Course Introduction

Week 1

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

COMP6131 Inaugural Offering at UNSW

Welcome to the inaugural offering of COMP6131 at UNSW!

COMP6131 Inaugural Offering at UNSW

Welcome to the inaugural offering of COMP6131 at UNSW!

- Pre-course survey: `https://forms.gle/r1VLFV8pPmdosPFe7`

COMP6131 Inaugural Offering at UNSW

Welcome to the inaugural offering of COMP6131 at UNSW!

- Pre-course survey: `https://forms.gle/r1VLFV8pPmdosPFe7`
- Give me some key words about your understanding of this course.

COMP6131 Inaugural Offering at UNSW

Welcome to the inaugural offering of COMP6131 at UNSW!

- Pre-course survey: `https://forms.gle/r1VLFV8pPmdosPFe7`
- Give me some key words about your understanding of this course.
- Give me some courses that you think are related.

COMP6131 Inaugural Offering at UNSW

Welcome to the inaugural offering of COMP6131 at UNSW!

- Pre-course survey: `https://forms.gle/r1VLFV8pPmdosPFe7`
- Give me some key words about your understanding of this course.
- Give me some courses that you think are related.
    - System and Software Security Assessment (COMP6447)
    - Security Engineering and Cyber Security (COMP6441/COMP6841)
    - Programming Languages and Compilers (COMP3131/COMP9102)
    - Advanced C++ Programming (COMP6771)
    - Algorithmic Verification (COMP3153/COMP9153)

COMP6131 Inaugural Offering at UNSW

Welcome to the inaugural offering of COMP6131 at UNSW!

- Pre-course survey: `https://forms.gle/r1VLFV8pPmdosPFe7`
- Give me some key words about your understanding of this course.
- Give me some courses that you think are related.
  - System and Software Security Assessment (COMP6447)
  - Security Engineering and Cyber Security (COMP6441/COMP6841)
  - Programming Languages and Compilers (COMP3131/COMP9102)
  - Advanced C++ Programming (COMP6771)
  - Algorithmic Verification (COMP3153/COMP9153)

Your active participation in and off-class discussions, as well as your feedback, will be invaluable. Hope to make your learning experience an enjoyable and rewarding one.

Administration and Important Course Links

- Course convenor and lecturer: A/Prof. Yulei Sui
- Email: cs6131@cse.unsw.edu.au
- Course webpage: `https://webcms3.cse.unsw.edu.au/COMP6131/24T2`
- Lab-Exercise/Assignment specifications, and code templates: `https://github.com/SVF-tools/Software-Security-Analysis`
- Course ED forums: `https://edstem.org/au/join/7tPnP2`
- Important messages will be announced on the course homepage or via email.
- Course admin:
  - Xiao Cheng (xiao.cheng@unsw.edu.au)
- Lab Demonstrator:
  - Kaiqi Liang (kaiqi.liang@unsw.edu.au)
- Course keywords: Static Analysis and Verification, Security Vulnerabilities, Control- and Data-Flows, Symbolic Execution, Abstract Interpretation

Lectures and Labs

- Lecture
    - Time: 11:00 - 13:00, Friday
    - Location: Old Main Building **G31 (K-K15-G31)**
- Lab
    - Time: 13:00 - 15:00, Friday
    - Location: Old Main Building **G32 (K-K15-G32)**
- Consultation (appointment is preferred)
    - Time: 15:00 - 16:00, Friday
    - Location: 501H, K17

Course Aim

In this course, you will learn to create **automated code analysis and verification tools** using a **modern compiler** and an **open-source** static analysis framework, to perform **code comprehension**, **vulnerability detection** and code **verification** in real-world software systems.

Teaching Strategy and Rationale

This course has three major components:

- **Lectures** (10 weeks excluding Week 6 for study break)
- **Labs** (10 weeks excluding Week 6 for study break)
- **Assignments** (Assignments 1-3)

This is a project-based course and you are expected to produce a tool towards the end of the course and **NO paper examination** is required!

Teaching Strategy and Rationale

This course has three major components:

- **Lectures** (10 weeks excluding Week 6 for study break)
- **Labs** (10 weeks excluding Week 6 for study break)
- **Assignments** (Assignments 1-3)

This is a project-based course and you are expected to produce a tool towards the end of the course and **NO paper examination** is required!

Assessment Type	Name	Percentage %
Lab work	Quiz-1 & Exercise-1	10%
	Quiz-2 & Exercise-3	10%
	Quiz-3 & Exercise-3	10%
Assignment-1	Information flow tracking	20%
Assignment-2	Symbolic execution	25%
Assignment-3	Abstract interpretation	25%

Lectures

Course contents:

- Foundational **theories of static code analysis and verification** aimed at detecting bugs and verifying the absence of bugs.
- Some practical **demonstrations of examples and coding**
- **Problem-solving skills** (algorithms, testing, debugging)
- Lecture slides typically available before each lecture. Lectures are recorded.

Lectures

Course contents:

- Foundational **theories of static code analysis and verification** aimed at detecting bugs and verifying the absence of bugs.
- Some practical **demonstrations of examples and coding**
- **Problem-solving skills** (algorithms, testing, debugging)
- Lecture slides typically available before each lecture. Lectures are recorded.

Get the most out of COMP6131:

- **Attend the lectures/labs and get involved!** Ask questions in class and on Ed forums (no pasting code solutions allowed).
- **Be open-minded.** While you will use C++ to implement your code checker for analyzing C programs in this course. Consider developing a code checker using the learned theories within a modern compiler setting.
- **Research and development mentality.** Keep your curiosity to learn the most recent/advanced source code analysis techniques in this course.

Labs

Labs: Hands-on experience includes **preparatory activities** and building the **skills needed for assignments** through completing. Labs are typically not recorded and we may try to record some demonstrations.

- three set of **quizzes** (multiple choice questions)
- three **coding exercises** (small-scale)

Labs

Labs: Hands-on experience includes **preparatory activities** and building the **skills needed for assignments** through completing. Labs are typically not recorded and we may try to record some demonstrations.

- three set of **quizzes** (multiple choice questions)
- three **coding exercises** (small-scale)

Submission and marking

- Done **individually**
- Submitted a single cpp file in each lab exercise by uploading to WebCMS or via `give`.
- Automarked (with manual checks and partial marking) against our internal tests.

Assignments

Three assignments: Each assignment is built on the previous one to develop code-checking tools capable of:

- Assignment-1: tracking tainted information flows
- Assignment-2: performing symbolic execution
- Assignment-3: conducting abstract interpretation

Assignments

Three assignments: Each assignment is built on the previous one to develop code-checking tools capable of:

- Assignment-1: tracking tainted information flows
- Assignment-2: performing symbolic execution
- Assignment-3: conducting abstract interpretation

Best practice for completing an assignment (e.g., Assignment-1)?

- **Correct way**: Lab-Quiz-1 → Lab-Exercise-1 → Assignment-1
- **Incorrect way**: all other orders

Assumed Knowledge

- Should have
  - Experience in writing, debugging, and testing programs in C (COMP2521, COMP9024).
  - Knowledge of using Git and programming IDEs like vim or VSCode.
  - Willingness to learn and open-mindedness.
- Nice to have
  - Some knowledge of object-oriented programming (you will get more practice in C++ programming in our labs).
  - Some background knowledge of compilers (e.g., LLVM and SVF).
  - Some knowledge of secure coding.
  - Experience at different programming ”levels” (e.g. low-level, high-level).

Learning Outcomes

- Practice **system programming skills** to develop **code analysis and verification techniques** to address code security and reliability problems.
  - **High-quality coding**: Commit to writing high-quality, error-free, and high-performance code, especially within the context of large-scale codebases.
  - **Compiler basics**: Gain insights into compilation, code representation, low-level instructions, code debugging and profiling.
  - **Vulnerability Assessment**: Understand common vulnerabilities, such as tainted information flow, buffer overflows, and assertion errors.
  - **Open-source static analysis framework**: learn to build practical tools on top of open-source frameworks like SVF.
  - **Formal Verification**: Understand formal methods and techniques for verifying code correctness using mathematical and logical reasoning tools.

Learning Outcomes

- Practice **system programming skills** to develop **code analysis and verification techniques** to address code security and reliability problems.
  - **High-quality coding**: Commit to writing high-quality, error-free, and high-performance code, especially within the context of large-scale codebases.
  - **Compiler basics**: Gain insights into compilation, code representation, low-level instructions, code debugging and profiling.
  - **Vulnerability Assessment**: Understand common vulnerabilities, such as tainted information flow, buffer overflows, and assertion errors.
  - **Open-source static analysis framework**: learn to build practical tools on top of open-source frameworks like SVF.
  - **Formal Verification**: Understand formal methods and techniques for verifying code correctness using mathematical and logical reasoning tools.
- Career and job roles
  - Software Engineer; Security Analyst/Engineer; Compiler Engineer; Formal Methods Engineer; Software Reliability Engineer; Embedded Systems Developer; Research Scientist (in Academia or Industry);

Course Schedule

Week	Content	Lab & Assignment Start	Due (23:59, Wednesday)
1	**Lecture**: Course Overview and Introduction **Lab**: C++ practices, vulnerability assessment and compiler IR	Quiz-1 + Exercise-1 (10%)	-
2	**Lecture**: Control and Data Flows **Lab**: Code graphs, SVF, constraints solving	Assignment 1 (20%)	-
3	**Lecture**: Pointer Aliasing and Taint Tracking **Lab**: Tainted information flow tracking	-	Quiz-1 + Exercise-1
4	**Lecture**: Code Verification Basis **Lab**: Verification concepts, predicate logic	Quiz-2 + Exercise-2 (10%)	Assignment-1
5	**Lecture**: Automated Theorem Proving **Lab**: Manual assertion-based verification using Z3	Assignment 2 (25%)	-
6	**Flexibility Week**	-	-
7	**Lecture**: Code Verification using Symbolic Execution **Lab**: Automated code assertion verification using Z3	-	Quiz-2 + Exercise-2
8	**Lecture**: Abstract Interpretation Foundations **Lab**: Basic concepts and examples	Quiz-3 + Exercise-3 (10%)	Assignment-2
9	**Lecture**: Code Verification using Abstract Interpretation **Lab**: Manual assertion-based verification using Z3	Assignment 3 (25%)	-
10	**Lecture**: Buffer Overflow Detection using Abstract Interpretation **Lab**: Implementation and testing	-	Quiz-3 + Exercise-3 Assignment-3 (Week 11)

Assessment Guidelines

- **Joint Work Prohibited**: Collaboration on this assignment is not allowed. Each quiz, exercise, and assignment is submitted and marked individually.
- **Individual Submission**: The work you submit must be entirely your own. Submitting any work, even partially, written by someone else is prohibited.
- **Assessment Marking**: Submissions will be examined both automatically and manually for external authorship.
- **Prohibition on Sharing Work**: Sharing, publishing, or distributing your assignment is not allowed even after the course ends. Do not share your work with anyone other than the COMP6131 teaching staff. **Do not publish your lab or assignment code online (e.g., on a public GitHub repository)**, as they may be used by future students.

Violation of these conditions may result in an academic integrity investigation. For more information, read the UNSW Student Code.

Marking and Plagiarism

- Please refer to specs for each assessment before you start at WebCMS
- No extension is allowed and late submission is strongly discouraged.

Marking and Plagiarism

- Please refer to specs for each assessment before you start at WebCMS
- No extension is allowed and late submission is strongly discouraged.
- The UNSW standard late penalty for assessment is 5% per day for 5 days - this is implemented hourly for this assignment. Your assignment mark will be reduced by 0.2% for each hour (or part thereof) late past the submission deadline.
    - For example, if an assignment worth 60% was submitted half an hour late, it would be awarded 59.8%, whereas if it was submitted past 10 hours late, it would be awarded 57.8%.
- Beware - submissions 5 or more days late will receive zero marks. This again is the UNSW standard assessment policy.
- The marking results will normally be released one week after each submission deadline.

Plagiarism: see course outline for penalties

Course Materials and Resources

No single textbook covers all the course content. Recommended references and an abundance of online materials are available below:

- Static Value-Flow Analysis Framework for Source Code
  - `https://github.com/SVF-tools/Teaching-Software-Security-Analysis`
  - `https://github.com/SVF-tools/SVF`
- Compilers: Principles, Techniques, and Tools Hardcover, `https://www.amazon.com.au/Compilers-Alfred-V-Aho/dp/0321486811`
- LLVM Compiler `https://llvm.org/`
- Symbolic Execution `https://en.wikipedia.org/wiki/Symbolic_execution`
- Abstract Interpretation `https://en.wikipedia.org/wiki/Abstract_interpretation`
- Z3 Theorem Prover
  - `https://github.com/Z3Prover/z3`
  - `https://theory.stanford.edu/~nikolaj/programmingz3.html`

Introduction to Software Security Analysis

(Week 1)

Yulei Sui

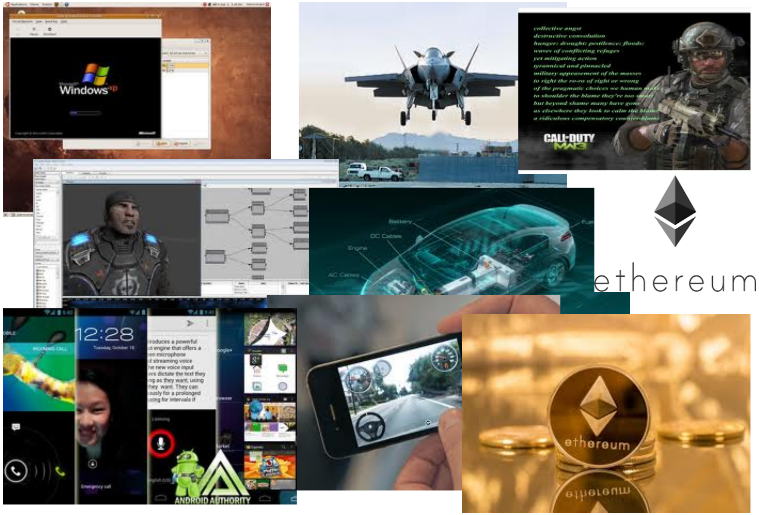
School of Computer Science and Engineering

University of New South Wales, Australia

Outline

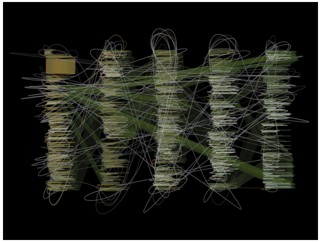
- Background and Introduction to Software Analysis and Verification
- Course Project Structure (Labs and Assignments)
- Vulnerability Assessment and Secure Coding.

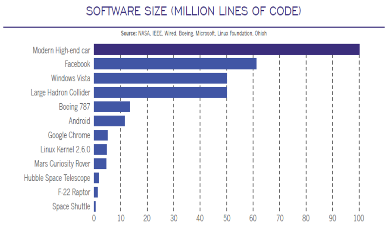
Software Is Everywhere

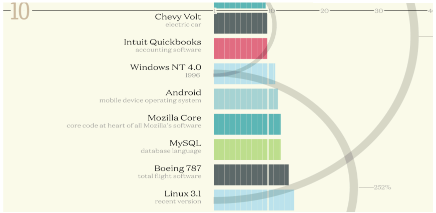


Modern System Software

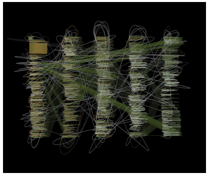
– Extremely Large and Complex

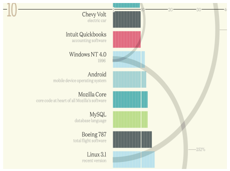




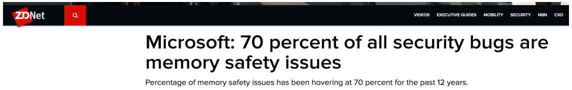


Software Becomes More Buggy





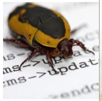
More Complex!



Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

```
A problem has been detected and Windows has been shut down to prevent damage
to your computer.

IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x0000000A (0x0227001d, 0x00000002, 0x00000000, 0x804eba3a)

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further
assistance.
```



Memory Leaks

Buffer Overflows

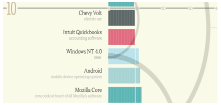
Null Pointers

Use-After-Frees

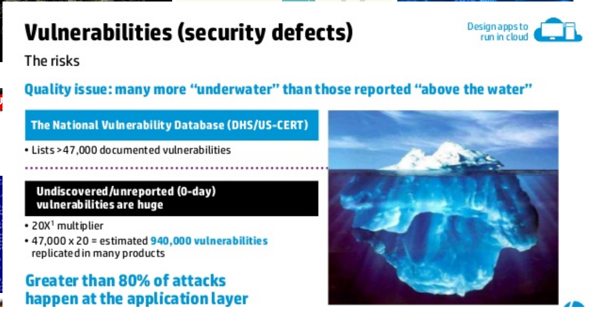
Data-races

More Buggy!

Software Becomes More Buggy



More Complex!



Vulnerabilities (security defects)

Design apps to run in cloud

The risks

Quality issue: many more "underwater" than those reported "above the water"

The National Vulnerability Database (DHS/US-CERT)

- Lists >47,000 documented vulnerabilities

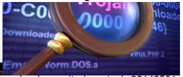
Undiscovered/unreported (0-day) vulnerabilities are huge

- 20X[1] multiplier
- 47,000 x 20 = estimated **940,000 vulnerabilities** replicated in many products

Greater than 80% of attacks happen at the application layer

Public vulnerabilities are tip of the iceberg !

More Buggy!



Data-races

https://www.slideshare.net/innotech_conference/hp-cloud-security-inno-tech-20140501

Code Review by Developers

However ...

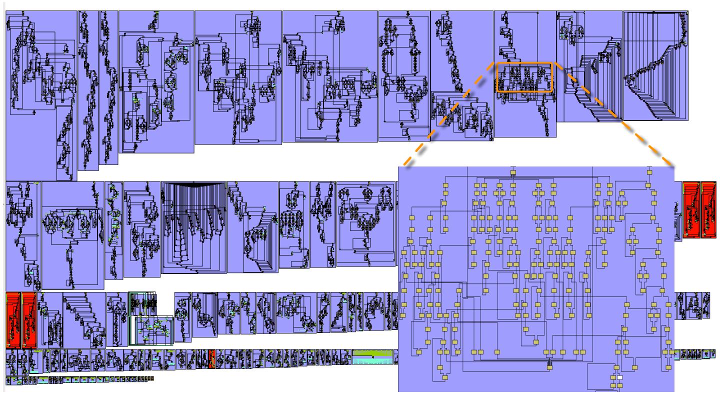
incomplete debug report

A large project (e.g., consists of millions of lines of code) is almost impossible to be manually checked by human :

-- intractable due to potentially unbounded number of paths that must be analyze
-- undecidable in the presence of dynamically allocated memory and recursive data structures

How about real-world large programs?

Whole-Program CFG of `twolf` (20.5K lines of code)

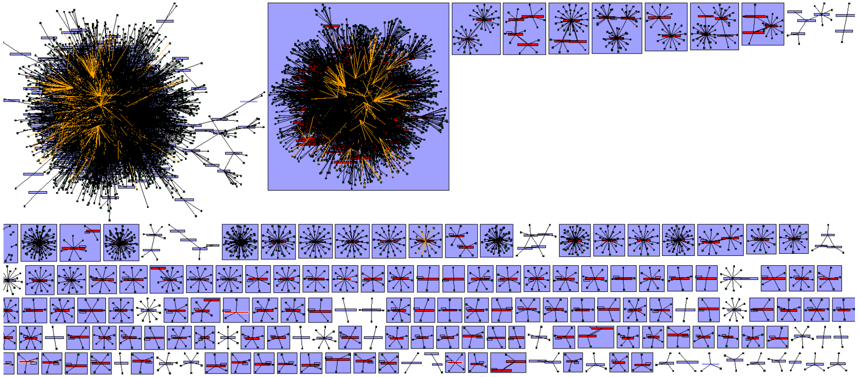


#functions: 194

#pointers: 20773   #loads/stores: 8657 Costly to reason about flow of values on CFGs!

How about real-world large programs?

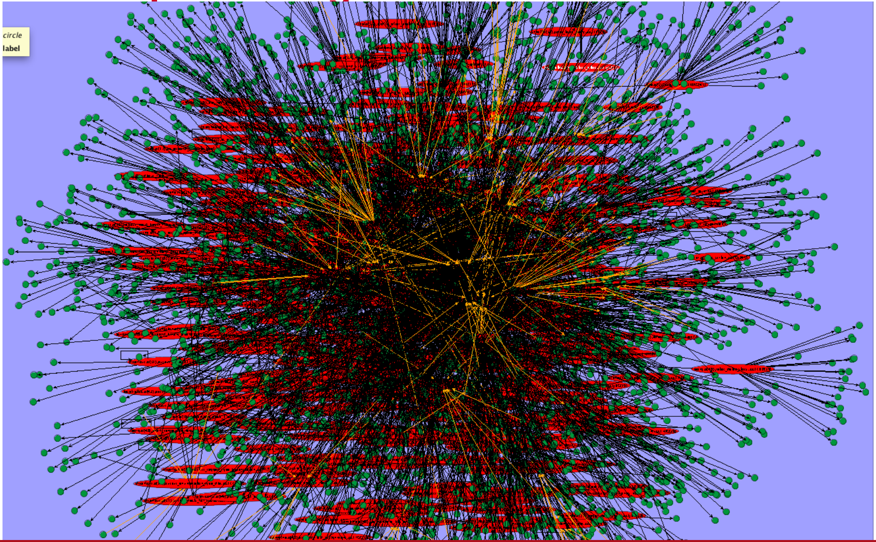
Call Graph of `gcc` (230.5K lines of code)



#functions: 2256 #pointers: 134380 #loads/stores: 51543

Costly to reason about flow of values on CFGs!

Call Graph of 176.gcc



Automated Code Analysis and Verification

Automatically analyzing and assuring the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.

- **Software Analysis** (Week 1-3, Assignment 1)
  - Aim to ***find existence of bugs*** (If there exist a path, a bug can/may be triggered)
- **Software Verification** (Week 4-5,7-10, Assignments 2 and 3)
  - Aim to ***prove absence of bugs*** (For all paths, user specification should be satisfied and no bug should be triggered)

Automated Code Analysis and Verification

- Software analysis and verification are useful for proving the correctness, safety and security of a program, and a key aspect of testing can execute as expected.
    - ”Have we made what we were trying to make?”
        - Are we building the system right?
        - Does our design meet the user expectations?
        - Does the implementation conform to the specifications?

Why Software Analysis and Verification?

- Better quality in terms of more secure and reliable software
  - Help reduce the chances of system failures and crashes
  - Cut down the number of defects found during the later stages of development
  - Rule out the existence of any backdoor vulnerability to bypass a program's authentication
- Reduce time to market
  - Less time for debugging.
  - Less time for later phase testing and bug fixing
- Consistent with user expectations/specifications
  - Assist the team in developing a software product that conforms to the specified requirements
  - Help get a better understanding of (legacy) parts of a software product

What Types of Analysis/Verification We Have?

Code verification vs design verification

- **Design approach**: analyzing and verifying the design of a software system.
  - Design specs: specification languages for components of a system. For example,
    - Z language for business requirements,
    - Promela for Communicating Sequential Processes
    - B method based on Abstract Machine Notation.
    - Specification Language (VDM-SL)
    - . . .

What Types of Analysis/Verification We Have?

Code verification vs design verification

- **Design approach**: analyzing and verifying the design of a software system.
  - Design specs: specification languages for components of a system. For example,
    - Z language for business requirements,
    - Promela for Communicating Sequential Processes
    - B method based on Abstract Machine Notation.
    - Specification Language (VDM-SL)
    - . . .
- **Code approach**: verifying correctness of source code (**This course**)
  - Code specs (e.g., return a sorted list and free of memory errors):
    - Assertions and pre/postconditions in Hoare logic (design by contract)
    - Passing user-provided test cases
    - No crashes and free of memory errors
    - . . .

How to Perform Code Analysis and Verification?

How to Perform Code Analysis and Verification?

Prove or disprove the correctness of your code against the specifications via

- **Dynamic approach** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - . . .

How to Perform Code Analysis and Verification?

Prove or disprove the correctness of your code against the specifications via

- **Dynamic approach** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - . . .
- **Static approach** (inspecting the code before it runs) (**This course**)
  - **All path verification** which aims to prove that a program satisfies the specification of its behavior by reasoning about all possible program paths
    - **Control- and Data-flow analysis** (computing control- and data-dependence of a program on code graphs to collect reachability properties) - **Assignment 1**

How to Perform Code Analysis and Verification?

Prove or disprove the correctness of your code against the specifications via

- **Dynamic approach** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - ...
- **Static approach** (inspecting the code before it runs) (**This course**)
  - **All path verification** which aims to prove that a program satisfies the specification of its behavior by reasoning about all possible program paths
    - **Control- and Data-flow analysis** (computing control- and data-dependence of a program on code graphs to collect reachability properties) - **Assignment 1**
    - **Symbolic execution** (a practical way to use symbolic expressions instead of concrete values to explore the possible program paths) - **Assignment 2**

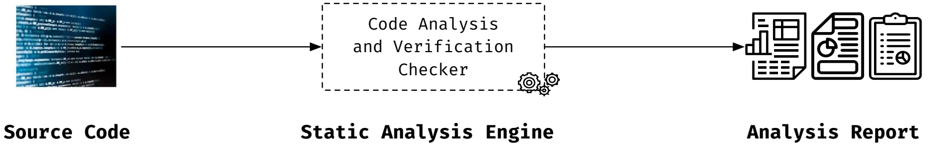
How to Perform Code Analysis and Verification?

Prove or disprove the correctness of your code against the specifications via

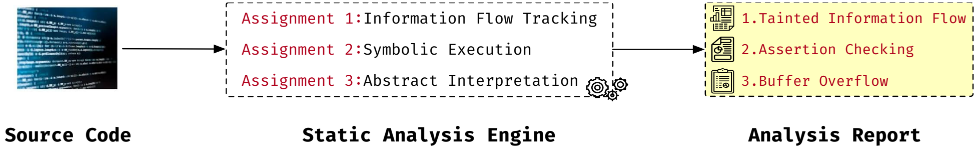
- **Dynamic approach** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - ...
- **Static approach** (inspecting the code before it runs) (**This course**)
  - **All path verification** which aims to prove that a program satisfies the specification of its behavior by reasoning about all possible program paths
    - **Control- and Data-flow analysis** (computing control- and data-dependence of a program on code graphs to collect reachability properties) - **Assignment 1**
    - **Symbolic execution** (a practical way to use symbolic expressions instead of concrete values to explore the possible program paths) - **Assignment 2**
    - **Abstract interpretation** (a general theory of sound approximation of a program through program abstractions or abstract values) - **Assignment 3**

The Project of This Course

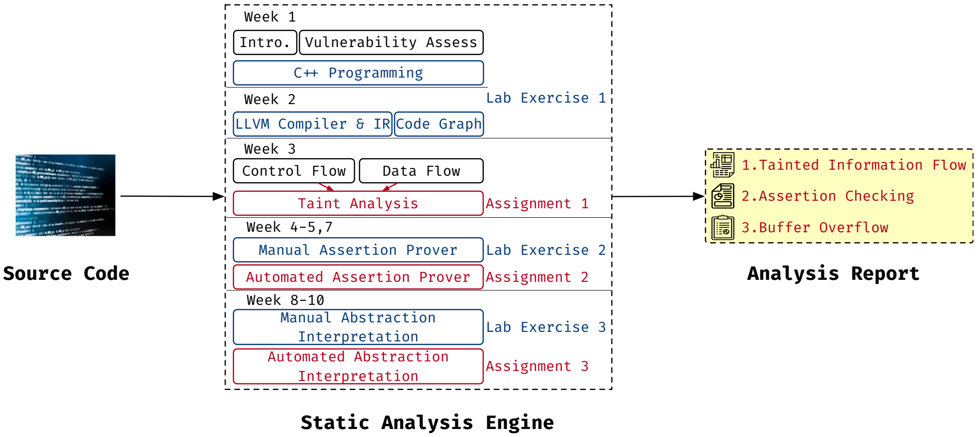
Goal of this course: develop your own software verification tool in 10-week time.
More concretely: develop a static analysis engine in C++ to analyze and verify C programs for bug detection at compile time.



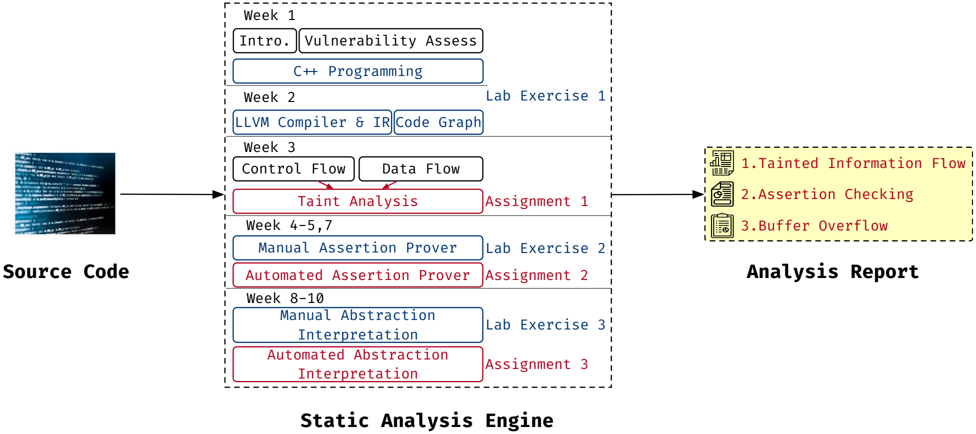
The Project of This Course


Assignment 1:Information Flow Tracking
Assignment 2:Symbolic Execution
Assignment 3:Abstract Interpretation
1.Tainted Information Flow
2.Assertion Checking
3.Buffer Overflow
Source Code
Static Analysis Engine
Analysis Report

The Project of This Course

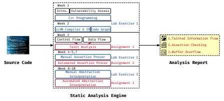


The Project of This Course

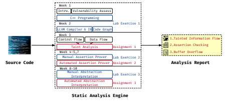


The Project of This Course

The project sounds complicated?



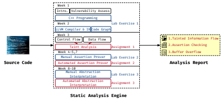
The Project of This Course



The project sounds complicated?

- Do I need to implement it from scratch?

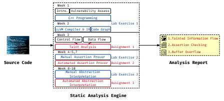
The Project of This Course



The project sounds complicated?

- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (`https://github.com/SVF-tools/SVF`)
  - **SVF**, an impactful code analysis framework developed and maintained by UNSW for 10+ years (ICSE, OOPSLA and SAS Distinguished Paper awards).
- How many lines of code do I need to write?

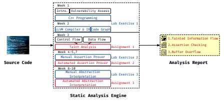
The Project of This Course



The project sounds complicated?

- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (`https://github.com/SVF-tools/SVF`)
  - **SVF**, an impactful code analysis framework developed and maintained by UNSW for 10+ years (ICSE, OOPSLA and SAS Distinguished Paper awards).
- How many lines of code do I need to write?
  - **2,000 lines** of core code **in total** for all the assessments
- Really? What are the challenges then?

The Project of This Course



The project sounds complicated?

- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (`https://github.com/SVF-tools/SVF`)
  - **SVF**, an impactful code analysis framework developed and maintained by UNSW for 10+ years (ICSE, OOPSLA and SAS Distinguished Paper awards).
- How many lines of code do I need to write?
  - **2,000 lines** of core code **in total** for all the assessments
- Really? What are the challenges then?
  - Good programming and debugging skills.
  - Understanding of basic compiler principles,
  - Knowledge of taint analysis, symbolic execution, and abstract interpretation.
  - **Please do attend each lecture and lab** to make sure you can keep up!