CSC2/455 Software Analysis and Improvement
Introduction

Sreepathi Pai

January 12, 2022

URCS

Outline

Classical Compiler Analysis

Program Analysis

Fundamental Issues

Administrivia

Outline

An example program

```
#include <stdio.h>

int main(void) {
  int N = 10000;
  int sum = 0;

  for(int i = 1; i < N; i++) {
    sum += i;
  }

  if(sum > 0) {
    printf("%d: %x\n", sum, sum);
  } else {
    printf("sum is zero\n");
  }

  return 0;
}
```

Compiled with `gcc -O0`

```
main:
    pushq    %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $10000, -4(%rbp)
    movl    $0, -12(%rbp)
    movl    $1, -8(%rbp)
    jmp    .L2
.L3:
    movl    -8(%rbp), %eax
    addl    %eax, -12(%rbp)
    addl    $1, -8(%rbp)
.L2:
    movl    -8(%rbp), %eax
    cmpl    -4(%rbp), %eax
    jl    .L3
    cmpl    $0, -12(%rbp)
    jle    .L4
    movl    -12(%rbp), %edx
    movl    -12(%rbp), %eax
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    jmp    .L5
.L4:
    leaq    .LC1(%rip), %rdi
    call    puts@PLT
.L5:
    movl    $0, %eax
    leave
    ret
```

Compiled with `gcc -O1`

```
main:
    subq    $8, %rsp
    movl    $9999, %eax
.L2:
    movl    $49995000, %ecx
    movl    $49995000, %edx
    leaq    .LC0(%rip), %rsi
    movl    $1, %edi
    movl    $0, %eax
    call    __printf_chk@PLT
    movl    $0, %eax
    addq    $8, %rsp
    ret
```

The compiler:

- Eliminated the `for` loop
  - Replaced it with the value computed
- Eliminated the `else` part of the `if/then`
  - Because it would never execute

How did it do that?

How do you think the compiler did this?

Compilers today

- GNU Compiler Collection (GCC)
- LLVM
  - Many proprietary compilers based on LLVM
- Visual Studio
- Proprietary compilers
  - Intel `icc`
  - NVIDIA/Portland Group

Outline

Analysis (and Verification)

- How many times will this loop execute?
- Will this condition always be true?
- Is this value always a constant?
- Are there bugs in a given piece code?
  - NULL pointer dereferences?
  - Data races (in concurrent code)?
- Is this code correct (according to some specification)?

Properties

- Safety properties
  - Informally, “something bad will never happen”
  - Formally, a property that is always true
- Liveness properties
  - Informally, “something good will eventually happen”
  - Formally, a property that will eventually be true
- Other properties:
  - All allocated memory is freed
  - All open files are closed

Program Analysis Tools today

- (Synopsys) Coverity
- GrammaTech CodeSonar
- Facebook Infer
- Frama-C
- PVS-Studio
- Microsoft SLAM
- Lots of others...

Outline

What does this Python program do?

```
for i in range(n):
    print("boom!")
```

Strategy: Run (or Interpret) the program

- Running the program and observing what it does is a perfectly reasonable way of analyzing a program
  - Maybe run it in a simulator/VM or interpreter
- What problems do you anticipate with this strategy?

Some potential problems

- Too long
- Infinite loop (aka non-termination)
- Number of inputs may be infinite!
    - Behaviour may depend on input

What does this program do?

```
def collatz(n): # n is a positive integer
    while n > 1:
        print(n)
        if n % 2 == 0:
            n = n // 2 # integer division
        else:
            n = 3 * n + 1

    for i in range(n):
        print("boom!")
```

Some runs: n=5

```
5
16
8
4
2
1
boom!
```

Some runs: n=12

```
12
6
3
10
5
16
8
4
2
1
boom!
```

Analyzing this program

- This program will print only 1 `boom!`
- *If* the loop terminates
  - Only if n is always reduced to 1
  - Is it always? ($500 reward!)
- Can we determine if the loop terminates?
  - For any n?
  - For a fixed n?

Undecidability: The Halting Problem

- In general, an algorithm cannot determine if a program will terminate on a given input
- What does this imply for program analysis?
    - End of this course?

Program Analysis

- Program analysis needed for optimization ("making programs faster")
  - Reducing number of operations
  - Substituting cheaper operations
  - Increasing parallelism of operations
- Also need for verification ("security")
  - Will this program crash for any input?
  - Will this program leak memory? (`malloc` but no corresponding `free`)
  - Will this program read another user's files?
  - Can a program be subverted to obtain `root` access?
- Computers everywhere, such questions far more important now!
  - 4.4B people have smartphones!

Mission Impossible?

- No general technique to analyze programs
  - Many different approaches
- We will study many of these
  - Basic
  - Advanced
  - Recent advances

Roadmap of the course: Part I

- Dataflow Analysis (DFA)
  - Mostly used in compilers
  - Automatic, fast, and approximate
- You'll construct the "mid-end" of a compiler for a subset of the C language
  - Introduces you to the engineering and theoretical aspects
- One specialized non-compiler tool: Oracle's Soufflé

Program Analysis: Part II

- Abstract Interpretation (AI)
  - Automatic, slower, and approximate
  - Specialized tools: Facebook Infer, many others
- Model Checking
  - Automatic, fast (but not as fast as DFA), and precise
  - Can be intractable
  - We'll look at CBMC, perhaps also Spin
- Symbolic Execution
  - Automatic, fast, and precise
  - Can be intractable
  - We'll look at KLEE and maybe Angr
- Deductive Techniques
  - Manual to semi-automated, as fast as you think, and precise
  - Essentially involves writing proofs
  - We'll look at (Microsoft) Dafny

Some things program analysis makes possible

- Fast Javascript
  - pioneered by Google's V8
- Safe in-kernel execution of user-provided code
  - Linux eBPF
  - pioneered by Sun's DTrace
- Safe systems programming languages
  - Rust
- Airplanes in the sky
  - Remember Boeing 737 MAX?

Outline

People

- Instructor: Dr. Sreepathi Pai
  - E-mail: sree@cs.rochester.edu
  - Office: In cyberspace, see Blackboard for Zoom link
  - Office Hours: Tuesday 13:00 to 14:00 (or by appointment)
- TAs:
  - Paul Ouellette

Places

- Class: Zoom (for now), Gavett 301 (when we are back in person)
  - M,W 1025–1140
- Course Website
  - `https://cs.rochester.edu/~sree/courses/csc-255-455/spring-2022/`
- Blackboard
  - Announcements
- Gradescope
  - Assignments, Homeworks, Grades, etc.
- Piazza
  - Help

References

- Three textbooks
  - Aho, Lam, Ullman, Sethi, *Compilers: Principles, Techniques and Tools*
  - Cooper and Torczon, *Engineering a Compiler*
  - Rival and Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective*, MIT Press, 2020
- This course requires a lot of reading!
- See Blackboard for information on accessing Reserves

Grading

- Homeworks: 15%
- Assignments: 70% (5 to 6)
- Exams/Project: 15% (TBD)
- Graduate students should expect to read a lot more, and work on harder problems.

There is no fixed grading curve.

See course website for late submissions policy.

Academic Honesty

- Unless explicitly allowed, you may not show your code to other students
- You may discuss, brainstorm, etc. with your fellow students but all submitted work must be your own
- All help received must be acknowledged in writing when submitting your assignments and homeworks
- All external code you use must be clearly marked as such in your submission
    - Use a comment and provide URL if appropriate
- If in doubt, ask the instructor
- It is a violation of course honesty to make your assignments on GitHub (or similar sites) public

All violations of academic honesty will be dealt with strictly as per UR's Academic Honesty Policy.

Course Goals

- Write your own compilers
- Modify existing compilers
- Read about new analyses (in research papers)
- Create new analyses
- Use analysis tools and frameworks