Program Analysis
(CSCI-4450/CSCI-6450)
Spring 2022

www.cs.rpi.edu/~milanova/csci4450/

Ana Milanova
Office: https://rensselaer.webex.com/meet/milana2
Email: milanova@cs.rpi.edu
Office hours: Mondays and Thursdays 4-5PM, or by appointment
Outline

- Logistics
  - www.cs.rpi.edu/~milanova/csci4450/
- Program analysis, introduction
- Course topics, tools and homework
- Introduction to Dataflow Analysis
Logistics

- Course webpage
  - http://www.cs.rpi.edu/~milanova/csci4450

- Schedule, Notes, Reading
  - Schedule, lecture slides and assigned reading

- Submitty
  - All homework submission and grades, forum
  - Check forum regularly for announcements
Logistics

Recommended reading

- **Types and Programming Languages**, by Benjamin C. Pierce.
- MIT’s Open Courseware Class on Program Analysis
- **Principles of Program Analysis** by Flemming Nielson, Hanne Riis Nielson, and Chris Hankin.

Papers and lecture notes
Logistics

Syllabus

www.cs.rpi.edu/~milanova/csci4450/syllabus.htm

Topics, outcomes, policies, and grading

- Take-home final: 25%
- Homework: 40%
- Paper presentation: 10%
- In-class quizzes (6-8): 20%
- Attendance and participation: 5%
Logistics

- Assignments are to be completed individually unless otherwise specified.
- Quizzes are in-class, open-notes, and may be completed individually or in small groups.
Late Homework

- Homework assignments must be submitted in Submitty by 12pm on the due date.
- You have 6 late days for the semester, with a max of 2 late days per assignment.
- Exceptions to policy granted may be granted.
Program Analysis

- Tools and techniques that help us reason about the run-time behavior of the program
  - Dynamic analysis – **during** program execution
    - Static instrumentation
    - Dynamic (binary) instrumentation (DBI)
  - **Static analysis** – **before** program execution
    - E.g., Java compiler’s definite-assignment-check
    - E.g., Type checking and type inference are forms of static analysis
    - E.g., Dafny-style verification
    - And many, many more!
E.g., is there uninitialized memory?

- char buf[64]; -> definition-free path -> use of buf
- or char * buf = malloc(64); -> definition-free path -> use of buf

E.g., is there an information leak?

```c
void * fp = &exit // sensitive source
...
x->f = fp;
y = x;
fp1 = y->f;
printf("libc exit function @ %p\n" fp1) // sink
```
Program Analysis in Security

- E.g., is there an information leak?
  
  ```
  gold += fight(map[row*3+col]); // source
  ...
  print_highscore(gold); // printf(param) leak, sink
  ```

- E.g., is there a TICTOU bug?
  
  ```
  char * buf = malloc(bar->name_len);
  ...
  modifies_bar(bar); // we can detect side-effects!
  ...
  memcpy(buf, bar->name, bar->name_len);
  ```

- E.g., Is there a buffer overflow? Many analyses
Our focus will be Static Analysis

- Many techniques
  - Decades of research and Turing Awards!
  - Dataflow analysis and abstract interpretation
    - Kildall ’73, Kam and Ullman ’77, Cousot & Cousot ’77
  - Types and type-based analysis
    - Following John Backus’ “Can Programming…” ’78
  - Axiomatic semantics (i.e., Hoare Logic)
    - C.A.R. Hoare’s “An Axiomatic Basis for Computer Programming”, ’69
Static Analysis

What this course is mostly about

- How can we define the meaning of programs
- How can we model behavior of programs, and prove theorems about programs

- How can we use tools and build tools that automatically reason about programs

Many applications
Applications

- Compiler optimization, traditional application
  - We’ll start with dataflow analysis
- Finding bugs, verifying the absence of bugs
- Improving security and privacy
- Designing languages that prevent bugs
- Refactoring and testing
- Improving energy efficiency
- Education. Submitty uses static analysis!
Examples of Properties Deducible by Static Analysis

- Can $x$ ever be null at program point $i$: $x.m()$?
- Can $y$ be different than 1 at program point $i$: $x = y*10$?
- Can $n$ at $x[n]$ cause out-of-bounds access?
- Does an app leak private data (e.g., phone number, phone identifier, location) to ad networks?
  - Answer: Yes!
Examples of Properties Deducible by Static Analysis

- What inputs avoid divide-by-zero at \( \frac{x}{y} \)?
  \[
  \{ x \neq 1 \land x \neq -2 \}
  \]

- \( y = x + 4; \)
- \( \text{if} \ (x > 0) \{ \)
  \( \quad y = x^2 - 1; \)
- \( \} \)
- \( \text{else} \ { \)
  \( \quad y = y + x; \)
- \( \} \)
- \( \{ y \neq 0 \} \)
- \( x = x/y; \)

- Formalism of Axiomatic Semantics (Hoare logic)
  - Different from dataflow and types
  - Allows us to specify program behavior with preconditions and postconditions that form logical assertions
    - Support complex logics
    - Enables reasoning about correctness
Nature of Static Analysis

- To remain computable, static analysis must approximate. It is undecidable to find exactly what happens at runtime.
  - Typically, analysis errs on the safe (sound) side -- that is, it over-approximates
    - E.g., analysis reports that $x$ at some program point can be either 1 or 2, even though $x$ is always 1
    - A type system rejects correct programs
  - Sometimes, analysis is unsafe (unsound) --- that is, it under-approximates
A static analysis is said to be safe (also, sound, correct) if it over-approximates, that is, it accounts for every execution path.

E.g., in previous example the analysis that reports $x$ in \{1, 2\} is safe, but so is the one that reports $x$ in \{1, 2, 21\}.

An analysis that reports $x$ in \{0, 2\} is unsafe (unsound, incorrect).
Analysis Safety

Safety is crucial when analysis enables compiler optimizations. Why?

E.g., an unsafe analysis may report that $y$ is 1 at $z = y \times 10$ along all execution paths, while in fact there is an execution path that sets $y$ to 10. If the optimizing compiler changes $z = y \times 10$ to $z = 10$, the program produces incorrect result along the path when $y$ is 10!
Safety is often relinquished when analysis is used in static debugging tools. Why?

E.g., suppose we have a piece of code that contains 10 “true” null-pointer dereferences.

- Safe analysis A reports 100 potential null-pointer dereferences (all 10 “true” bugs and 90 “false-positives”).
- Unsafe analysis B reports 10 potential null-pointer dereferences (8 “true” and 2 false-positives). Which one would you take?
Analysis Precision

- Analysis precision refers to how “close” results are to actual runtime
  - E.g., in our running example, the analysis that reports $x$ in \( \{1,2\} \) is more precise than the one that reports $x$ in \( \{1,2,21\} \).
  - Typically, we use the term precision with safe analysis (safe analysis has 100% recall)

- Wide spectrum of static analyses and tradeoff between cost and precision
Outline

- Logistics
  www.cs.rpi.edu/~milanova/csci4450/

- Static analysis, introduction

- Course topics, tools, and homework

- Introduction to Dataflow analysis
Course Topics

- Dataflow analysis
  - Lattices, transfer functions, dataflow frameworks
  - Classical analyses: constant propagation and points-to analysis
  - Binary analysis

- Abstract interpretation (a more powerful formalism)
  - Abstract vs. concrete semantics
  - Galois connections
Course Topics

- Types and type-based analysis
  - Simply typed Lambda calculus
  - Type systems and type soundness
  - Simple type inference
  - Hindley Milner type inference
Course Topics

- Axiomatic semantics
  - You know already: Hoare logic!
  - Logics to specify assertions (as you know them, \( P \) and \( Q \) in \( \{ P \} \) code \( \{ Q \} \))
  - SMT solvers and proving Hoare triples
Historical Perspective

- “An axiomatic basis for computer programming” by C.A.R. Hoare 1969
  - Great enthusiasm about verification 1970-ties
- “Social processes and proofs of theorems and programs” by De Millo, Lipton, and Perlis 1979
  - Credited with setting back work on formal verification
- “Can programming be liberated… A functional style and its algebra of programs” by John Backus 1977
  - Research on functional programming, type theory
- Z3 theorem prover from Microsoft about 2005
  - Lots of new enthusiasm about verification and symbolic execution
Tools and Programming Languages

- Soot
- Z3
- Ghidra (optional)
- Java
- Haskell
- OCaml
Homework Assignments

- There will be 6-7 homework assignments
  - Each makes about 5-6% of your grade
  - Larger assignments are broken into 2-3 parts
  - Some are individual, some are team assignments

- Submitty!

CSCI 4450/6450, A Milanova
Homework Assignments

- HW1
  - Problem set to practice dataflow analysis

- HW2-HW4
  - Classical OO analyses in Soot: the CHA, RTA, and XTA family of analyses

- HW5-HW6
  - Problem set to practice abstract interpretation/type inference concepts
  - Implement simple type inference (and maybe Hindley Milner) in Haskell
Homework Assignments

- HW7
  - Implement a simple verifier for C using Z3

- If you are interested in Binary analysis, we’ll replace parts of HW2-HW4 and HW7 with Ghidra projects
  - E.g., a taint analysis, a buffer overflow analysis, or other
Dataflow Analysis
Gary Kildall, “A Unified Approach to Global Program Optimization”, POPL 1973

John Kam and Jeff Ullman, “Monotone Dataflow Analysis Frameworks”, Acta Inf. 1977
Outline

- Motivation and origin of dataflow analysis: compiler optimization
- Overview of the compiler
- Classical compiler optimizations
- Control flow graphs

Reading:
  - Dragon Book, Chapter 9.1
Overview of the Compiler

- Phases of the compiler
  - Lexical Analyzer (scanner)
  - Syntax Analyzer (parser)
  - Semantic Analyzer and Intermediate Code Generator
  - Machine-Independent Code Optimizer
  - Code Generator
  - Machine-Dependent Code Optimizer
Overview of the Compiler

An optimization is a semantics-preserving transformation
Classical Compiler Optimizations

- We will show the classical optimizations using an example Fortran loop
- Opportunities for optimization due to automatic generation of intermediate code

```fortran
sum = 0
10  sum = sum + a[i]*a[i]
```

...
1. \( \text{sum} = 0 \) \hspace{2cm} \text{initialize sum}
2. \( i = 1 \) \hspace{2cm} \text{initialize loop counter}
3. \( \text{if } i > n \text{ goto 15} \) \hspace{2cm} \text{loop test, check for limit}
4. \( t1 = \text{addr}(a) - 4 \)
5. \( t2 = i \times 4 \)
6. \( t3 = t1[t2] \)
7. \( t4 = \text{addr}(a) - 4 \)
8. \( t5 = i \times 4 \)
9. \( t6 = t4[t5] \)
10. \( t7 = t3 \times t6 \)
11. \( \text{sum} = t8 \) \hspace{2cm} \text{increment sum}
12. \( i = i + 1 \) \hspace{2cm} \text{increment loop counter}
13. \( \text{goto 3} \)
14. \( \text{...} \)
1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i*4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i*4
9. t6 = t4[t5]
10. t7 = t3*t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. ...
Common Subexpression Elimination

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i*4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i*4
9. t6 = t4[t5]
10. t7 = t3*t6
11. sum = t8
12. i = i + 1
13. goto 3
14. ...

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i*4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i*4
9. t6 = t4[t5]
10. t7 = t3*t6
10a t7 = t3*t3
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
After Common Subexpression Elimination

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i*4
6. t3 = t1[t2]
10a t7 = t3*t3
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
Copy Propagation

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
10a t7 = t3 * t3
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. ...

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
10a t7 = t3 * t3
11. t8 = sum + t7
11a sum = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. ...
After Copy Propagation

1. \( \text{sum} = 0 \)
2. \( i = 1 \)
3. \( \text{if } i > n \text{ goto } 15 \)
4. \( t1 = \text{addr}(a) - 4 \)
5. \( t2 = i \times 4 \)
6. \( t3 = t1[t2] \)
10a \( t7 = t3 \times t3 \)
11a \( \text{sum} = \text{sum} + t7 \)
13. \( i = i + 1 \)
14. goto 3
15. ...
Invariant Code Motion

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …
After Invariant Code Motion

1. sum = 0
2. i = 1
   2a t1 = addr(a) − 4
3. if i > n goto 15
5. t2 = i * 4
6. t3 = t1[ t2 ]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …
1. sum = 0
2. i = 1
3. if i > n goto 15
5. t2 = i * 4
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. ...

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
3. if i > n goto 15
5. t2 = i * 4
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
13. i = i + 1
14. goto 3
15. ...
After Strength Reduction

1. \text{sum} = 0
2. \text{i} = 1
2a \text{t1} = \text{addr}(a) - 4
2b \text{t2} = \text{i} \times 4
3. \text{if} \ i > n \ \text{goto} \ 15
6. \text{t3} = \text{t1}[\text{t2}]
10a \text{t7} = \text{t3} \times \text{t3}
11a \text{sum} = \text{sum} + \text{t7}
11b \text{t2} = \text{t2} + 4
13. \text{i} = \text{i} + 1
14. \text{goto} \ 3
15. ...
1. \textit{sum} = 0
2. \textit{i} = 1
2a \textit{t1} = \text{addr(}a\text{)} - 4
2b \textit{t2} = \textit{i} \times 4
3. \textit{if} \textit{i} > \textit{n} \textbf{goto} 15
6. \textit{t3} = \textit{t1}[\textit{t2}]
10a \textit{t7} = \textit{t3} \times \textit{t3}
11a \textit{sum} = \textit{sum} + \textit{t7}
11b \textit{t2} = \textit{t2} + 4
13. \textit{i} = \textit{i} + 1
14. \textbf{goto} 3
15. …

1. \textit{sum} = 0
2. \textit{i} = 1
2a \textit{t1} = \text{addr(}a\text{)} - 4
2b \textit{t2} = \textit{i} \times 4
2c \textit{t9} = \textit{n} \times 4
3. \textit{if} \textit{i} > \textit{n} \textbf{goto} 15
3a \textit{if} \textit{t2} > \textit{t9} \textbf{goto} 15
6. \textit{t3} = \textit{t1}[\textit{t2}]
10a \textit{t7} = \textit{t3} \times \textit{t3}
11a \textit{sum} = \textit{sum} + \textit{t7}
11b \textit{t2} = \textit{t2} + 4
13. \textit{i} = \textit{i} + 1
14. \textbf{goto} 3a
15. …
After Test Elision and Induction

Variable Elimination

1. \( \text{sum} = 0 \)
2. \( i = 1 \)
2a \( t_1 = \text{addr}(a) - 4 \)
2b \( t_2 = i \times 4 \)
2c \( t_9 = n \times 4 \)
3a \( \text{if } t_2 > t_9 \text{ goto 15} \)
6. \( t_3 = t_1[t_2] \)
10a \( t_7 = t_3 \times t_3 \)
11a \( \text{sum} = \text{sum} + t_7 \)
11b \( t_2 = t_2 + 4 \)
14. \( \text{goto 3a} \)
15. …
Constant Propagation and Dead Code Elimination

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
2c t9 = n * 4
3a if t2 > t9 goto 15
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
14. goto 3a
15. ...

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
2c t9 = n * 4
2d t2 = 4
3a if t2 > t9 goto 15
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
14. goto 3a
15. ...

CSCI 4450/6450, A. Milanova
New Control Flow Graph

1. $\text{sum} = 0$
2. $t1 = \text{addr}(a) - 4$
3. $t9 = n \times 4$
4. $t2 = 4$

5. if $t2 > t9$ goto 11

6. $t3 = t1[t2]$
7. $t7 = t3 \times t3$
8. $\text{sum} = \text{sum} + t7$
9. $t2 = t2 + 4$
10. goto 5

$T \rightarrow 11. \ldots$
$F \leftarrow$
Classical Compiler Optimizations

To summarize

- Common subexpression elimination
- Copy propagation
- Strength reduction
- Test elision and induction variable elimination
- Constant propagation
- Dead code elimination

Dataflow analysis enables these optimizations
Building the Control Flow Graph

Build the CFG from linear 3-address code:

- Step 1: partition code into basic blocks
  - Basic blocks are the nodes in the CFG
- Step 2: add control flow edges

Aside: in Principles of Software, we built a CFG from structured (AST) IR:

- $S ::= x = y \text{ op } z \mid S;S \mid \text{ if (b) then } S \text{ else } S \mid \text{ while (b) } S$
Step 1. Partition Code Into Basic Blocks

1. Determine the leader statements:
   (i) First program statement
   (ii) Targets of conditional or unconditional goto’s
   (iii) Any statement following a goto

2. For each leader, its basic block consists of the leader and all statements up to, but not including, the next leader or the end of the program

CSCI 4450/6450, A Milanova
Question. Find the Leader Statements

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i*4
6. t3 = t1[t2]
7. t4 = addr(a) – 4
8. t5 = i*4
9. t6 = t5[t5]
10. t7 = t3*t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. ...
Step 2. Add Control Flow Edges

There is a directed edge from basic block $B_1$ to block $B_2$ if $B_2$ can immediately follow $B_1$ in some execution sequence.

Determine edges as follows:

(i) There is an edge from $B_1$ to $B_2$ if $B_2$ follows $B_1$ in three-address code, and $B_1$ does not end in an unconditional goto.

(ii) There is an edge from $B_1$ to $B_2$ if there is a goto from the last statement in $B_1$ to the first statement in $B_2$. 

CSCI 4450/6450, A Milanova
1. \( \text{sum} = 0 \)
2. \( i = 1 \)
3. \( \text{if } i > n \text{ goto 15} \)
4. \( t1 = \text{addr}(a) - 4 \)
5. \( t2 = i \times 4 \)
6. \( t3 = t1[t2] \)
7. \( t4 = \text{addr}(a) - 4 \)
8. \( t5 = i \times 4 \)
9. \( t6 = t5[t5] \)
10. \( t7 = t3 \times t6 \)
11. \( \text{sum} = t8 \)
12. \( i = i + 1 \)
13. \( \text{goto 3} \)
14. \( … \)
Next Class

- Dataflow analysis
- Four classical dataflow analysis problems