Introduction
Stack-smashing is very related to compiler
Some of the biggest software vulnerabilities are due to stack smashing
We will look at that in one of the following lectures

The exam will be open-note (not open laptop) - you can bring whatever reference you want
You are not supposed to memorize codes so you can bring required references with you.
Final exam = Homework questions + some multiple-choice questions + true/false questions
Practice final will be out until the end of next week

Control Flow Analysis
All the stuff you learned about has been around for decades
Sometimes your pcode might have repeated statements or dead code
To remove these kinds of extras from our code, we have another field that deals with that: compiler optimization

Compiler optimization is done at compile time (different than dynamic programming)
There is a lot of ways to write the same code that has the same functionality

All variables are stored in the stack
To figure out which register to choose for which variable, we have compiler optimization algorithms

To implement x^2, we can have many ways
For example, we can use bit shift. But, in general, if you want to implement exponent, you cannot always use bit shifting. For ^2 case, you can prefer to use bit shifting for optimization.

Compiler optimization: faster and smaller programs

These days, compilers don’t generate machine code directly - instead, it generates intermediate code first
Industry standard intermediate code is LLVM
It is something that code from any programming language can be translated to
Very close to machine code
AST is hard to optimize:

- Language specific
- Processed by tree traversal: we cannot find order of steps the program is doing easily
- Little information about what program does

Instead, we can make use of control-flow graphs for optimization, which is easier to analyze

Control-Flow Graphs
Nodes are straight line codes, edges are branches
Loops are represented by back edges
Conditional branches have more than one edge.

The equivalent C code for the control-flow graph included in the lecture slides:

```
while( ++x <= 0 ) {
  b++;
}
return x;
```

State Transitions
Trace: the order of nodes that we meet when we traverse the control flow graph
The possible traces of a program can be infinite

Some possible questions for code analysis:

- How many times a branch is taken?
- Is this branch ever taken?

It might be impossible even for a human to sit down and figure out
Trying to figure out all possible traces are in compile time is not an easy task to do

Control flow graph represents every possible trace that you can generate from a program

There is a bunch of classic graph algorithms to deal with control flow graphs
Just like tree traversal algorithms, there are depth-first search, breath-first search algorithms for graphs

There are couple of different algorithms in order to create control flow graph:

- AST → CFG (Control flow graph)
- IR → CFG (they can look at machine code and infer what is control flow graph)

Graph algorithms are conservative: having infinitely many states is not a problem while analyzing CFG

Applications of control flow graph (CFG): Optimization

- **Register allocation** to figure out how to optimally allocate registers to avoid slower memory operations
- **Removing dead code** to have smaller programs in size
- **Constant propagation/folding**: we can figure out the results of some expressions in compile time and propagate the resulting value. We can avoid additional load/store operations by following such approach
- **Common subexpression evaluation** If you are using the same expression more than once in the program, compiler can recognize it, evaluate it once and replace all places it is used with the result
- And more

With control flow, you can also do some other program analysis:

- malloc without free: You can detect malloc without free. You can prove if malloc is dominated by free
- Null pointer error
- Divide-by-zero
- Use-after-free
- Uninitialized variable use
- Array out-of-bounds access

Despite these tasks are hard to achieve by manual inspections, we have programs to do that

Control flow analysis vs data flow analysis

Control flow graphs are per-function.
One of the challenges is figuring out which function is being called when function pointers are used
You can point to functions themselves using function pointers - which complicates code. You can run arbitrary section of your memory using a branch to some function pointer.

Halting problem: can I write a program that can detect if another program has an infinite loop **in general**
So, checking if a program includes while(true) is not enough - because we want a general solution

Data Flow Analysis
Control flow analysis tells us all possible ways that you can step through in your program
Data flow analysis tells you the flow of the data in your program rather than the control
Optimization is one reason why data flow analysis is used, e.g., to use registers instead of load/store operations to avoid load/store overhead from memory