Program Analysis:

Introduction and Basics

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

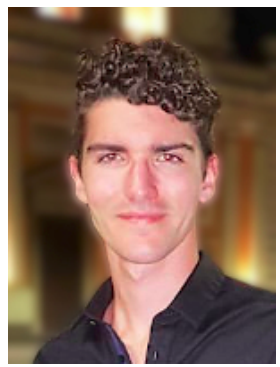
Winter 2023/2024

About Me: Michael Pradel

- **Since 9/2019: Full Professor at University of Stuttgart**
- **Before Stuttgart**
  - Studies at TU Dresden, ECP (Paris), and EPFL (Lausanne)
  - PhD at ETH Zurich, Switzerland
  - Postdoctoral researcher at UC Berkeley, USA
  - Assistant Professor at TU Darmstadt
  - Sabbatical at Facebook, Menlo Park, USA

About the Software Lab






- **My research group since 2014**
- **Focus: Tools and techniques for building reliable, efficient, and secure software**
  - Program testing and analysis
  - Machine learning, security
- **Thesis and job opportunities**

Plan for Today

- **Introduction**
  - What the course is about
  - Why it is interesting
  - How it can help you
- **Organization**
  - Lectures, exercises, course project
  - Final exam
- **Foundations**
  - Grammars, ASTs, CFGs, etc.

What is program analysis?

Program Testing & Analysis

What you probably know:

- **Manual testing or semi-automated testing:**
  JUnit, Pytest, Selenium, etc.
- **Manual ”analysis” of programs:**
  Code inspection, debugging, etc.

Focus of this course:

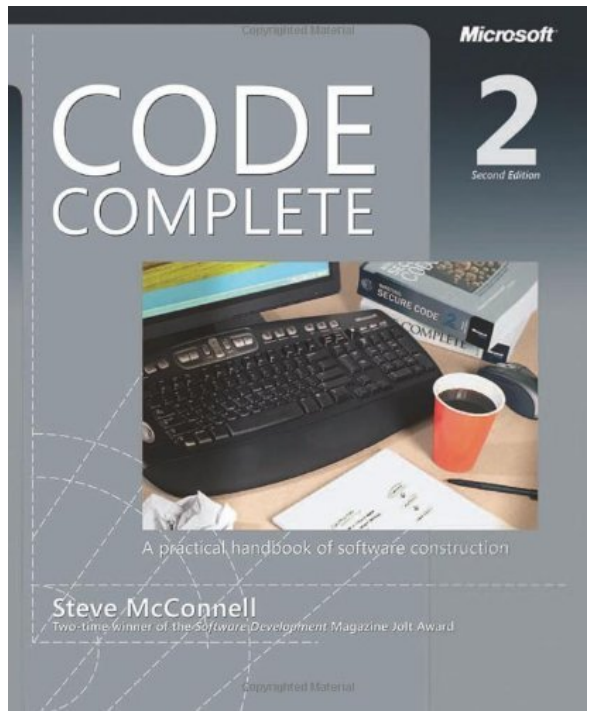
Automated testing and program analysis

Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm

Why Do We Need It?

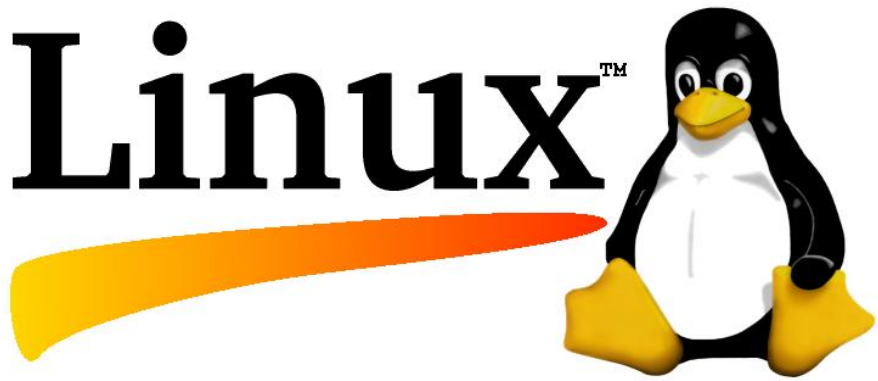
- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm



0.5-25/KLoC in delivered software

Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm



1.5 years to find a bug

[Palix2011]

Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm

Ariane 5

Northeast blackout

Therac-25

What is Program Analysis?

- **Automated analysis of program behavior, e.g., to**
  - find programming errors
  - optimize performance
  - find security vulnerabilities

What is Program Analysis?

- **Automated analysis of program behavior, e.g., to**
  - find programming errors
  - optimize performance
  - find security vulnerabilities

What is Program Analysis?

- **Automated analysis of program behavior, e.g., to**
  - find programming errors
  - optimize performance
  - find security vulnerabilities

Static vs. Dynamic Analysis

Static

- Analyze source code, byte code, or binary
- Typically:
  - Consider all inputs
  - Overapproximate possible behavior

Dynamic

- Analyze program execution
- Typically:
  - Consider current input
  - Underapproximate possible behavior

Static vs. Dynamic Analysis

Static

- Analyze source code, byte code, or binary
- Typically:
  - Consider all inputs
  - Overapproximate possible behavior

E.g., compilers, lint-like tools

Dynamic

- Analyze program execution
- Typically:
  - Consider current input
  - Underapproximate possible behavior

E.g., automated testing, profilers

Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
  out = "no";
if (r === 1)
  out = "maybe";
console.log(out);
```

What are the possible outputs?

Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
  out = "no";
if (r === 1)
  out = "maybe"; // infeasible path
console.log(out);
```

Overapproximation: ”yes”, ”no”, ”maybe”

- Consider all paths (that are feasible based on limited knowledge)

Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
  out = "no";
if (r === 1)
  out = "maybe"; // infeasible path
console.log(out);
```

Underapproximation: ”yes”

- Execute the program once

Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
  out = "no";
if (r === 1)
  out = "maybe"; // infeasible path
console.log(out);
```

Sound and complete: ”yes”, ”no”

- For this example: Can explore both feasible paths

Another Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);
```

What are the possible outputs?

Another Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);
```

Overapproximation: Any value

- Consider all paths (that are feasible based on limited knowledge about `random()`)

Another Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);
```

Underapproximation:

Some number in [0,2), e.g., 1.234

- Execute the program once

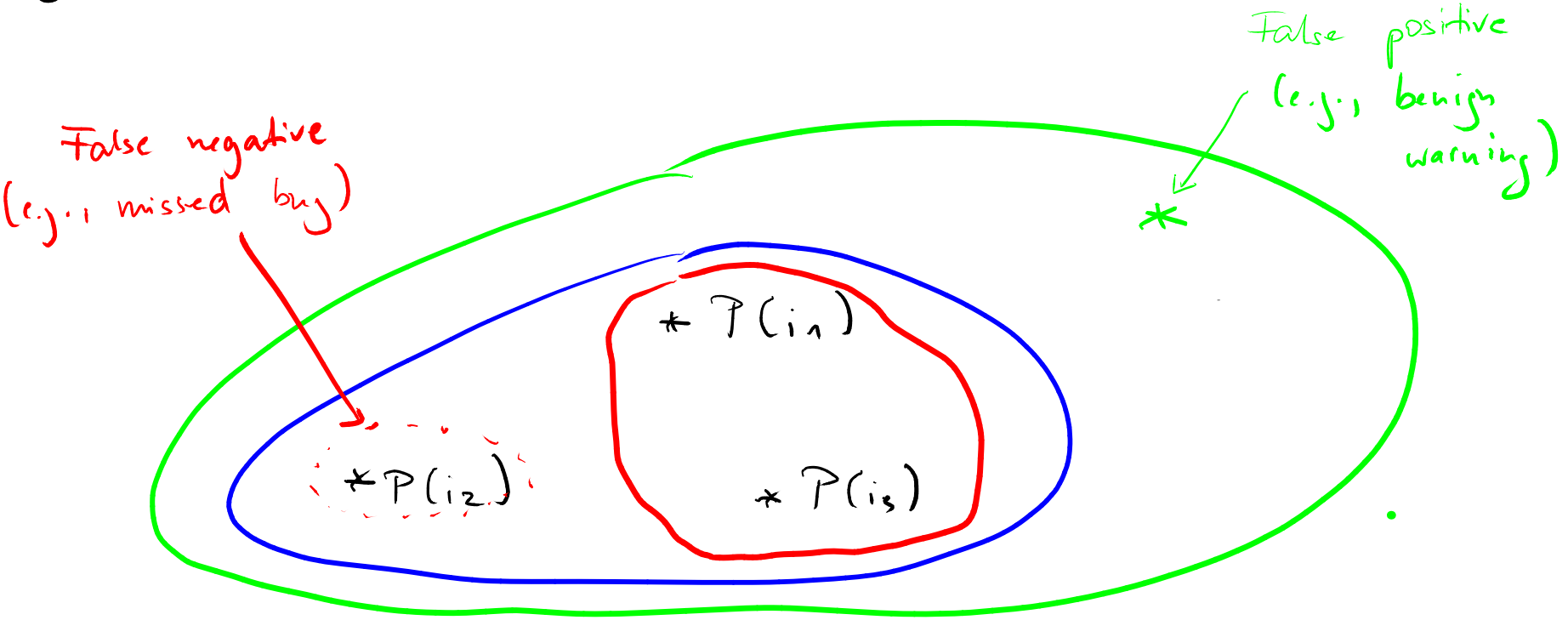
Another Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);
```

Sound and complete?

- Exploring all possible outputs:
  Practically impossible
- This is the case for most real-world programs

Program P, Input i, Behavior $P(i)$



All possible behaviors (want this, ideally)

Overapproximation (most static analysis)

Underapproximation (e.g., testing, dynamic analysis)

Test Generation

- **Dynamic analysis:**

  Requires input to run the program

- **Test generation:**

  Creates inputs automatically

- **Examples**
  - Generate JUnit tests:

    Input = sequence of method calls
  - UI-level test generation:

    Input = sequence UI events
  - Fuzz-test a compiler: Input = program

How Does All This Help Me?

- **Use program analysis tools**
  - Improve the quality of your code
- **Understand program analysis**
  - Better understanding of program behavior
- **Create your own tools**

Plan for Today

- **Introduction**
  - What the course is about
  - Why it is interesting
  - How it can help you
- **Organization** 
  - Lectures, exercises, course project
  - Final exam
- **Foundations**
  - Grammars, ASTs, CFGs, etc.

Organization

- Lectures
- Exercises
- Course project
- Final exam

Organization

Grading:

- **Lectures**
- **Exercises** → **10%**
- **Course project** → **40%**
- **Final exam** → **50%**

Lectures

- **15 lectures**
- **Mondays (3:45pm) and Tuesdays (11:30am)**
  - Not all slots are used: Check the schedule

Exercises

- **4 exercises**
- **Pen and paper**
- **Timeline**
  - Published on day X
  - Submission due on X + 7 days
  - Discussion session soon afterwards
- **Individual work: No collaboration or sharing of solutions**

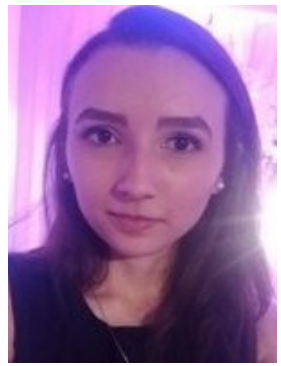
Course Project

- **Design, implement, and evaluate a program analysis based on an existing framework**
  - Dynamic analaysis of Python code
  - Based on DynaPyt framework: https://github.com/sola-st/DynaPyt
- **Individual, independent project**

Mentoring

Each student has a mentor

Aryaz Eghbali



Beatriz Souza

- First point of contact for all project-related questions
- Three 1:1 progress meetings
- Email or schedule additional meetings when needed

Course Project: Timeline

- **Published on November 6**
- **Three progress meetings**
- **Due on February 1**
  - Implementation and results
  - Report
- **Presentation: February 5 to 9**

Course Project: Timeline

Grading:
(% of overall grade)

- **Published on November 6**
- **Three progress meetings** — **10%**
- **Due on February 1** — **10%**
  - Implementation and results — **10%**
  - Report — **10%**
- **Presentation: February 5 to 9**

Academic Integrity

- Work you submit must be your own
- Unauthorized group efforts and any form of plagiarism are considered academic dishonesty and will be punished
- Allowed to discuss the problem with your peers, but not to reuse any part of an existing solution

Final Exam

- **Content of lectures and reading material**
- **Written**
- **One hour**
- **Open-book**
  - Tests your understanding, not your knowledge

Vertiefungsprüfung

Alternative to written exam:

Combined oral exam

- A.k.a. “Vertiefungsprüfung”
- Oral exam about content of two related courses
- Specialization/“Vertiefungslinie”:

  Software analysis
- Rules for course project etc. are the same

Content

Introduction and basics

Operational semantics

Data flow analysis

Slicing

Dynamic analysis frameworks

Test generation (fuzzing, symbolic)

Information flow analysis

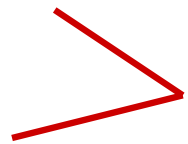
Call graphs

Path profiling

Analyzing concurrent programs

Content

Introduction and basics
Operational semantics
Data flow analysis
Slicing
Dynamic analysis frameworks
Test generation (fuzzing, symbolic)
Information flow analysis
Call graphs
Path profiling
Analyzing concurrent programs



Foundations

Content

Introduction and basics

Operational semantics

Data flow analysis

Slicing

Dynamic analysis frameworks

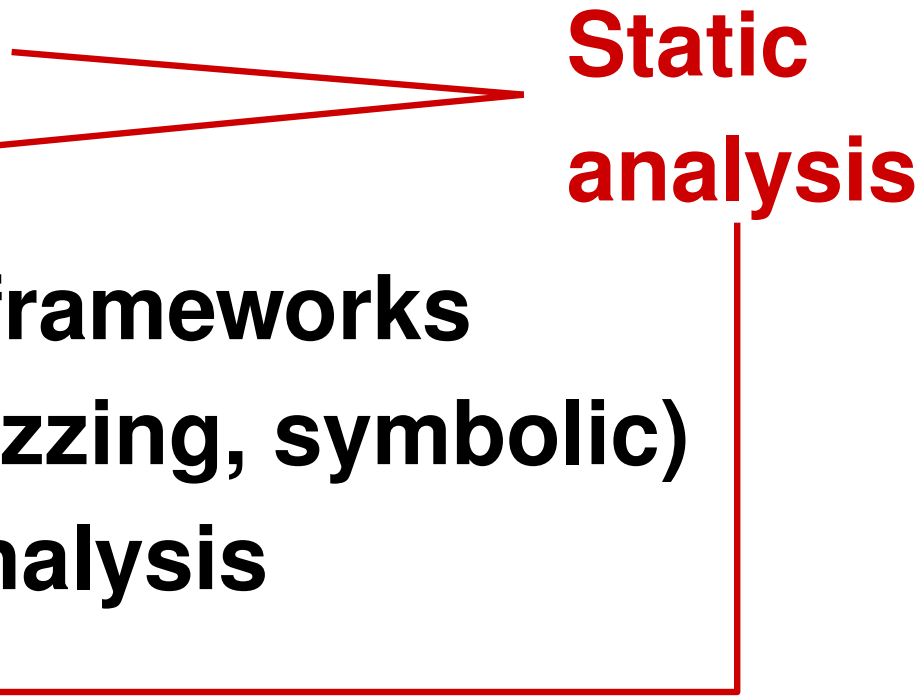
Test generation (fuzzing, symbolic)

Information flow analysis

Call graphs

Path profiling

Analyzing concurrent programs



Content

Introduction and basics

Operational semantics

Data flow analysis

Slicing

Dynamic analysis frameworks

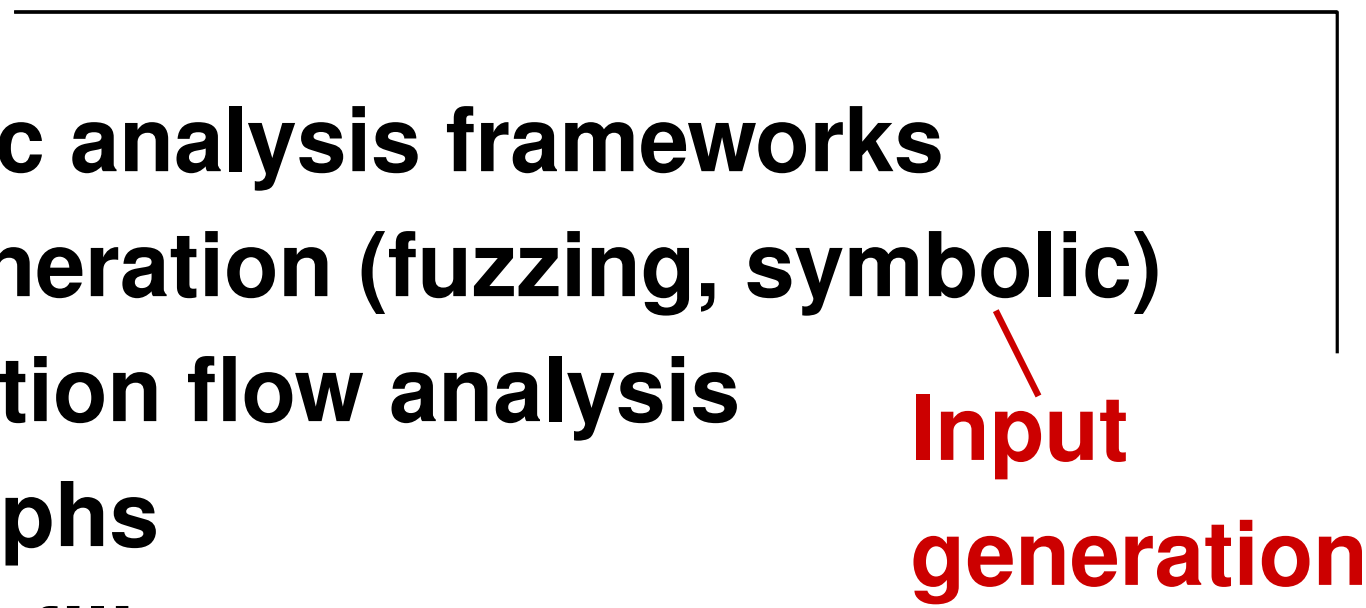
Test generation (fuzzing, symbolic)

Information flow analysis

Call graphs

Path profiling

Analyzing concurrent programs



Content

Introduction and basics

Operational semantics

Data flow analysis

Slicing

Dynamic analysis frameworks

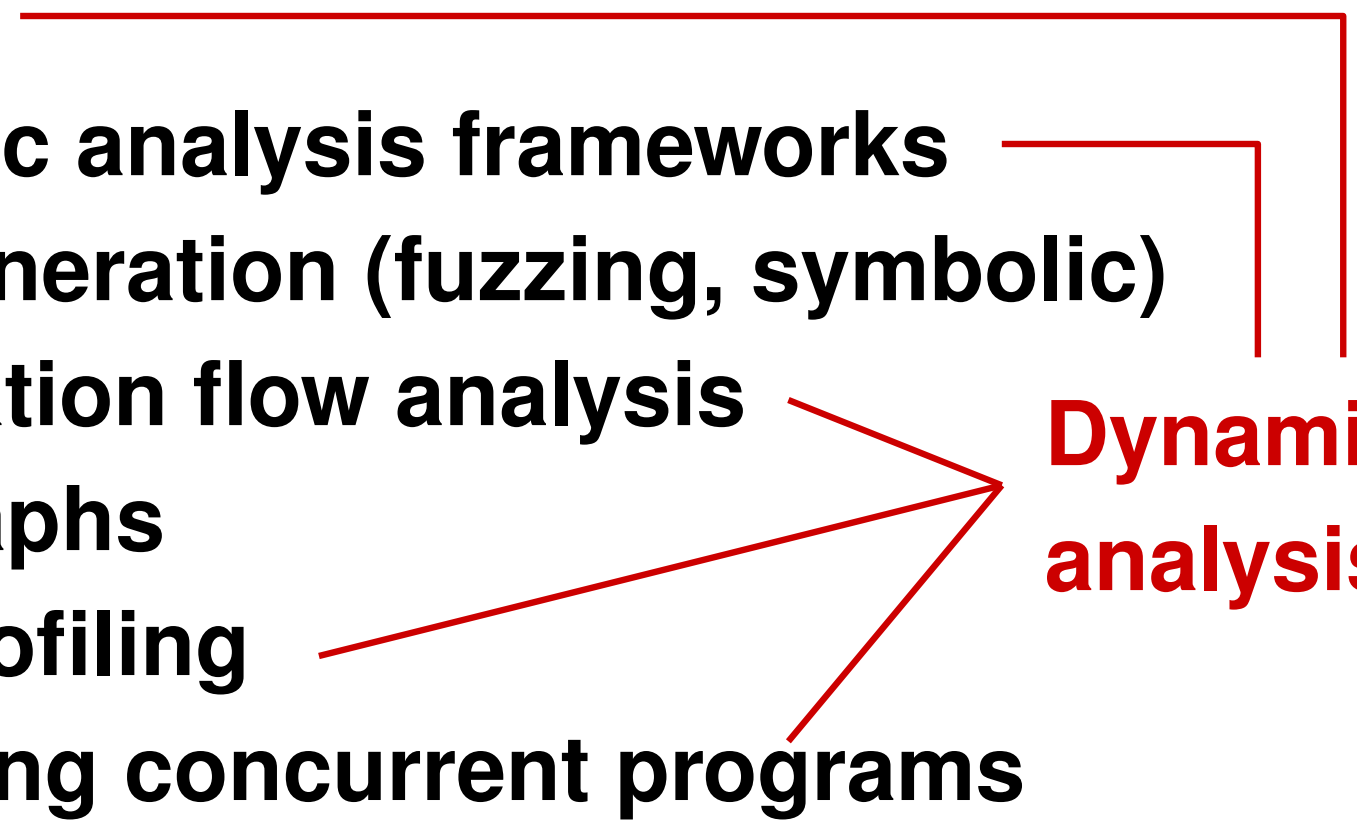
Test generation (fuzzing, symbolic)

Information flow analysis

Call graphs

Path profiling

Analyzing concurrent programs



Content

Introduction and basics

Operational semantics — Exercise 1

Data flow analysis — Exercise 2

Slicing

Dynamic analysis frameworks — Exercise 3

Test generation (fuzzing, symbolic)

Information flow analysis — Exercise 4

Call graphs

Path profiling

Analyzing concurrent programs

Content

Introduction and basics

Operational semantics

Data flow analysis

Slicing

Dynamic analysis frameworks

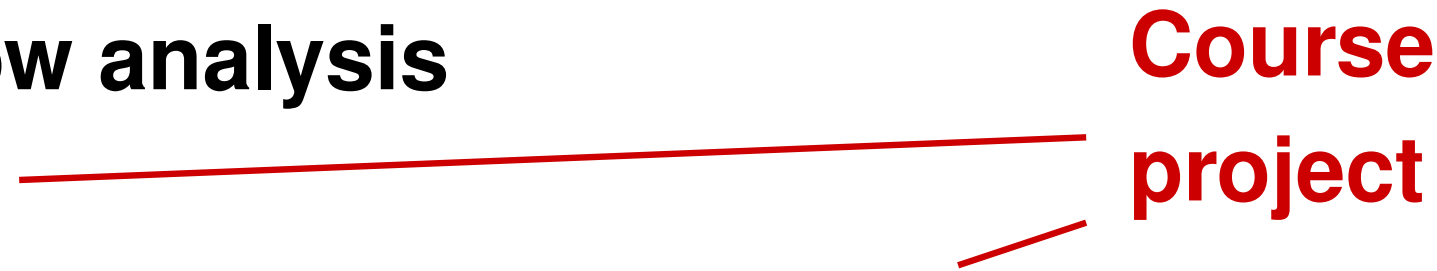
Test generation (fuzzing, symbolic)

Information flow analysis

Call graphs

Path profiling

Analyzing concurrent programs



Learning Material

There is no script or single book that covers everything

- Slides and hand-written nodes:
  Available after lecture
- Pointers to papers, book chapters, and web resources

Programming Language

Most concepts taught in this course:

Language-independent

Examples:

Various programming languages

- JavaScript, Java, C++, Python, etc.

Course project: Python

- Both target language and analysis language

Schedule

- **Classroom activities**
  - Lectures and discussion of exercises
- **Individually scheduled activities**
  - Progress meetings
  - Project presentations
- **Asynchronous activities**
  - Working on exercises and project
- **Strict deadlines**
  - Submission of exercises and course project

Ilias

Platform for questions, discussions, and sharing additional material

- Please register for the course
- Use it for all questions related to the course
- Messages sent to all students go via the Ilias forum (pro tip: enable notifications)

Link to Ilias course on
software-lab.org/teaching/winter2023/pa/

A Friendly Warning

This is not going to be an easy course!

- Do the exercises
- Work regularly on the course project

... but the effort is worth it!

Plan for Today

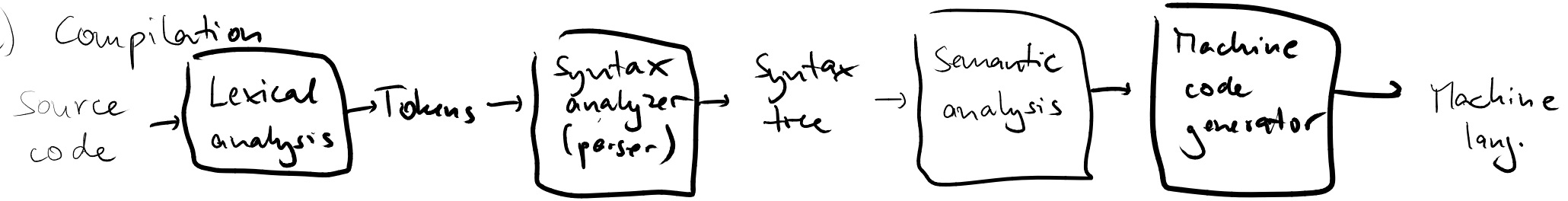
- **Introduction**
  - What the course is about
  - Why it is interesting
  - How it can help you
- **Organization**
  - Course projects
  - Term paper
  - Mid-term and final exam
- **Foundations** 
  - Grammars, ASTs, CFGs, etc.

PL = syntax + semantics + implementation

form — meaning — execute

Implementation

a) Compilation



b) Interpreter

c) Hybrid

↳ e.g., Java, JavaScript

Syntax

a) Grammar → Which programs are syntactically correct?

4 parts: terminals Σ, non-terminals N, productions P,
initial symbol $s \in N$

Example: Arithmetic expression

$\Sigma = \{0, 1, 2, \ldots, 9, +, -\}$

$N = \{Exp, Num, Op, Digit\}$

$s = Exp$

$P = Exp \rightarrow Num \mid Exp\ Op\ Exp$

$Op \rightarrow + \mid -$

$Num \rightarrow Digit \mid Digit\ Num$

$Digit \rightarrow 0 \mid 1 \mid \ldots \mid 9$

What is part of the language?

a) 12 - 2 ✓

b) 2 + (12 - 4) ✗

c) 11 * 4 ✗

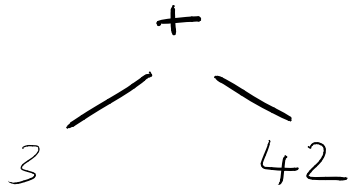
d) 12345678901234 ✓

b) Abstract syntax tree

- abstract grammar
  - ↳ e.g., $E \rightarrow n \mid O_p(E, E)$

    $O_p \rightarrow + \mid -$
  - ↳ terminals = tokens

- Example : $3 + 42$



Control flow graphs

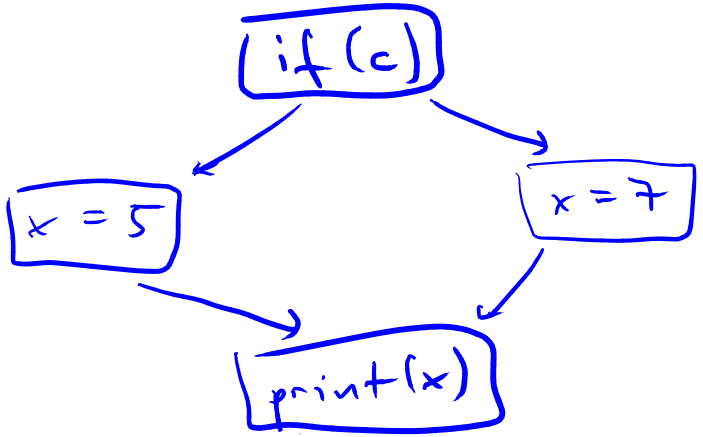
↳ Model flow of control through a program

$G = (N, E)$ where N... basic blocks = sequence of operations that are executed together

E... possible transfers of control

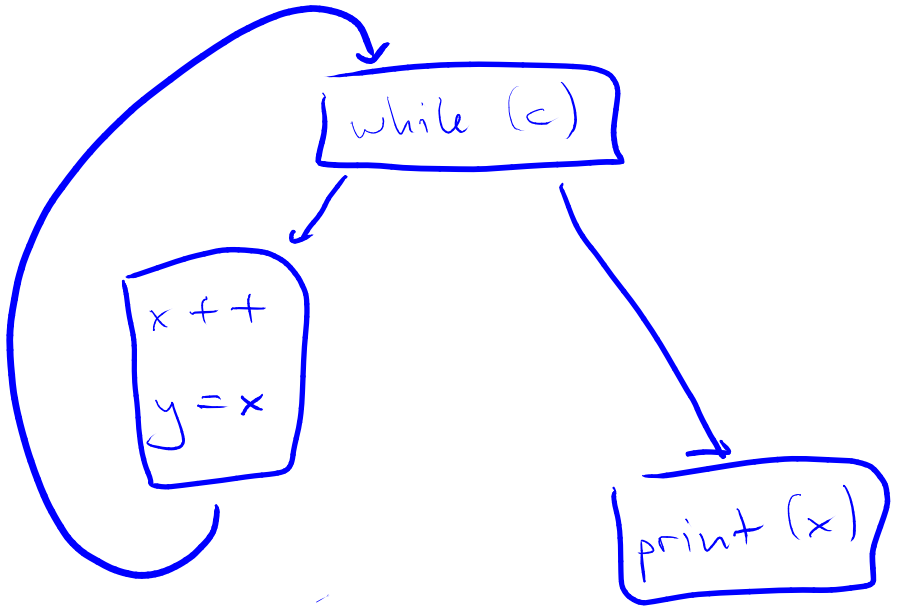
Ex. 1

```
if (c)
    x = 5
else
    x = 7
print (x)
```



Ex. 2

```
while (c) {
    x++;
    y = x;
}
print (x);
```



Data Dependence Graph

↳ Model flow of data from "definitions" to "uses"

$G = (N, E)$ where N ... operations

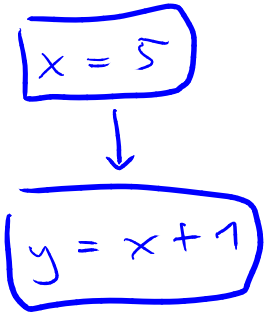
E ... possible def-use relations

$e = (n_1, n_2)$ means that n_2 may use data defined at n_1

Ex:

$x = 5$

$y = x + 1$



Ex. 2

```
x = 3
y = 5
if (x ≥ 1)
    y = x
z = x + y
```