SCSR – 2023/24

Software Correctness, Security, and Reliability

Lecture 1: Introduction

Software and System Verification Team

https://ssv.dais.unive.it/

Software and System Verification Team

- **Main topics**
  - Application of formal methods and static analysis to software systems for detecting security vulnerabilities and privacy leaks
  - Requirement Engineering
  - Watermarking tecniques for data and software ownership protection

SSV
Software & System Verification @ Ca' Foscari
research group

- **Ongoing projects**
  - PNRR SERICS: Static Analysis for Software Security
  - PNRR INEST: Specification and Verification of Robotic Systems
  - MUR PRIN PADS4Health: User-Centric Privacy Management

Let me introduce myself (tino cortesi)

Master of Science in Mathematics, University of Padova, 1986

PhD in Applied Math and Informatics, Univ. of Padova, 1992

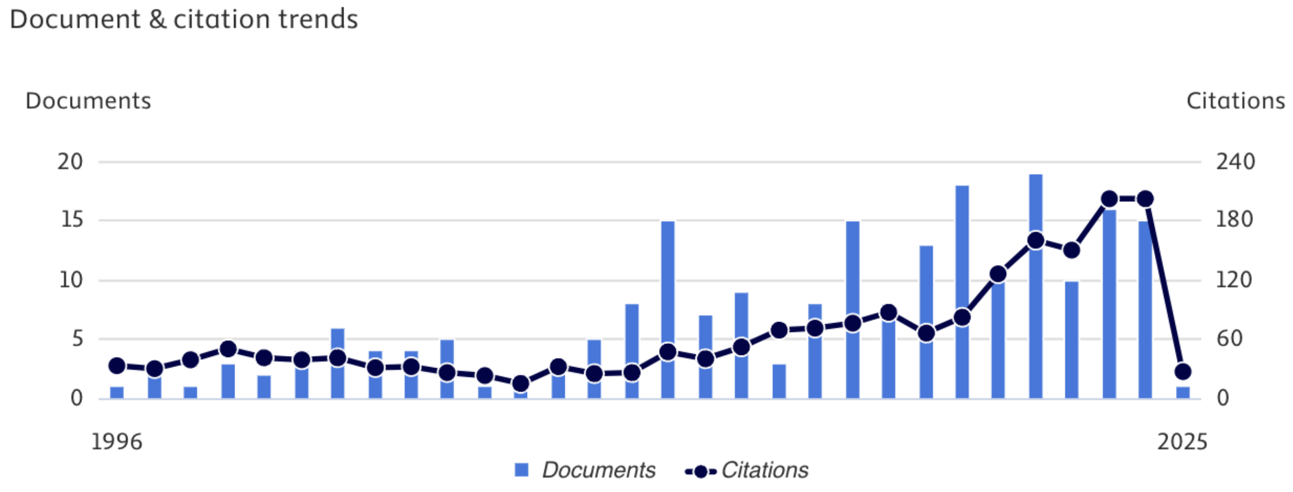
Post-doc associate, Brown University, USA, 1992/’93

Professor at Ca’ Foscari University of Venice, Italy since 1994

Research on Software Engineering and Static Analysis Techniques

More than 200 papers indexed in Scopus (www.scopus.com)

H-index: 23



A few recent publications within the SSV team

» M.Pérez Gort, **A.Cortesi**: A robust scheme for securing relational data incremental watermarking. International Journal of Information Management Data Insights, vol. 5(1) (2025)

» P.Ferrara, V.Arceri, **A.Cortesi**: Challenges of software verification: the past, the present, the future. International Journal on Software Tools for Technology Transfer, vol. 26(4), pp. 421–430 (2024)

» M.Roy, R. Bag, N.Deb, **A.Cortesi**, R.Chaki, N.Chaki: SCARS: Suturing wounds due to conflicts between non-functional requirements in autonomous and robotic systems. Software - Practice and Experience, vol. 54(5), pp. 759–795 (2024)

» S. Das, N. Deb, N. Chaki, **A.Cortesi**: Minimising conflicts among run-time non-functional requirements within DevOps. Systems Engineering 27(1): 177-198 (2024)

» S. Das, N. Deb, A.Cortesi, N. Chaki: Extracting goal models from natural language requirement specifications. Journal of Systems and Software, vol.211, 111981 (2024)

» L.Negrini, P.Ferrara, V.Arceri, **A.Cortesi**: LiSA: A Generic Framework for Multilanguage Static Analysis. Intelligent Systems Reference. Library vol.238, pp. 19–42 (2023)

» L.Olivieri, L.Negrini, V.Arceri, F.Tagliaferro, P.Ferrara, **A.Cortesi**, F.Spoto: Information Flow Analysis for Detecting Non-Determinism in Blockchain. Proc. ECOOP 2023: 23:1-23:25 (2023)

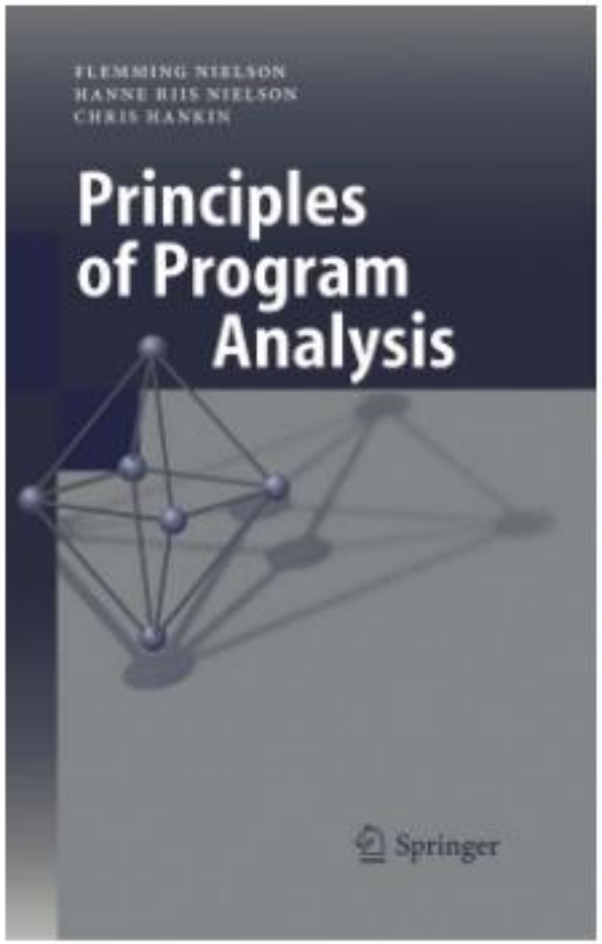
» S. Das, N. Deb, N. Chaki, **A.Cortesi**: Driving the Technology Value Stream by Analyzing App Reviews. IEEE Transactions on Software Engineering 49(7): 3753-3770 (2023)

» R. Krishnasrija, A. K. Mandal, **A. Cortes**i: A lightweight mutual and transitive authentication mechanism for IoT network. Ad Hoc Networks 138: 103003 (2023)

Keywords…

Abstract Interpretation
Static Analysis
Widening
Strings
IoT
Domains
Requirements
Vulnerability
Dataflow
Types
Mobile
Prioritization
Software Engineering
NFR
Taint
Database
DAPA
Fixpoint
Relational
Blockchain
LiSA
Logic
Formal
Privacy
Android
Java
Completeness
Narrowing
Watermarking
Robotic Software
Product Operators
Cross-Programming
Mobile Ambients
Information Leakage
Program
C

A reference book for SCSR



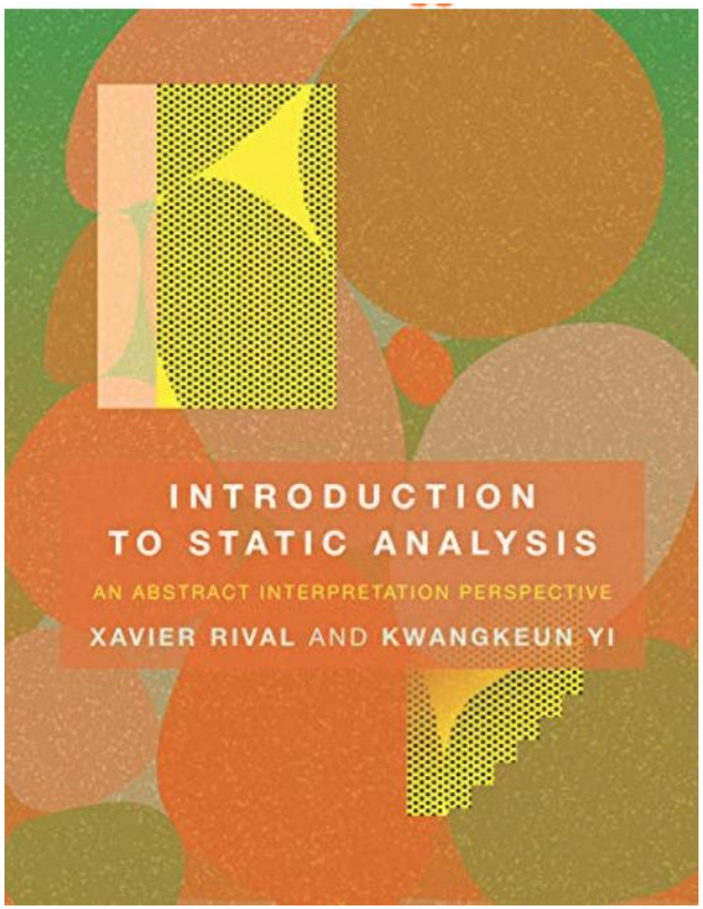
Principles of Program Analysis

By Flemming Nelson, Hanne Riis Nielson, Chris Hankin

This book is unique in providing an overview of the four major approaches to program analysis: data flow analysis, constraint-based analysis, abstract interpretation, and type and effect systems.

Springer, 2004

Another reference book


INTRODUCTION TO STATIC ANALYSIS
AN ABSTRACT INTERPRETATION PERSPECTIVE
XAVIER RIVAL AND KWANGKEUN YI

Introduction to Static Analysis

An Abstract Interpretation Perspective

By Xavier Rival and Kwangkeun Yi

A self-contained introduction to abstract interpretation–based static analysis, an essential resource for students, developers, and users.

MIT Press, 2020

and if you need more… the Bible for SCSR

Principles of Abstract Interpretation

by Patrick Cousot

Introduction to abstract interpretation, with examples of applications to the semantics, specification, verification, and static analysis of computer programs.

MIT Press, 2021

What is Software Reliability

- IEEE 610.12-1990 defines reliability as "The ability of a system or component to perform its required functions under stated conditions for a specified period of time."
- IEEE 982.1-1988 defines Software Reliability Management as "The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project constraints such as resources, schedule and performance.“
- Using these definitions, software reliability is comprised of three activities:
  - Error prevention.
  - Fault detection and removal.
  - Measurements to maximize reliability, specifically measures that support the first two activities.

Why Is Software Verification Important?

- One of the most prominent challenges for IT.
  - Software bugs cost the U.S. economy about $60 billion each year (0.6% of the GDP) .
- Security is becoming a necessity.
  - The worldwide economic loss caused by all forms of overt attacks is about $250 billion.
- Software defects make programming so painful.
- Stories
  - The Role of Software in Spacecraft Accidents (http://sunnyday.mit.edu/papers/jsr.pdf)
  - Some of the Worst Software Bugs https://raygun.com/blog/costly-software-errors-history/

Verification and validation

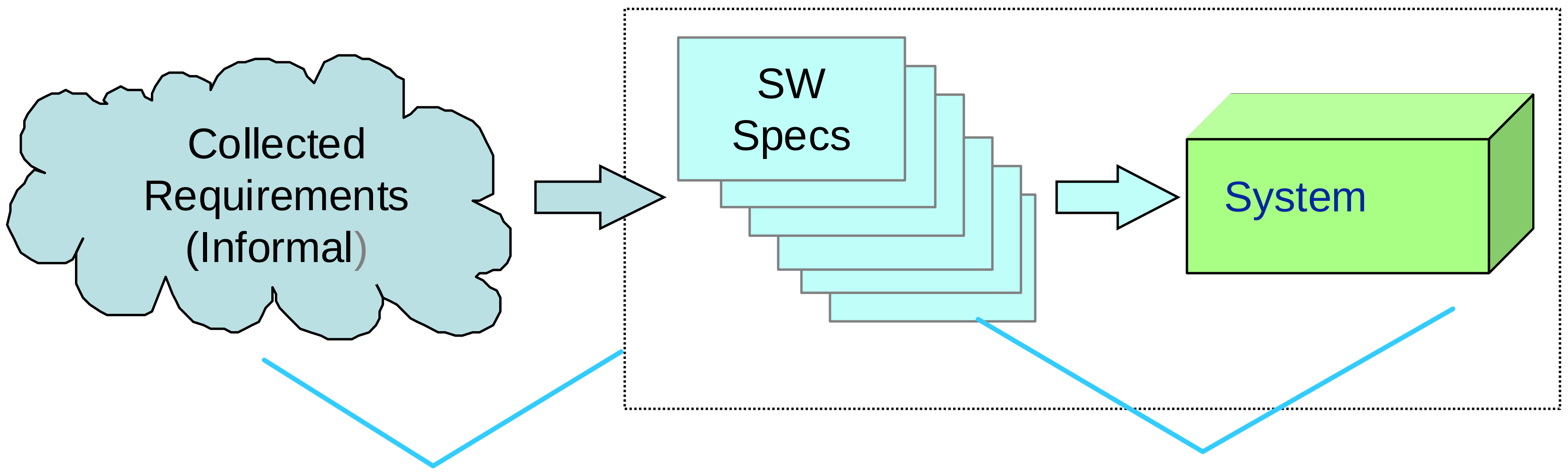
- Validation:
  does the software system meets the user's real needs?

  are we building the right software?

- Verification:
  does the software system meets the requirements specifications?

  are we building the software right?

Validation and Verification



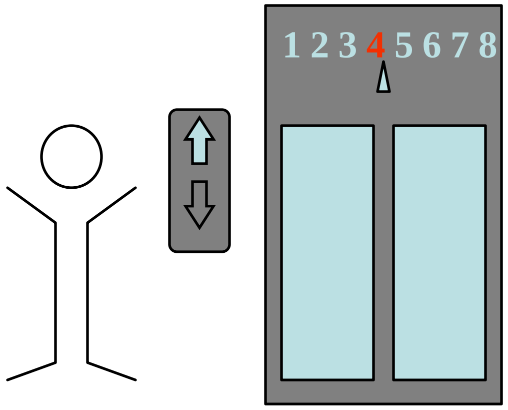
Validation

Includes usability testing, user feedback

Verification

Includes testing, inspections, static analysis

Verification versus validation



Example: elevator response

» Unverifiable (but **validatable**) specification:
if a user presses a request button at floor i, an available elevator must arrive at floor i in a reasonable time.

» **Verifiable** specification:
if a user presses a request button at floor i, an available elevator must arrive at floor i within 30 seconds...

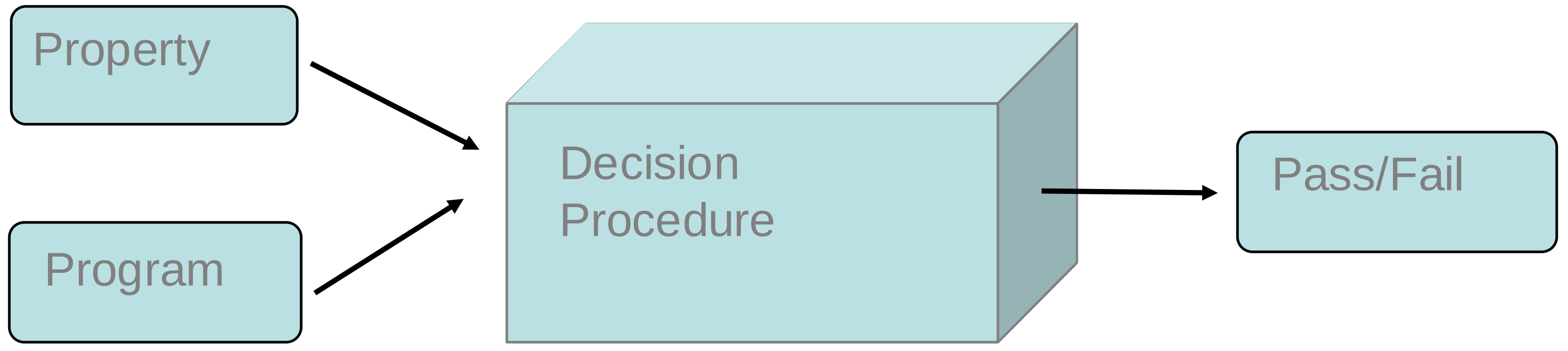
Verification

- The task is to verify some correctness statement about a program:
- The **functional correctness**, i.e. that the program offers the services described in its specification document
- The absence of **non-functional errors:**
  - The satisfaction of space / time contraints
  - The absence of run-time errors
  - The satisfaction of security constraints
- We mean verification in a strong sense: we look for **guarantees**.

Very bad news: there is no silver bullet

- Don’t expect a verification method that is:
  - **Automatic**: does not require human interaction
  - **Powerful**: able to prove non-trivial properties
  - **Sound**: never prove the property if it doesn’t hold
    never forcely claim correctness
  - **Complete**: always prove the property if it holds
    never fail to claim correctness if the program is correct
- **Rice’s Theorem** states that all non trivial properties of program behaviour in a Turing-complete programming language are undecidable.
- Undecidability of a property means that there is no automatic verification method that is sound and complete

You can't ~~always~~ *ever* get what you want



Behavioural properties are undecidable

the halting problem can be embedded in almost every property of interest

What to give up?

- Sound? We can’t give it up!!!
- Powerful? How much as possible
- Automatic:
  deductive verification and interactive theorem proving
- Completeness
  dataflow analysis, type systems, abstract interpretation

The final goal: reliable software systems

- Quality dilemma: quality / features / time
- More efficient methods for test and verification are always needed
  - No ‘silver-bullet’ when it comes to testing.
  - Formal verification is on the rise...

Testing / Formal Verification

A very crude dichotomy:

Testing
Correct with respect to the *set of test inputs*, and reference system
Easy to perform
Dynamic

In practice:

Many types of testing,

Testing / Formal Verification

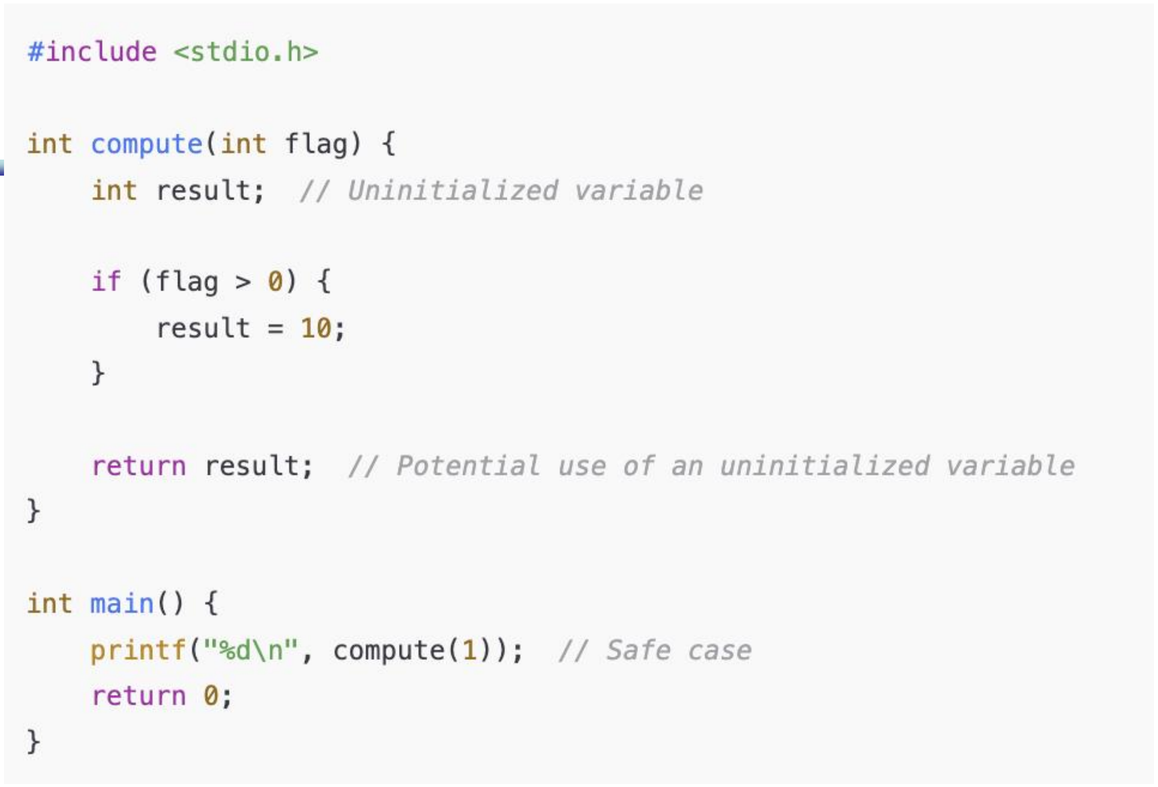
A very crude dichotomy:

Testing	Formal Verification
Correct with respect to the *set of test inputs*, and reference system	Correct with respect to *all inputs*, with respect to a *formal specification*
Easy to perform	Decidability problems, Computational problems,
Dynamic	Static

In practice:

Many types of testing,
Many types of formal verification.

Example



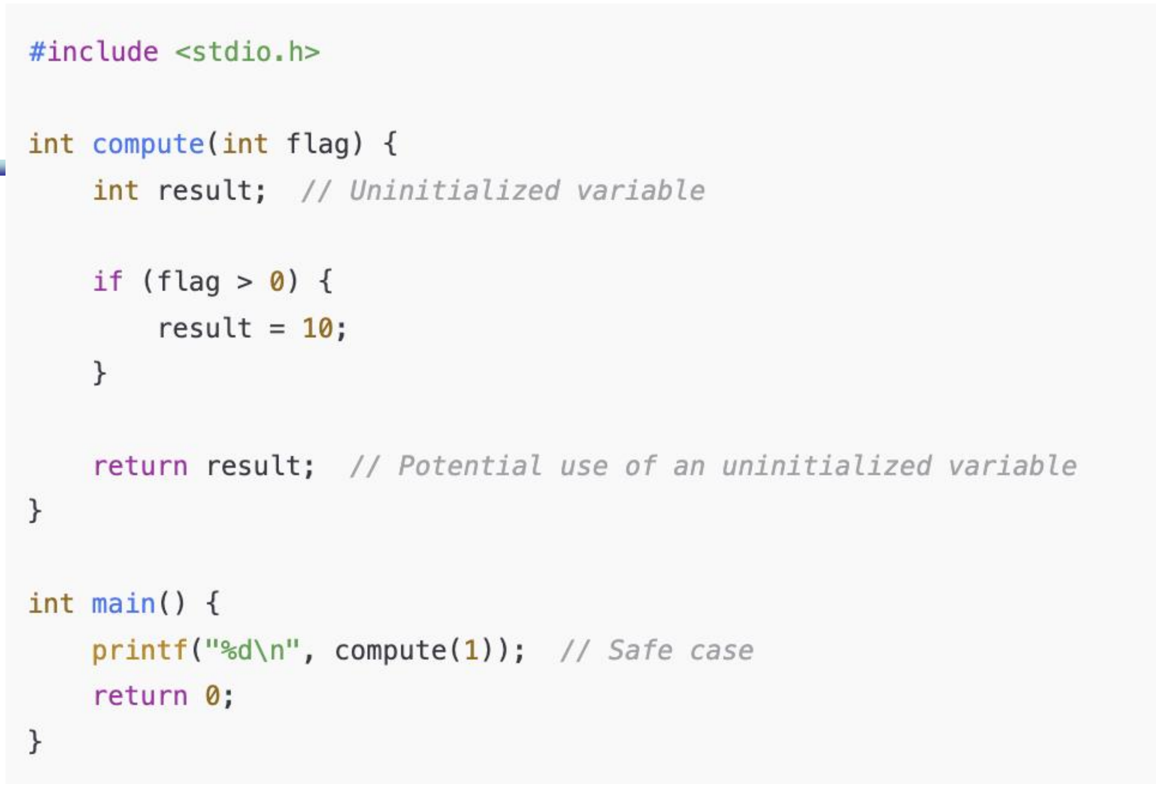
```c
#include <stdio.h>

int compute(int flag) {
    int result;  // Uninitialized variable

    if (flag > 0) {
        result = 10;
    }

    return result;  // Potential use of an uninitialized variable
}

int main() {
    printf("%d\n", compute(1));  // Safe case
    return 0;
}
```

if we run the test `compute(1)` everything seems fine, because flag > 0, so result = 10 is assigned. The test does not expose the bug.

However, if we call `compute(-1)` the function will return an uninitialized variable, leading to undefined behavior (the value of `result` is unpredictable).

If the test suite does not include this specific case, the bug remains undetected.

Example



```c
#include <stdio.h>

int compute(int flag) {
    int result;  // Uninitialized variable

    if (flag > 0) {
        result = 10;
    }

    return result;  // Potential use of an uninitialized variable
}

int main() {
    printf("%d\n", compute(1));  // Safe case
    return 0;
}
```

A static analysis tool (e.g., Clang Static Analyzer, Coverity, or GCC -Wall) detects this issue without execution by identifying that result may be used uninitialized if flag <= 0.

For example, running GCC with warnings enabled:

```
gcc -Wall -Wextra -o program program.c
```

outputs:

```
warning: ‘result’ is used uninitialized in this function
```

Our approach

Formal = based on rigorous mathematical logic concepts.

Semantics-based = based in a formal specification of the “meaning” of the program

Fundamental Limit: Undecidability

Rice Thorem

Any non-trivial semantic property of programs is undecidable.

Classical Example: Termination

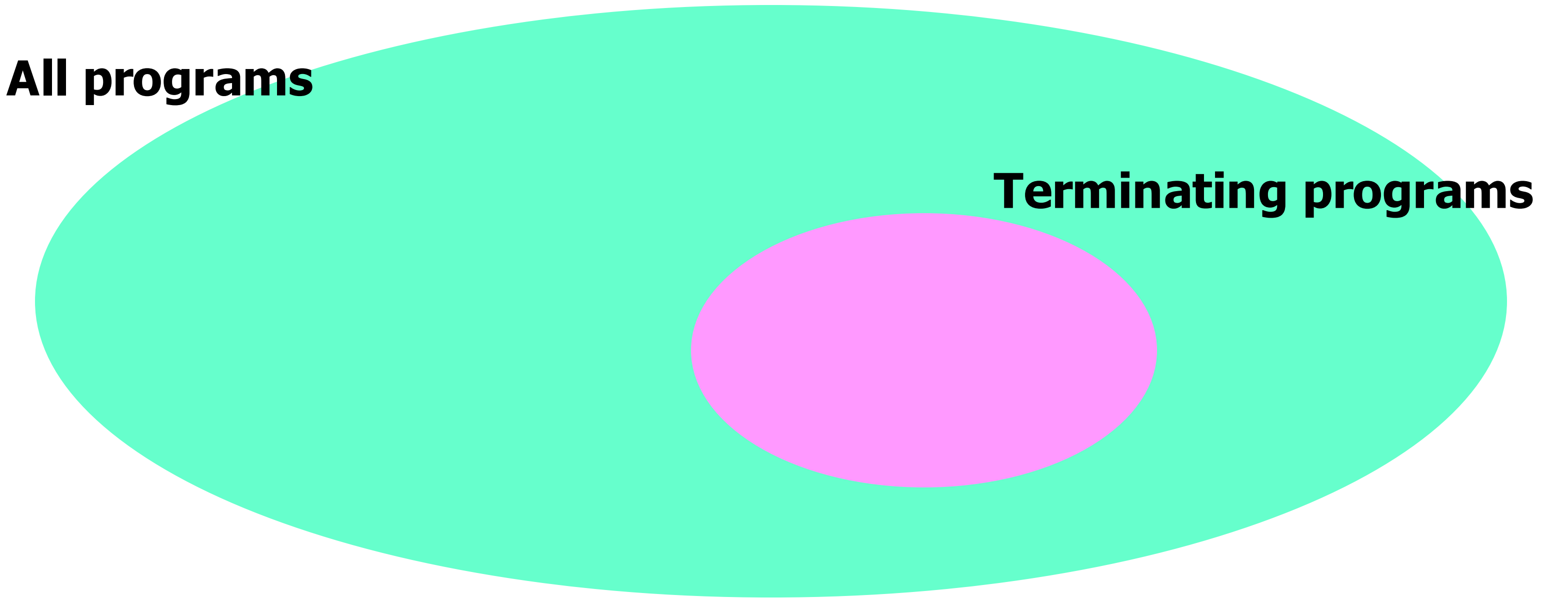
There exists no algorithm which can solve the halting problem: given a description of a program as input, decide whether the program terminates or it loops forever.

Incompleteness of Program Analysis

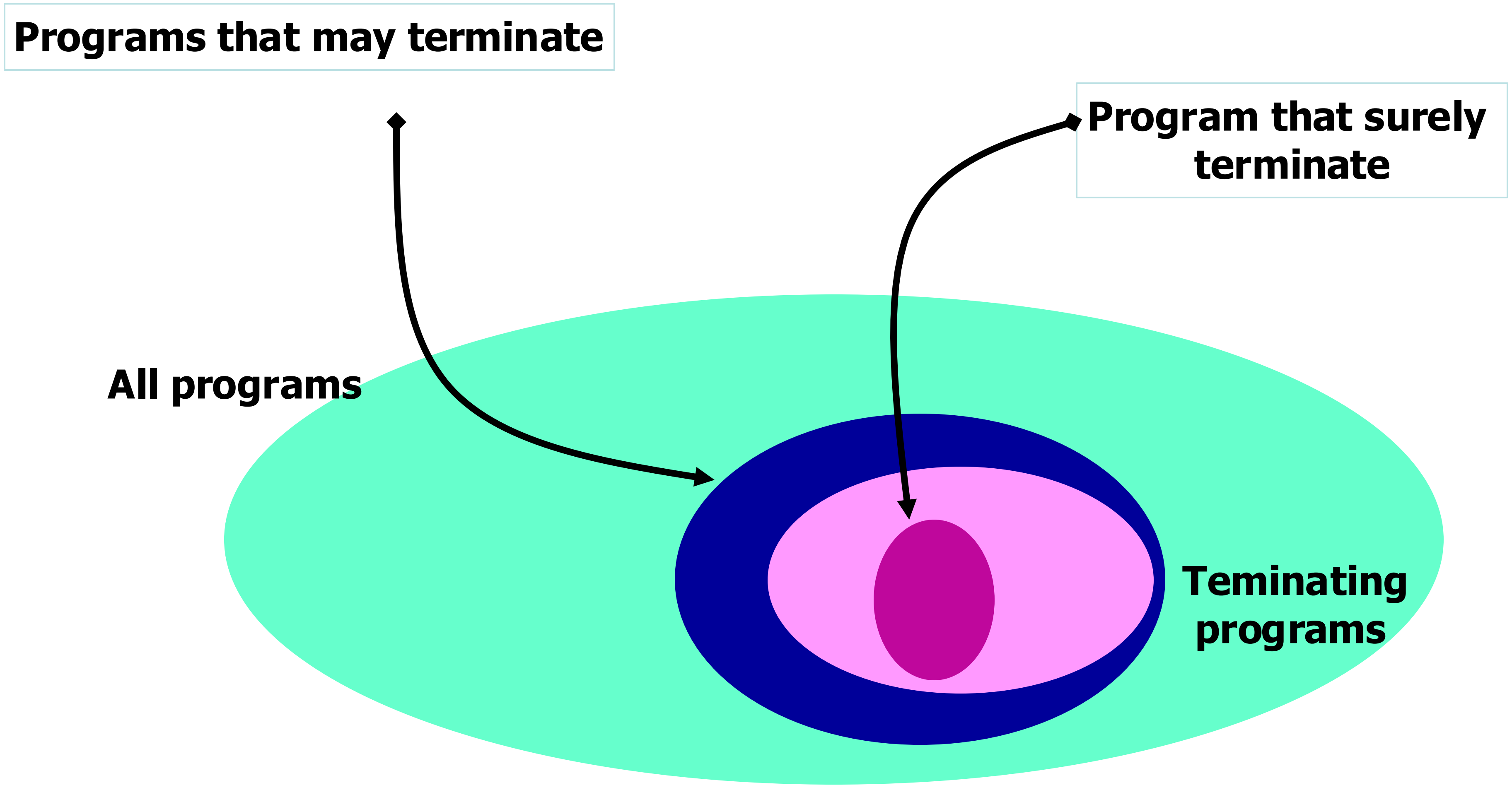
- Discovering a sufficient set of properties for checking every operation of a program is an undecidable problem!
- **False positives**:
  operations that are safe in reality but which cannot be decided safe or unsafe from the properties inferred by static analysis.

Example

- We all know that the halting problem is not decidable
- Therefore there is no program P(_) that tells us if a program Q terminates for any input.



Under- and Over-approximation



Some Terminology

- **Sound**:
  An analysis of a program P with respect to a property F is sound if the analysis returns true only when the program does satisfy the property. If the program does not satisfy the property the analysis returns either false or “don’t know”. (no false negatives)
- **Complete**:
  An analysis of a program P with respect to a property F is complete if the analysis always returns true when the program actually does satisfy the property. (no false negatives and no false positives).

Precision versus Efficiency

- Precision: number of program operations that can be decided safe or unsafe by the analyzer**.**
- Metrics: false positives, true positives, false negatives.

Precision and computational complexity are strongly related

- Tradeoff precision/efficiency: limit in the average precision and scalability of a given analyzer
- Greater precision and scalability is achieved through **specialization**

Specialization

- Tailoring the program analyzer algorithms for a specific class of programs (flight control commands, digital signal processing, etc.)
- Precision and scalability is guaranteed for this class of programs only
- Requires a lot of try-and-test to fine-tune the algorithms

Main issues

1. False Positives and False Negatives

One of the major drawbacks of static analysis is the generation of false positives and false negatives. False positives occur when the tool flags an issue that is not actually a problem, leading developers to spend unnecessary time reviewing and dismissing incorrect warnings. This can cause frustration and reduce confidence in the tool.

On the other hand, false negatives happen when the tool fails to detect real vulnerabilities or defects, potentially leaving serious bugs unnoticed until later in the development cycle. Achieving a balance between detecting real issues and minimizing noise is a fundamental challenge.

If a tool is too conservative, it may miss crucial flaws, whereas if it is too aggressive, it can overwhelm developers with unnecessary warnings.

Main issues

2. Scalability and Performance

Static analysis tools must process large and complex codebases efficiently, but as the size and complexity of software grow, the computational cost increases significantly. Advanced static analysis techniques, such as abstract interpretation or symbolic execution, can be time-consuming and resource-intensive, making them impractical for very large projects.

This issue is particularly challenging in modern continuous integration/continuous deployment (CI/CD) pipelines, where developers expect fast feedback. If the analysis takes too long, it can disrupt development workflows and reduce productivity.

Optimizing these tools to handle real-world software efficiently while maintaining accuracy remains an ongoing challenge in research and tool development.

Main issues

3. Handling Dynamic Features

Modern programming languages and frameworks heavily rely on dynamic features such as reflection, runtime code generation, dynamic typing, and polymorphism. These elements make it difficult for static analysis tools to accurately infer program behavior because much of the execution logic is determined at runtime, rather than being explicitly present in the static code.

For example, languages like JavaScript, Python, and Ruby use dynamic typing and allow developers to modify objects at runtime, making static type inference difficult. Similarly, reflection in Java or C# enables programs to inspect and modify themselves dynamically, posing a significant challenge for static analysis.

As a result, many tools struggle to provide comprehensive coverage of dynamically constructed code, leading to potential blind spots in analysis results.

Main issues

4. Usability and Developer Adoption

Even the most advanced static analysis tools are ineffective if developers do not use them consistently.

A common issue is poor usability, where tools produce complex reports with cryptic messages that require expert knowledge to interpret. If developers find the results difficult to understand or act upon, they may ignore warnings or disable the tool altogether.

Additionally, integrating static analysis into existing development workflows should be seamless—if it disrupts productivity or causes unnecessary delays, teams may be reluctant to adopt it.

To improve adoption, modern tools aim to provide clear explanations, actionable recommendations, and integration with IDEs and CI/CD pipelines, making static analysis a practical and valuable part of software development rather than an obstacle.

Course goals

- Understand basic techniques for doing program analyses and verification
  - these techniques are the cornerstone of a variety of program analysis tools
  - they may come in handy, no matter what research you end up doing
- Get a feeling for what research is like in the area by reading research papers, and getting your feet wet in current research discussions
  - useful if you don’t have a research area picked
  - also useful if you have a research area picked: seeing what research is like in other fields will give you a broader perspective

A different approach

- Most of the courses have all lectures given by a single professor, possibly with the support of a tutor
- This course will be different: you'll get lectures by members of the SSV team covering different topics
  - Tino Cortesi: Principles of Dataflow and Abstract Interpretation
  - Pietro Ferrara: Software Quality Assurance
  - Giacomo Zanatta: The LiSA static Analyser
  - Luca Olivieri: Smart contracts analysis

A different approach

- » In most of the courses you first get into the nightmare of a theory, and only at the end (possibly) you see how and where it results useful to solve problems
- » This course will be different: problems come first – theory comes after, as a way to find suitable solutions.
- » (Almost) every week an assignment will be given.
  - – The deadlines are fixed & strict!
  - – No plagiarism, please!
  - – The tasks should be uploaded in github (details next week)
- » The assignment will be discussed the following week, and will be the basis for the next step

Course topics

- Techniques for representing programs & properties
- Techniques for analyzing and verifying programs
- Applications of these techniques in the context of a software quality assurance process
- Getting familiar with some industrial static analysers

Course topics (more details)

- Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Dataflow Graph
  - Static Single Assignment
  - Control Dependence Graph
  - Program Dependence Graph
  - Call Graph

Course topics (more details)

- AnalysisTechniques
  - Abstract interpretation
  - Dataflow Analysis
  - Interprocedural analysis

  - Interaction between transformations and analyses
  - Maintaining the program representation intact

Course pre-requisites

- No compilers background necessary
- Some familiarity with algebraic structures
  - I will review what is necessary in class, but it helps if you know it already
- Some familiarity with program semantics
  - we will focus on semantics-based approaches
- A standard computer science curriculum will most likely cover the above
  - Talk to me if you think you don’t have the pre-requisites

To pass the exam...

Type A

A final project work will be assigned:

- » Gain confidence with a static analyser
- » Perform the analysis on simple imperative software code
- » Design and implement a few simple new features

Type B (for those who decide to do their master thesis in SCSR)

- » The design and development of a new branch of LiSA

Evaluation

- The final evaluation will be based on an oral discussion on tasks and final project work:
  - Quality of the tasks (50%)
  - Correctness and completeness of the final project work (50%)

Questions?