# Intro to Software Analysis

CS390R - UMass Amherst

# Course Information

- Project 4 due shortly after Patriot’s day holiday

# Today’s Content

- Intro
- Dynamic Program Analysis
- Static Program Analysis
- Hybrid Program Analysis
- Various Representations (Call Graph, CFG, DFG, IR, SSA)
- Program Invariants
- Concrete vs Abstract State
- Dataflow Analysis
- Additional Resources

# What is Program Analysis

- Extract and analyze various properties of a program to make informed decisions towards some goals (eg. compilers analyzing programs you write to implement various optimizations)
- 3 main types we will be covering:
  - Dynamic - analysis happens at execution time
  - Static - analysis is done without running the program (eg. at compile-time or based on the on-disk executable)
  - Hybrid analysis combining both of the above

# Dynamic Program Analysis

- Printf-debugging - Manually instrumenting program to learn information about it during execution
- Fuzzing - Run program repeatedly with different inputs to find crashes
- Valgrind - Detects memory leaks at runtime
- Dynamorio/Intel-pin - Trace program execution
- Debugging program with GDB

# Static Program Analysis

- Coverity/Lint - programs used to automatically find bugs in software
- Decompiler Scripting - Writing scripts for eg. Ghidra that allow you to automate some tedious tasks
- Dataflow Analysis - Analysis of how data flows through programs, heavily used by compilers for optimizations

# Hybrid Analysis

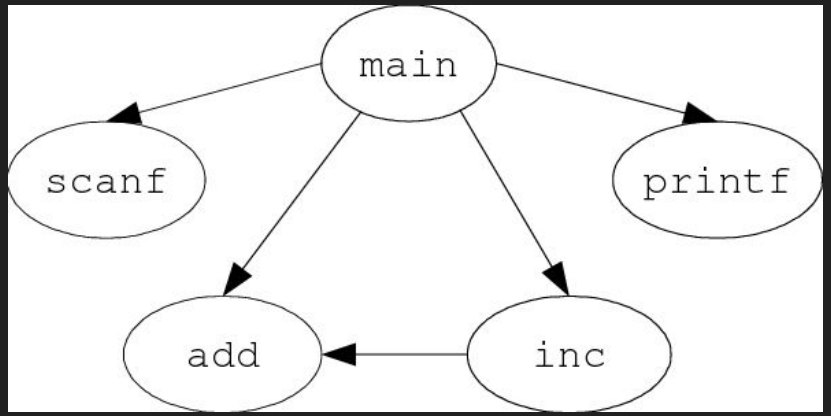
- LLVM - Compiler backend that performs various analysis passes on internal data-structures to optimize the final program
  - LLVM also accepts custom user-written passes which users can use to instrument programs to their desires
- Symbolic Execution - Transform a program into a symbolic state/a set of mathematical equations and solve for a solution

# Useful Program Representations

- Call-Graph
- IR - Intermediate Representation
- CFG - Control Flow Graph
- DFG - Data Flow Graph
- SSA - Single Static Assignment

# Call Graph

- This representation form abstracts functions to single nodes in a graph, and represents control-flow relations between functions using graph-edges
- This type of representation is useful if you want to perform some high-level analysis of how different functions interact
- Analysis passes can set ‘predicate informers’ that derive the requirements which a function places on its callers.This can then be used alongside the call graph to verify that all callers respect the derived properties



# IR - Intermediate Representation

- Code representation used internally by compilers/decompilers to represent code
- This means that a compiler that can compile to multiple architectures does not need separate optimization passes for each architecture. It can just compile the source-code down to the IR, perform optimizations on that, and then transform the IR to assembly for the requested architecture
- Most modern compilers, interpreters, and decompiler uses these (llvm, python, javascript, ghidra, binary ninja)

```
006120a6  6a01            push    0x1 {var_3c}
006120a8  e88af7ffff      call    ___scrt_initialize_crt
006120ad  59              pop     ecx  {0x1}
006120ae  84c0            test    al, al
006120b0  7507            jne     0x6120b9

006120b2  6a07            push    0x7
006120b4  e858040000      call    sub_612511

006120b9  32db            xor     bl, bl  {0x0}
006120bb  885de7          mov     byte [ebp-0x19 {var_1d_1}], bl  {0x0}
006120be  8365fc00        and     dword [ebp-0x4 {var_8_1}], 0x0
006120c2  e83bf7ffff      call    sub_611802
006120c7  8845dc          mov     byte [ebp-0x24 {var_28_1}], al
006120ca  a134857800      mov     eax, dword [data_788534]
006120cf  33c9            xor     ecx, ecx
006120d1  41              inc     ecx
006120d2  3bc1            cmp     eax, ecx
006120d4  74dc            je      0x6120b2
```

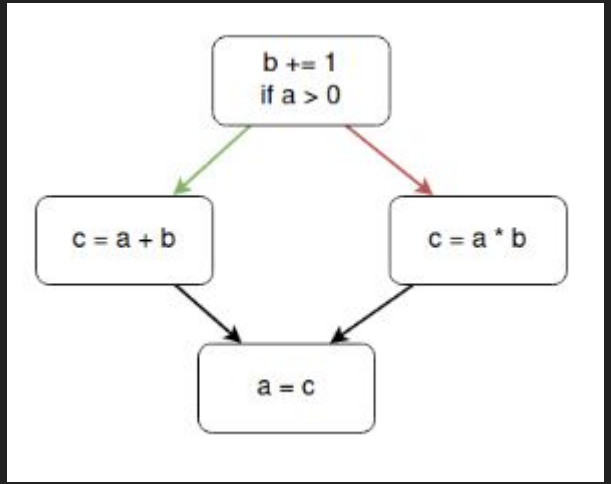
```
22 @ 006120a6  var_3c = 1
23 @ 006120a8  eax = ___scrt_initialize_crt(1)
24 @ 006120ad  ecx = 1
25 @ 006120ad  esp_1 = &var_38
26 @ 006120b0  if (eax != 0) then 27 @ 0x6120b9 else 34 @ 0x6120b2

27 @ 006120b9  ebx = 0
28 @ 006120bb  var_1d_1 = nullptr
29 @ 006120be  var_8_1 = 0
30 @ 006120c2  eax_1 = sub_611802()
31 @ 006120c7  var_28_1 = eax_1
32 @ 006120ca  eax_2 = [&data_788534].d
33 @ 006120d4  if (eax_2 == 1) then 34 @ 0x6120b2 else 38 @ 0x6120d8
```

# CFG - Control Flow Graph

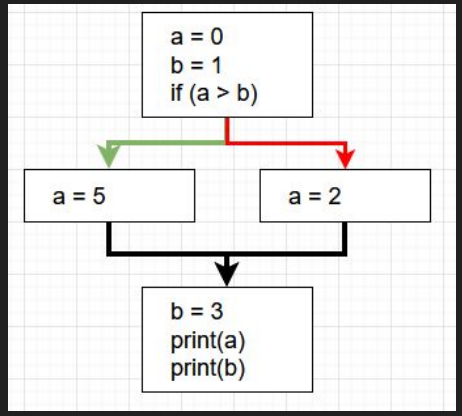
- This representation groups instructions that are always executed sequentially into graph nodes
- Control-flow is represented using edges between code-blocks
- This is generally a very intuitive representation and is very widely used
- This graph-structure is not optimized for data-flow analysis, so a separate graph is usually used alongside this in compilers

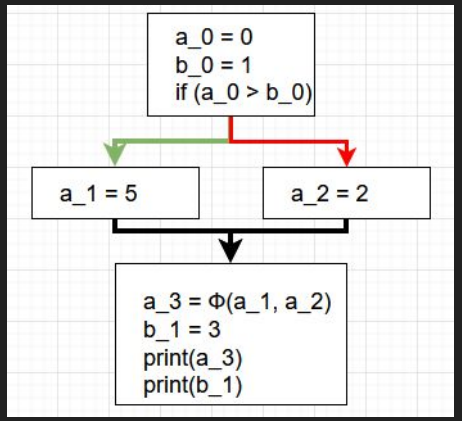
```
function f(a, b) {
    b++;
    if (a > 0) {
        c = a + b;
    }
    else {
        c = a * b;
    }
    a = c;
}
```



# SSA - Single Static Assignment

- IR that requires each variable to be immutable and thus assigned exactly once, and each variable needs to be assigned before it is used
- Determining which variable to use after branching control-flow is a little harder, that's where phi-nodes come in.
- They are basically flow-sensitive copy instructions that are assigned based on incoming CFG edges
- Data-flow analysis and optimizations become simpler if each variable has 1 definition
- This form is also used in most compilers for optimizations including llvm

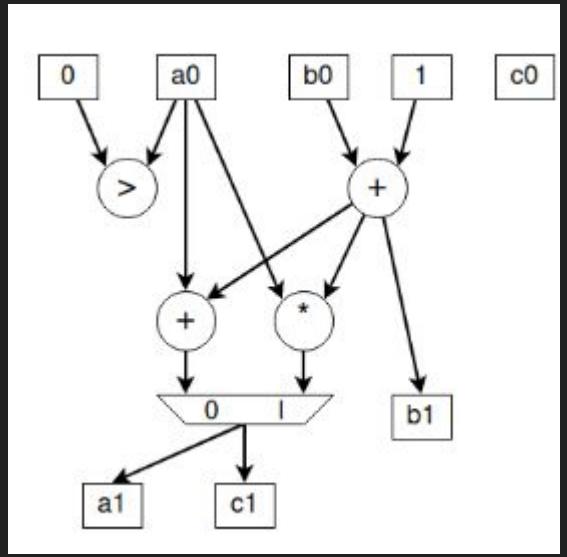


# DFG - Data Flow Graph

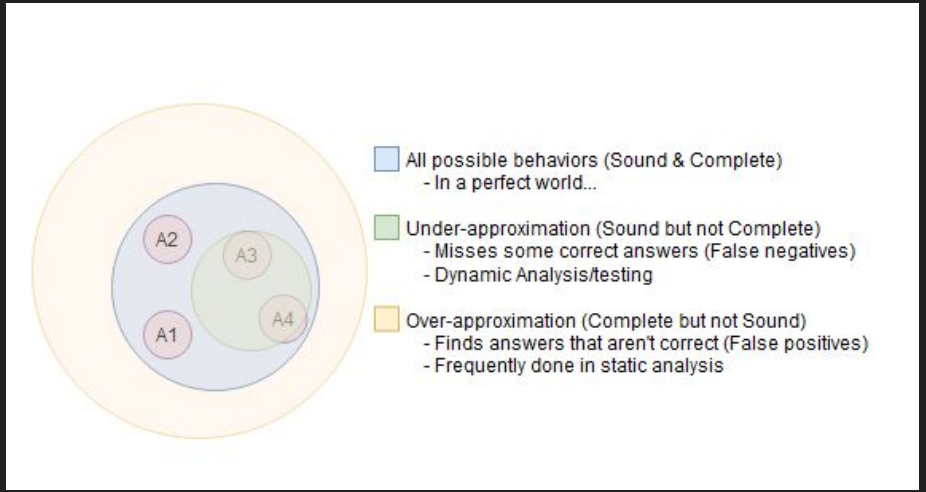
- Only focused on how data flows through a program, control-flow is ignored
- From a human perspective, this can look very messy, from a computer perspective however, this represents data-flow in a very easy-to-analyze manner
- The DFG does not enforce any node ordering at all, it just serves to track dependencies between data.
- Since this completely ignores control-flow, it is usually coupled with a CFG for analysis purposes

```
function f(a, b) {
    b++;
    if (a > 0) {
        c = a + b;
    }
    else {
        c = a * b;
    }
    a = c;
}
```



# Terminology

- **Termination** - The running analysis is proven to eventually complete
- **Completeness** - No false negatives (Returns all correct results, but might also accidentally return some false results)
- **Soundness** - No false positives (No incorrect results, might miss some correct results though)
- All static program analysis is undecidable, this can be proven through a reduction of the halting problem, this means that static analysis can never be sound & complete
- The design tradeoffs are usually determined by the consumer (compilers have different priorities than vuln-finding tools)



# Program Invariants

- A set of assumptions that are always valid for certain points of some program

```
void func(unsigned char arg) {
    unsigned char tmp = arg;

    while (++tmp != 23);

    if (tmp != 23) {
        puts("Error occurred");
    } else {
        puts("Success!");
    }
}
```

An invariant for this program could be that at the location of the if-statement, tmp will always be 23, so the compiler would be able to optimize out a big chunk of the code based on this invariant.

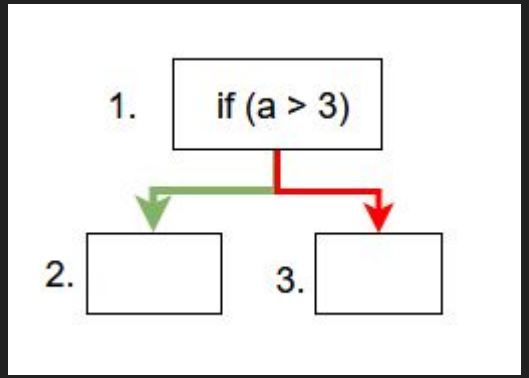
```
void func(unsigned char arg) {
    puts("Success!");
}
```

# Program Invariants - Dynamic Analysis

- With dynamic analysis, for more complex programs, we could never represent every single state, that’s why these are generally not complete.
- In the previous example, there are only 255 possible inputs (unsigned char), so we could solve this dynamically. For larger programs however, this becomes impossible

# Concrete vs Abstract State

- Concrete State
  - All variables have some exact value (eg. x == 3)
  - This means that to find all possible outputs out a u64 input, we would need to 2^64 different inputs. This is what most dynamic analysis relies on, and what we were doing with fuzzing
  - For complex programs this means that we will never be able to confidently say that we found all possible outputs because we are only able to test a subset of possible inputs
- Abstract State
  - Variables are represented via some abstraction, so for the example on the right, at block 2. we would record that ‘a > 3’, and for block 3, ‘a < =3’, thus creating different program states whenever branches occur. This is super powerful, but has some difficult to work around drawbacks
  - This type of abstraction is heavily used in a program analysis technique called symbolic execution that we will cover in a later lecture



# Program Invariants - Static Analysis

- Getting back to our earlier program, using an abstracted state, we could use abstract variables to determine that tmp will always be 23 after the while loop, thus making the “Error occurred” branch impossible
- A compiler could use this information to exclude the impossible branch from the generated program using a technique called dead-code elimination
- A bug-hunting program analysis tool might be able to use this to determine that a size passed to an ‘fgets’ call can never be too large for the buffer, thus reducing false positives it emits

```
void func(unsigned char arg) {
    unsigned char tmp = arg;

    while (++tmp != 23);

    if (tmp != 23) {
        puts("Error occurred");
    } else {
        puts("Success!");
    }
}
```

# Data Flow Analysis

- This type of analysis views computation through transitions of data through expressions and variable assignments
- For many purposes, this is maintained at the basic-block level in terms of Gen(n) and Kill(n), where Gen(n) denotes the data-flow information generated in block-n and Kill(n) denotes the dataflow information invalidated in block-n
- Edges in a CFG denote predecessor and successor relationships where for an edge n1->n2, n1 is the predecessor of n2 and n2 is the successor of n1.
- Instructions are generally extracted, so for an expression: ‘c = a + b’ we don’t care about the add operation, but only note that ‘a’ & ‘b’ are used, and ‘c’ is defined
- Using these operations we can generate Gen/Kill sets for a program. These then let us determine at which point of a program various variables are valid or have already been killed off
- This is used by compilers for register allocation, and can also be used to eg. find use-after-free bugs for security-focused program analysis

# There is so much more to learn…

Program analysis and dataflow analysis, be it for compiler optimization purposes or security analysis is a massive field with decades of research behind it.

An entire course focused solely on the theory of the different aspects of this could occupy multiple 600-level course (and hopefully will someday at UMass). This is meant to be a practical course though, so we will leave the theory with this, and start applying some of the things covered today in the next lectures.

That being said, if you are interested in program analysis as a research-field, the last slide has some good resources to get you started

# Shoutout to UMass Alumn

- https://jakob.space/blog/what-ive-learned-about-formal-methods.html

# Additional Resources

- Principles of Program Analysis (Textbook)
- Data Flow Analysis: Theory and Practice (Textbook)
- Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities (Textbook)
- University of Pennsylvania Course https://www.youtube.com/playlist?list=PLF3-CvSRq2SYXEiS80KuZQ80q8K2aHLQX
- University of Stuttgart Course https://www.youtube.com/playlist?list=PLBmY8PAxzwIEGtnJiucyGAnwWpxACE633
- https://edmcman.github.io/papers/oakland10.pdf