choice depends on

the application of the analysis as there is no single "best" static analysis design.

CS 547 - Software Analysis

## QUIZ: Example Static Analysis Problem

Find statements that divide-by-zero

Use abstract domain:

```

void main() {
  while (x > 0) {
    S1: ... 1/x ...
    if (x > 0) {
      x--;
    } else {
      x++;
    }
  }
  S2: ... 1/x ...
}

```

Penn Engineering Property of Penn Engineering | 28

{QUIZ SLIDE}

Let's do a quiz to reinforce the concepts we've covered so far. Consider the problem of finding statements that may divide by zero. We will use a static analysis to solve this problem on the shown program, which contains two division statements S1 and S2.

An abstract domain suitable for this problem is the **sign domain**. It is a finite domain in which an abstract value can be one of the following:

- zero (0), which represents the integer value 0,
- minus (-), which represents any negative integer value,
- plus (+), which represents any positive integer value,
- top, denoted by ?, which represents unknown, and
- bottom, which represents uninitialized by the analysis.

Assume that your static analysis interprets conditionals such as  $x > 0$ .

CS 547 - Software Analysis

## QUIZ: Example Static Analysis Problem

Find statements that divide-by-zero

Use abstract domain:

S1:

S2:

Penn Engineering Property of Penn Engineering | 29

{QUIZ SLIDE}

For your convenience, here is the control-flow graph corresponding to the program. Enter the abstract value of x at program points S1 and S2.



CIS 547 - Software Analysis

### QUIZ: Example Static Analysis Problem

Find statements that divide-by-zero

Use abstract domain:

```

    graph TD
      Top["?"] --> Minus["-"]
      Top --> Zero["0"]
      Top --> Plus["+"]
      Minus --> Bottom["⊥"]
      Zero --> Bottom
      Plus --> Bottom
  
```

S1:  S2:  Property of Penn Engineering | 30

{SOLUTION SLIDE}

The answer depends on if we decide to interpret conditionals. If our analysis does not interpret conditionals, then it is easy to see that  $x$  is unknown at every program point. Since this is not very useful, we presumed that the analysis does interpret conditionals. The solution then is as follows. At S1,  $x$  is always positive, meaning that a divide-by-zero error cannot occur. At S2,  $x$  is unknown, so it is possible that a divide-by-zero error **may** occur. The reason is the statement that decrements  $x$ :  $x$  is always positive before this statement and therefore it could be positive or zero after the statement. In our abstract domain, the fact that  $x$  may be positive or zero is represented using the abstract value 'top'.

Notice that this approximation loses some information since 'top' denotes that  $x$  may be positive, zero, or negative at S2, when in fact  $x$  can never be negative at S2. Such approximations are so fundamental to program analyses that we will next characterize program analyses based on the kinds of approximations they make.

CIS 547 - Software Analysis

### READING

## A Menagerie of Program Abstractions

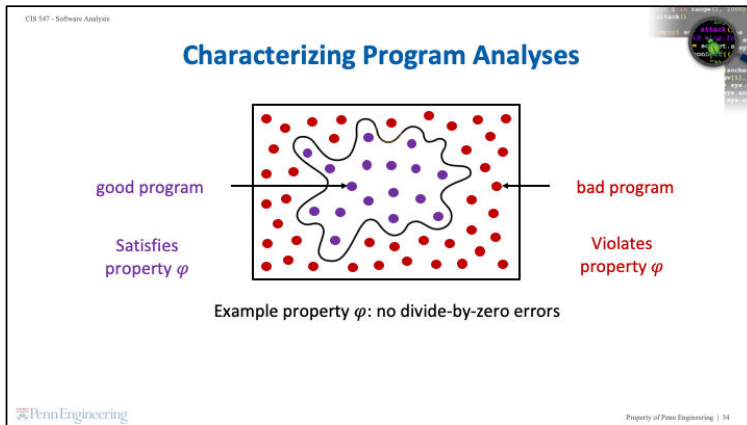
Property of Penn Engineering | 31

## LESSON

### Tradeoffs in Software Analysis

## SEGMENT

### Characterizing Analyses: Soundness and Completeness

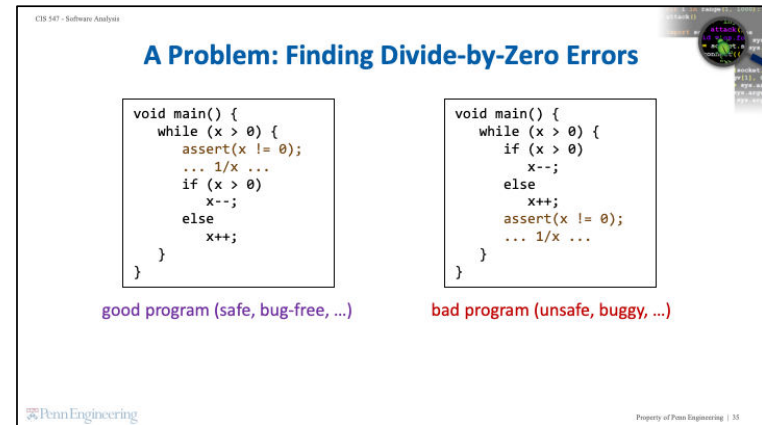


Since there are many different ways to design a program analysis, it is important that we have terms to characterize them. When we introduced static analysis, we discussed that the abstraction causes us to lose some information, sacrificing completeness. However, the analysis is **sound** because when it concludes some fact about the program, it is indeed true in all runs of the program. Now, we will dive deeper into these terms, and their consequences.

This figure shows a cross section of all possible programs plotted in three dimensions.

A program is labelled as "good" if it satisfies a particular property  $\varphi$ , and "bad" if it violates the property.

For example, let us consider a divide-by-zero analysis. A "good" program satisfies the property  $\varphi$  that it contains no divide-by-zero errors whereas a "bad" program violates this property in that it contains a divide-by-zero error. Consider this figure as the ground truth that our analysis is attempting to discover.

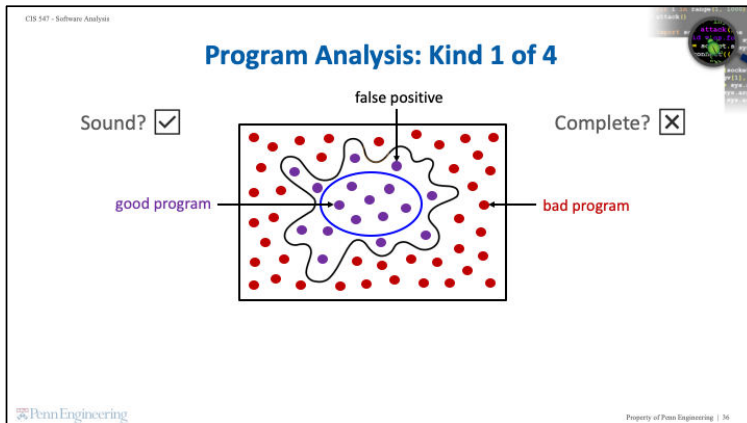


Let us consider our previous example.

The program on the left does not contain any divide-by-zero error as  $x$  is greater than zero at the entry of the loop body in all iterations of the loop. So, this program is considered **good**.

The program on the right **does** contain a divide-by-zero error in at least one run of the program. Consider the iteration in which  $x$  equals 1 at the entry of the loop body. The value is decremented to zero prior to dividing. So, this program is considered **bad**.

Given this ground truth, let us now proceed to characterize different program analyses in two aspects that we have informally considered thus far: soundness and completeness.



An analysis either accepts or rejects a given program.

We will depict each analysis in our illustration by an oval: the programs inside the oval are accepted by the analysis while the programs outside the oval are rejected by it.

An analysis is **sound** if it never accepts bad programs. That is, whenever a sound analysis accepts a program, that program is indeed free of divide-by-zero errors.

An analysis is **complete** if it never rejects good programs. That is, whenever a complete analysis rejects a program, that program indeed contains a divide-by-zero error.

A trivial analysis would be sound if it rejected every program. Conversely, a trivial analysis would be complete if it accepted every program. Thus, in designing a useful analysis, it is important to balance these two properties.

Now that we have covered these definitions, let us consider the example analysis, depicted by the blue oval. Let's take a moment to consider whether this analysis is sound, complete, or neither. *[pause]*

The example analysis is sound but incomplete. It is sound as all programs it accepts are indeed good. It is not complete because it rejects some good programs.

We call a good program that is rejected by an analysis as a **false positive**. The static analysis examples we saw earlier in this module are examples of this kind of analysis. Let's ascertain this next by applying one of those static analyses to our example good and bad programs.

Example: A Static Analysis

```

void main() {
  while (x > 0) {
    assert(x != 0);
    ... 1/x ...
  }
}

```

[x=?] or [x=+] → ... 1/x ...

good program

```

void main() {
  while (x > 0) {
    if (x > 0)
      x--;
    else
      x++;
    ... 1/x ...
  }
}

```

[x=?] → ... 1/x ...

bad program

Penn Engineering Property of Penn Engineering | 37

Consider the program on the left.

Consider the static analysis that uses the sign abstract domain and does not interpret conditionals.

At the point of the assertion, this analysis has no information about  $x$ . That is,  $x$  has abstract value 'unknown'. Such an analysis would reject this program and report a potential divide-by-zero error. This example shows how the choice of abstraction in an analysis can result in false positives.

Consider a different static analysis that uses the sign abstract domain but chooses to interpret conditionals.

This analysis would conclude that  $x$  is always positive at the point of the assertion, and accept this program, thereby averting a false positive.

Finally, consider the program on the right.

Regardless of whether our static analysis interprets conditionals, it has no information about  $x$  at the point of the assertion. That is,  $x$  has abstract value 'unknown'. Therefore, the analysis rejects this program and reports a potential divide-by-zero error.

Program Analysis: Kind 2 of 4

Sound?  Complete?

good program bad program false negative

Penn Engineering Property of Penn Engineering | 38

Now, let's consider a different kind of program analysis, depicted by the green oval. Let's take a moment to consider whether this analysis is sound, complete, or neither.

This analysis is complete but not sound. It is complete as all programs it rejects are indeed bad. It is not sound because it accepts some bad programs.

We call a bad program that is accepted by an analysis as a **false negative**. Dynamic analysis is an example of this kind of analysis. Let's ascertain this next by applying a dynamic analysis to our example good and bad programs.

CS 547 - Software Analysis

### Example: A Dynamic Analysis

```
void main() {
  while (x > 0) {
    assert(x != 0);
    ... 1/x ...
    if (x > 0)
      x--;
    else
      x++;
  }
}
```

good program

[x = 3] →

[x = -3] →

```
void main() {
  while (x > 0) {
    if (x > 0)
      x--;
    else
      x++;
    assert(x != 0);
    ... 1/x ...
  }
}
```

bad program

Penn Engineering Property of Penn Engineering | 39

Recall that a dynamic analysis infers facts about a program by monitoring its runs on a suite of test inputs. Similar to a static analysis, a dynamic analysis may abstract away information -- for instance, to keep the monitoring overhead low -- but assume that the dynamic analysis we are considering here does not perform any abstraction.

Suppose this analysis runs the program on the left with value  $x=3$ .

In this run, the analysis does not encounter a divide-by-zero error. The analysis will likely try many other values of  $x$  before accepting this program.

Now, suppose the same dynamic analysis runs the program on the right with the value  $x=-3$ . In this run, the analysis again does not encounter a divide-by-zero error.

This is because we never enter the loop or execute the problematic code. More generally, as long as the analysis runs this program with values of  $x$  less-than-or-equal-to 0, it will not detect the divide-by-zero error. This example illustrates how a dynamic analysis can incur a false negative.

CS 547 - Software Analysis

### QUIZ: Dynamic vs. Static Analysis

Match each box with its corresponding feature.

	Dynamic	Static
Cost		
Effectiveness		

A. Unsound  
(may miss errors)

B. Proportional to  
program's run time

C. Proportional to  
program's size

D. Incomplete  
(may report spurious errors)

Penn Engineering Property of Penn Engineering | 40

{QUIZ SLIDE}

OK, it's time for another quiz. Dynamic and static analyses strike different tradeoffs in terms of their cost and effectiveness. Match each box with its corresponding feature.

QUIZ: Dynamic vs. Static Analysis

Match each box with its corresponding feature.

	Dynamic	Static
Cost	B. Proportional to program's run time	C. Proportional to program's size
Effectiveness	A. Unsound (may miss errors)	D. Incomplete (may report spurious errors)

Penn Engineering Property of Penn Engineering | 41

{SOLUTION SLIDE}

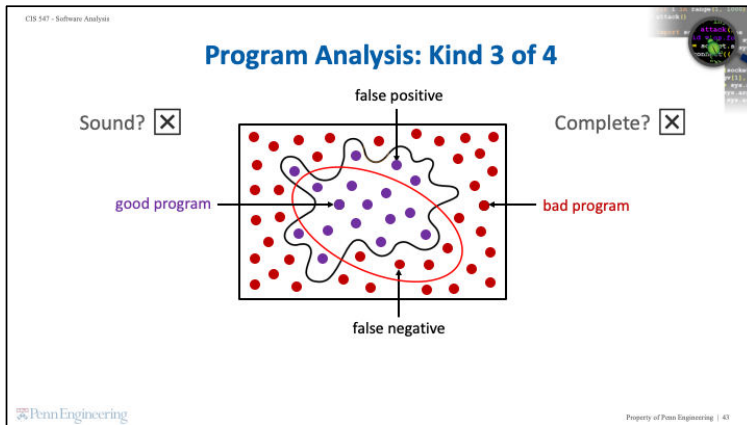
Let's review the answers. First we will focus on cost. Since dynamic analysis gathers information by running the program, its cost is proportional to the execution time of the program. A longer run thus costs more than a shorter one. Static analysis, on the other hand, gathers information by inspecting the program's code, and therefore its cost is proportional to the size of the program's source code. A larger program thus costs more than a smaller one.

Now let's look at effectiveness. As we saw in the divide-by-zero examples, a dynamic analysis may miss errors, as it inspects only a finite number of runs, whereas the program may contain an unbounded number of paths, some of which are not covered by those runs. We say that a dynamic analysis is inherently "unsound": in other words, it may produce false negatives. On the other hand, it is possible to design a static analysis that does not miss errors, but such an analysis may report spurious errors. We say that a static analysis is inherently "incomplete": in other words, it may produce false positives.

SEGMENT

Characterizing Analyses: F-Measure and Undecidability

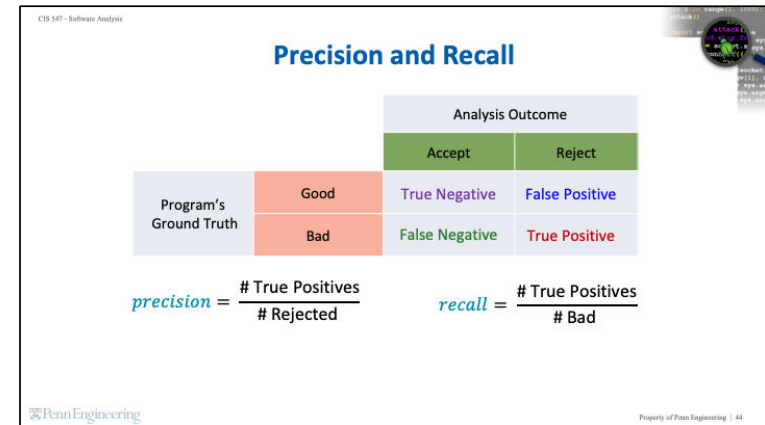
Penn Engineering Property of Penn Engineering | 42



Let us consider a new kind of program analysis depicted by the red oval. Take a moment to understand if this analysis is sound, complete, or neither.

This analysis is neither sound nor is it complete. It rejects some good programs, marked in blue, and accepts some bad programs, marked in green. Thus, it incurs both false positives and false negatives. However, that does not necessarily mean that this analysis is useless. In fact, this analysis may in practice be more accurate than the two kinds of analyses we previously discussed.

Consequently, we need a way to measure accuracy of analysis that goes beyond the absolute categorizations of soundness and completeness.



We arrive at four quadrants depending on the program's ground truth – whether it is good or bad – and the outcome of the analysis on the program – accept or reject. An "accurate" analysis should accept most good programs, reject most bad ones, and output few false positives and few false negatives.

**Precision** and **Recall** are standard ways to measure the accuracy of systems that output binary (yes/no) classifications. Since an analysis can be viewed as a system with binary output – it either accepts or rejects a given program – we adopt these metrics to quantify the accuracy of a given analysis.

**Precision** measures the number of bad programs among all programs that the analysis rejected. It measures the false positive rate of the analysis, and is calculated as the number of true positives divided by the total number of rejected programs.

**Recall** measures the number of bad programs that the analysis rejected among all bad programs. It measures the false negative rate of the analysis, and is calculated as the number of true positives divided by the total number of bad programs.

Informally, precision measures "how correct our results are" while recall measures "how complete are results are." In this way, precision and recall fill the gap between the absolute soundness and completeness categorizations.



CS 547 - Software Analysis

## F-Measure

$$precision = \frac{\# \text{ True Positives}}{\# \text{ Rejected}} \quad recall = \frac{\# \text{ True Positives}}{\# \text{ Bad}}$$

$$F \text{ measure} = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

- Caveat: May want to weigh **precision** and **recall** differently in practice
- Determined by the application's demands

Penn Engineering Property of Penn Engineering | 45

Recall that an accurate analysis is one with low false positive rate and low false negative rate.

This is captured using the **F-Measure**, or **F1 score**, which is a standard measure of accuracy that combines both precision and recall. It is computed by taking the harmonic mean of precision and recall.

An analysis that has perfect precision and recall would have an F-Measure of 1 while the worst analysis would have an F-Measure of 0. So, we aim for a combination of precision and recall that leads to an F-Measure near 1.

The F-Measure provides an intuitive way to quantify the accuracy of an analysis.

However, it assumes that precision and recall are equally important. This is not always the case.

The relative importance of precision and recall is highly dependent on the application of the analysis. Later in this lesson, we will see some examples in which different types of misclassifications incur different costs.

CS 547 - Software Analysis

## QUIZ: Comparing Program Analyses

Calculate the **precision**, **recall**, and **F-measure** of each of the following analyses:

	<b>precision</b>	<b>recall</b>	<b>F-measure</b>
sound and complete analysis			
sound analysis with 40% false positive (FP) rate			
complete analysis with 40% false negative (FN) rate			
analysis with 30% FP rate and 70% FN rate			

Penn Engineering Property of Penn Engineering | 46

{QUIZ SLIDE}

Let's do a quiz to reinforce our understanding of how to compare different program analyses by calculating their precision, recall, and F-Measure. Consider the following four analyses:

A sound and complete analysis;

A sound but incomplete analysis with a 40% false positive rate;

A complete but unsound analysis with a 40% false negative rate; and

An analysis that is neither sound nor complete, and has a 30% false positive rate and a 70% false negative rate.

CS 547 - Software Analysis

## QUIZ: Comparing Program Analyses

Calculate the **precision**, **recall**, and **F-measure** of each of the following analyses:

	precision	recall	F-measure
sound and complete analysis	1.00	1.00	1.00
sound analysis with 40% false positive (FP) rate	0.60	1.00	0.75
complete analysis with 40% false negative (FN) rate	1.00	0.60	0.75
analysis with 30% FP rate and 70% FN rate	0.70	0.30	0.42

Penn Engineering Property of Penn Engineering | 47

### {SOLUTION SLIDE}

The solution is as follows. An analysis that is sound and complete has the highest possible precision, recall, and F-Measure, which is 1.

Precision is the fraction of rejected programs that are bad whereas recall is the fraction of bad programs that are rejected.

For the 2<sup>nd</sup> analysis, we know from its false positive rate of 40% that the fraction of programs rejected by it that are good is 0.4. Therefore, its precision is  $1 - 0.4$ , or 0.6. Since this analysis is sound, its recall is 1. The F-Measure is the harmonic mean of the precision and recall, which is  $2 / (1/0.6 + 1/1) = 0.75$ .

For the 3<sup>rd</sup> analysis, we know from its false negative rate of 40% that the fraction of bad programs that are accepted by it is 0.4. Therefore, its recall is  $1 - 0.4$ , or 0.6. Since the analysis is complete, its precision is 1. The F-Measure is the same as before, that is, 0.75.

For the 4<sup>th</sup> analysis, we obtain a precision of 0.7 from its false positive rate of 30%, and a recall of 0.3 from its false negative rate of 70%. The F-Measure calculated as the harmonic mean of 0.7 and 0.3 evaluates to 0.42.

CS 547 - Software Analysis

## Program Analysis: Kind 4 of 4

Sound?

Complete?

good program

bad program

false positive

false negative

Penn Engineering Property of Penn Engineering | 48

Now, let's consider the last kind of analysis: an ideal analysis. This analysis perfectly classifies each program: it has no false positives and no false negatives. Equivalently, it has an F-Measure of 1. Such an analysis is both sound and complete. Is it possible to design such an analysis? Let's find out.

CIS 547 - Software Analysis

## Undecidability of Program Properties

- Can program analysis be **sound** and **complete**?
  - Not if we want it to **terminate!**
- Questions like “is a program point reachable on some input?” are **undecidable**
- Designing a program analysis is an art
  - **Tradeoffs** dictated by consumer

Penn Engineering Property of Penn Engineering | 49

Can a program analysis guarantee both soundness and completeness?

The answer is: not if we want the analysis to terminate on every given program!

Even seemingly simple program properties for realistic programming languages like C and Java are undecidable. An example such property is whether a given point in a given program is reachable on some input to that program.

You can find a link to recommended reading on the topic of undecidability in the lecture handout.

[https://en.wikipedia.org/wiki/Undecidable\\_problem](https://en.wikipedia.org/wiki/Undecidable_problem)

Designing a program analysis is thus an art that involves striking a suitable tradeoff between termination, soundness, and completeness.

This tradeoff is typically dictated by the consumer of the program analysis. Let's look at the primary consumers of program analysis next.

CIS 547 - Software Analysis

## READING

### Undecidability of Program Properties

Penn Engineering Property of Penn Engineering | 50

CIS 547 - Software Analysis

**SEGMENT**

**Who Needs Program Analysis?**

Penn Engineering

Property of Penn Engineering | 51

CIS 547 - Software Analysis

**Who Needs Program Analysis?**

Three primary consumers of program analysis:

- Compilers
- Software Quality Tools
- Integrated Development Environments (IDEs)

Penn Engineering

Property of Penn Engineering | 52

There are three primary consumers of program analysis: compilers, software quality tools, and integrated development environments.

CIS 547 - Software Analysis

## Compilers

- Bridge between high-level languages and architectures
- Use program analysis to generate efficient code

```
int p(int x) { return x*x; }
void main(int arg) {
  int z;
  if (arg != 0)
    z = p(6) + 6;
  else
    z = p(-7) - 7;
  print(z);
}
```

z = 42

➔

```
int p(int x) { return x*x; }
void main() {
  print(42);
}
```

- Runs faster
- More energy-efficient
- Smaller in size

Penn Engineering Property of Penn Engineering | 53

Compilers bridge the gap between high-level programming languages and advanced computer architectures.

They use program analyses to generate efficient code on a target architecture for programs written in a high-level source language.

Let us see a simple example of how a program analysis can help a compiler generate more efficient code.

Consider this example program.

We saw earlier in this lesson how a static analysis can discover the program invariant ( $z == 42$ ) at the end of this program. A compiler can use this invariant to simplify this program.

The simplified program simply prints 42. It is easy to see that this simplified program is more efficient than the original program: it runs faster, it is more energy-efficient, and it is smaller in size.

CIS 547 - Software Analysis

## Software Quality Tools

- Primary focus of this course
- Tools for testing, debugging, and verification
- Use program analysis for:
  - Finding programming errors
  - Proving program invariants
  - Generating test cases
  - Localizing causes of errors
  - ...

```
int p(int x) { return x * x; }
void main() {
  int z;
  if (getc() == 'a')
    z = p(6) + 6;
  else
    z = p(-7) - 7;
  if (z != 42)
    disaster();
}
```

z = 42

Penn Engineering Property of Penn Engineering | 54

The second key consumer of program analysis is software quality tools, which will be the primary focus of this course.

This category broadly includes tools programmers use for tasks to improve software quality, such as testing, debugging, and verification.

These tools use program analyses for various purposes such as finding programming errors, proving program invariants, generating test cases, and localizing the causes of errors.

Consider this example program again. The invariant ( $z == 42$ ) discovered at this program point by a static analysis could be used by a program verification tool to prove that this program will never call disaster.

CS 547 - Software Analysis

## Integrated Development Environments

- Examples: **Eclipse** and **Microsoft Visual Studio**
- Use program analysis to help programmers:
  - Understand programs
  - Refactor programs
    - Restructuring a program without changing its behavior
- Useful in dealing with large, complex programs

Penn Engineering Property of Penn Engineering | 55

The third main consumer of program analysis is integrated development environments such as Eclipse or Microsoft Visual Studio.

Such environments use program analyses to help programmers to understand programs and to refactor programs, which is the process of restructuring a program without changing its observable behavior.

These features are especially needed when dealing with large, complex programs which are common in practice.

CS 547 - Software Analysis

## QUIZ (1/3): Choosing a Program Analysis

For each analysis (**A1**, **A2**, **A3**), state whether it is sound, complete, both, or neither.

	Sound?	Complete?
<b>A1</b>		
<b>A2</b>		
<b>A3</b>		

Penn Engineering Property of Penn Engineering | 56

{QUIZ SLIDE}

Suppose the jagged SHADED portion denotes all programs that do NOT contain divide-by-zero errors, and the UNSHADED portion within the black rectangle denotes all programs that DO contain such errors.

Let  $A_1$ ,  $A_2$ , and  $A_3$  be different program analyses which check for divide-by-zero errors. Each analysis either ACCEPTS a given program (that is, declares it free of divide-by-zero errors) or REJECTS it (that is, declares that some divide-by-zero error exists in it).

For each analysis, the programs accepted by that analysis are contained INSIDE the corresponding oval, and the programs rejected by that analysis are contained OUTSIDE the corresponding oval.

Answer the following questions.

Is  $A_1$  Sound? Is  $A_1$  Complete?  
 Is  $A_2$  Sound? Is  $A_2$  Complete? And lastly,  
 Is  $A_3$  Sound? Is  $A_3$  Complete?

QUIZ (1/3): Choosing a Program Analysis

For each analysis (A1, A2, A3), state whether it is sound, complete, both, or neither.

space of all programs

	Sound?	Complete?
A1	No	No
A2	Yes	No
A3	No	Yes

{SOLUTION SLIDE}

Let's review the solutions. We will use the term 'correct program' for a program that does not contain any divide-by-zero error, and the term 'buggy program' for a program that does contain a divide-by-zero error.

A1 is not sound as it accepts some buggy programs. That is, it incurs false negatives. It is also not complete as it rejects some correct programs. That is, it incurs false positives.

A2 is sound as it never accepts buggy programs. That is, it does not incur false negatives. However, it is not complete as it rejects some correct programs. That is, it incurs false positives.

A3 is not sound as it accepts some buggy programs. That is, it incurs false negatives. However, it is complete as it never rejects correct programs. That is, it does not incur false positives.

QUIZ (2/3): Choosing a Program Analysis

NASA engineer Bob has been asked to apply program analysis to check divide-by-zero errors in software that will control NASA's next billion-dollar space mission.

	Sound?	Complete?
A1	No	No
A2	Yes	No
A3	No	Yes

Which analysis (A1/A2/A3) should Bob use?

{QUIZ SLIDE}

NASA engineer Bob has been asked to apply program analysis to check for divide-by-zero errors in software that will control NASA's next billion-dollar space mission. Which analysis (A1 / A2 / A3) should Bob use? Briefly justify your answer.

CS 547 - Software Analysis

### QUIZ (2/3): Choosing a Program Analysis

NASA engineer **Bob** has been asked to apply program analysis to check divide-by-zero errors in software that will control NASA's next billion-dollar space mission.

	Sound?	Complete?
<b>A1</b>	No	No
<b>A2</b>	Yes	No
<b>A3</b>	No	Yes

Which analysis (**A1/A2/A3**) should **Bob** use?

A2

Penn Engineering Property of Penn Engineering | 59

{SOLUTION SLIDE}

The correct answer is A2. Safety critical systems like infrastructure systems software, aviation and aerospace software, and real time systems must be proven to be sound -- lives depend on them! If the program is accepted by A2, then Bob can be sure there are no divide-by-zero errors in the code.

CS 547 - Software Analysis

### QUIZ (3/3): Choosing a Program Analysis

Microsoft developer **Ann** has agreed to apply program analysis to check divide-by-zero errors in her programs on the condition that it will not produce false alarms.

	Sound?	Complete?
<b>A1</b>	No	No
<b>A2</b>	Yes	No
<b>A3</b>	No	Yes

Which analysis (**A1/A2/A3**) should **Ann** use?

Penn Engineering Property of Penn Engineering | 60

{QUIZ SLIDE}

Microsoft developer Ann has agreed to apply program analysis to check for divide-by-zero errors in her programs, on the condition that it will not produce any false alarms. Which analysis (A1 / A2 / A3) should Ann use? Briefly justify your answer.



CIS 547 - Software Analysis

### QUIZ (3/3): Choosing a Program Analysis

Microsoft developer **Ann** has agreed to apply program analysis to check divide-by-zero errors in her programs on the condition that it will not produce false alarms.

	Sound?	Complete?
<b>A1</b>	No	No
<b>A2</b>	Yes	No
<b>A3</b>	No	Yes

Which analysis (**A1/A2/A3**) should **Ann** use?

**A3**

Penn Engineering Property of Penn Engineering | 61

{SOLUTION SLIDE}

The correct answer is A3. If Ann wants to use a program analysis that will not produce false positives, she should use a complete analysis. A3 is the only analysis that is complete. If it rejects a program, we can be sure that program has a divide-by-zero error.

CIS 547 - Software Analysis

## LESSON

### Review

Penn Engineering Property of Penn Engineering | 62

CS 547 - Software Analysis

SEGMENT

What Have We Learned?

Penn Engineering

Property of Penn Engineering | 63

CS 547 - Software Analysis

What Have We Learned?

- What is program analysis?
- Dynamic analysis vs. static analysis: pros and cons
- Program invariants
- Iterative approximation method for static analysis
- Characterizing program analyses

Undecidability => program analysis cannot ensure  
termination + soundness + completeness

- Who needs program analysis?

Penn Engineering

Property of Penn Engineering | 64

Let's recap the main topics that we have covered in this lesson.

First, we introduced program analysis, a process for automatically discovering useful facts about programs.

We then discussed two kinds of program analyses: dynamic analysis and static analysis. The primary difference between these two kinds of analyses is that dynamic analysis works by running the program whereas static analysis works by inspecting the program's code. We discussed the pros and cons of these two kinds of analyses.

We learnt about program invariants and their role as useful program facts. We discussed how dynamic analysis can discover likely invariants, and how static analysis can prove invariants.

We also saw step-by-step how static analysis can prove a certain kind of program invariant using the method of iterative approximation.

We then learned how to characterize program analyses in terms of soundness and completeness. We also explored precision and recall in an attempt to bridge to gap between these absolute categorizations.

We learnt that the undecidability of even seemingly simple program properties prevents program analyses from simultaneously guaranteeing the three desirable

features of termination, soundness, and completeness.

Finally, we discussed the three main consumers of program analysis: compilers, software quality tools, and integrated development environments.