Disassembly: Binary Analysis Fundamentals

COMP3703 Software Security

Slides prepared by H. Gunadi and A. Tiu, based on Chapter 6 of Andriesse's "Practical Binary Analysis", No Starch Press, 2019.

Outline

- Static disassembly: linear vs. recursive.
- Dynamic disassembly (a.k.a. execution tracing): testing, fuzzing, and symbolic execution.
- Structuring code and data
- Decompilation
- Intermediate Representation (IR)

Outline

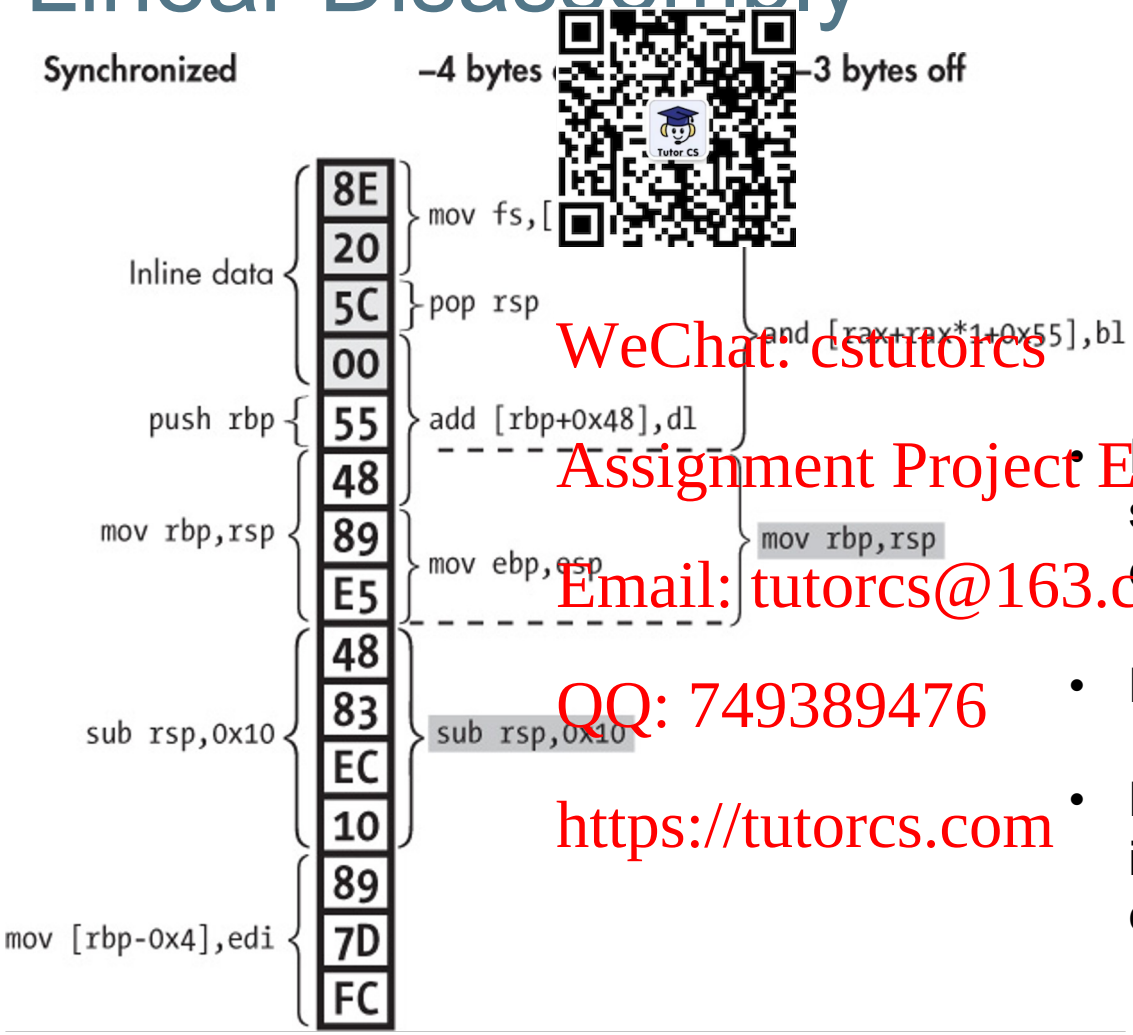
- Properties of binary analysis techniques:
  - Inter-procedural vs. intra-procedural.
  - Flow-sensitivity.
  - Context-sensitivity.
- Control-flow analysis: loop detection and cycle detection.
- Data-flow analysis:
  - Reaching Definition Analysis.
  - Use-def chains.
  - Program slicing.

Static Disassembly

- Goal: translate a binary into a form that a human can read or machine can process (for further analysis).
- Steps:
  1. Load a binary for processing
  2. Find all the machine instructions in the binary.
  3. Disassemble these instructions into human- or machine-readable form.
- Step 2 can be very difficult in practice.

Static Disassembly: Linear vs Recursive

Linear Disassembly



- Iterates through all code segments, decoding bytes consecutively.
- Example: objdump
- Potential *desynchronisation* if inline data is interpreted as code.

Example: disassembling bytes

There are a variety of t[illegible]can be used to see how byte sequences can be inte[illegible] x86 instructions. For example: ndisasm, objdump, etc.

Beware of misinterpreting data as instructions!

See http://ref.x86asm.net for list of x86 opcodes.


```
$ printf "\x8e\x20\x5c" | ndisasm -b 64 -
00000000  8E20              mov fs,[rax]
00000002  5C                pop rsp
$ printf "hello world" | ndisasm -b 64 -
00000000  68656C6C6F        push qword 0x6f6c6c65
00000005  20776F            and [rdi+0x6f],dh
00000008  726C              jc 0x76
0000000A  64                fs
```

Recursive Disassembly

- Starts from known entry points.
- Recursively follows control flow.
- Not all control flow is easy to follow, e.g., target of indirect jump.
- Used in many reverse-engineering applications.

Dynamic Disassembly

Dynamic Disassembly

- Runtime informa resolve indirect calls, distinguishing da de, etc.
- It allows for execution tracers to dump instructions, memory/register contents, etc.
- Code coverage problem:
  - It only sees the instructions that are actually executed during the analysis run.

Example: tracing with GDB

We can use gdb to [illegible] l the instructions executed in a run of a program.

- Using logging command and disable pagination, and then step over one instruction at a time. (By default output is saved in "gdb.txt")
- For example:


```
$ gdb -nx -q compilation_example
Reading symbols from compilation_example...
(gdb) set logging on
(gdb) set pagination off
(gdb) break main
Breakpoint 1 at 0x401130: file compilation_example.c, line 7.
(gdb) run
```

Code coverage: test suites

- Use known test [illegible] manually developed, to increase code coverage.
- Trying to cover as much of the program's functionality as possible.
- Ready-made test suites aren't always available.
- Application specific.

Code coverage: Fuzzing

- Fuzzers can automatically generate inputs.
- Favoring executing lots of tests to heavy duty analysis.
- Some well-known fuzzers: AFL, Libfuzzer.
- Sometimes it can be hard to reach code behind complex guards.
- Broadly divided into:
  - Generation-based fuzzer
  - Mutation-based fuzzer

Symbolic Execution

- Execute program with concrete values but with symbolic values, e.g., range of possible values.
- One execution path will generate a set of constraints.
- We can flip the branch by changing the constraints to that of the other branch.
- Plagued with path explosion.

Symbolic Execution

Code:		Constraints:
`x = int(argv[0])`	→	$x = a1$
`y = int(argv[1])`	→	$y = a2$
`z = x + y`	→	$z = a1 + a2$
`if(x < 5)`		
`foo(x, y, z)`	→	$(a1 < 5)$
`else`		
`bar(x, y, z)`	→	$(a1 \geq 5)$

Structuring Disassembled Code and Data

Structuring Disassembled Code

- Compartmentalization
  - Breaking code into logically connected chunks and make it easier to understand the relationship between chunks.
- Revealing control flow
  - Some structures can reveal control flow. Especially in visual representation, it can make it easier to see how control flows through the code and to get a quick idea of what the code does.

Functions

- Used to group logically connected code.
- Programs structured and properly divided into functions are much easier to understand.
- Most disassemblers make some effort to recover the original program's function structure (*function detection*).
- It can also help in automated analysis.

Function Detection: Challenges

- Binaries can be
  - We lose the information on the names, address, sizes, etc.
- Code belonging to a function might be scattered throughout the code section.
- Overlapping code blocks
  - Chunks of code may be shared between functions.
- Most disassemblers assumes that functions are contiguous and don't share code.
  - Holds true in many but not all cases.

Function Detection

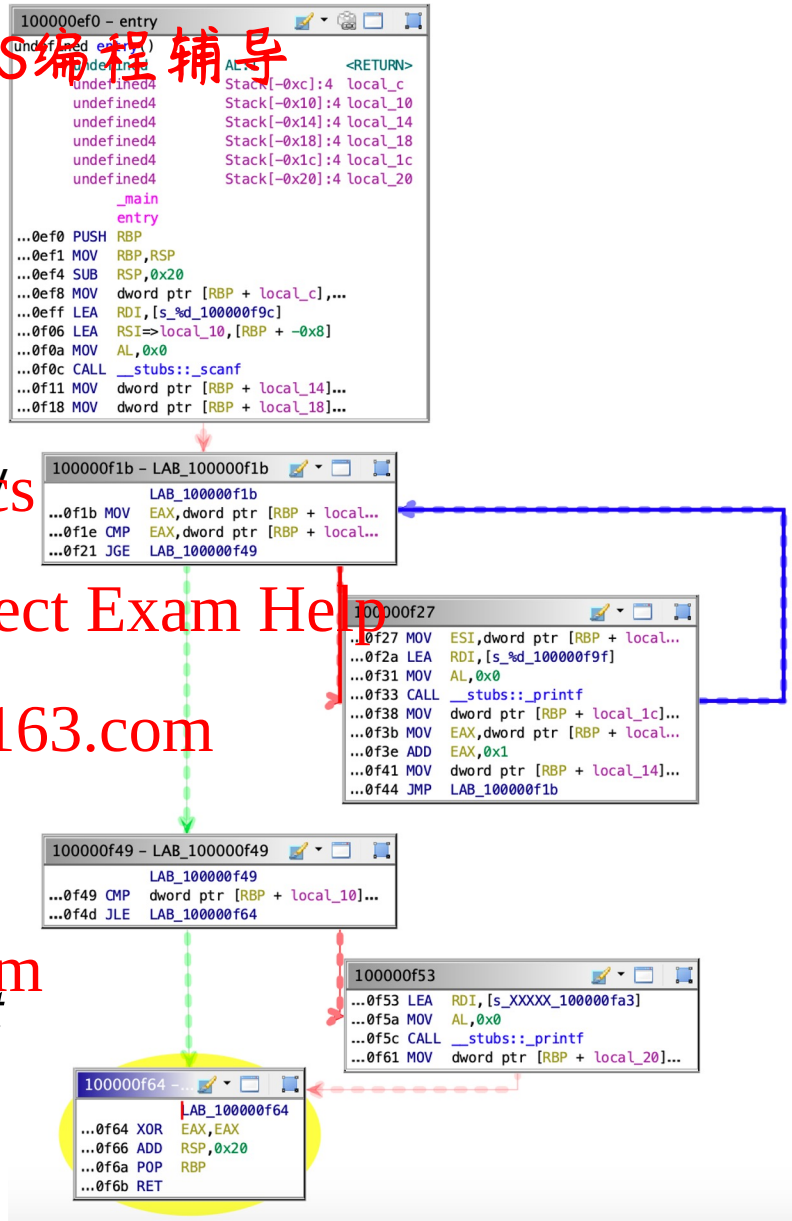
- Based on function s
  - Based on well-kn s, e.g., prologue `push ebp; mov ebp, esp ;` and epilogues, e.g., `leave; ret`.
  - This strategy is used in all well-known recursive disassemblers.
  - It can vary depending on the platform, compiler, and optimization level used.
- Functions that are called directly by the `call` instruction are easy to locate.
- Indirect or tail-call functions are more challenging to locate.
  - In an indirect call, the address of the target of the call can be stored in memory or in registers. In a direct call, the target is hardcoded.
  - A tail call is a call that appears at the end of a function. This may be optimised away by the compiler.

Control Flow Graph

- A *control flow gr*[redacted]G) is used to organize **the internals of a fu**[redacted]
- Useful for automated analysis, as well as manual analysis, especially for large functions.
- It provides a convenient graphical representation of the code structure, easier to get a feel on the structure at a glance.

Control Flow Graph

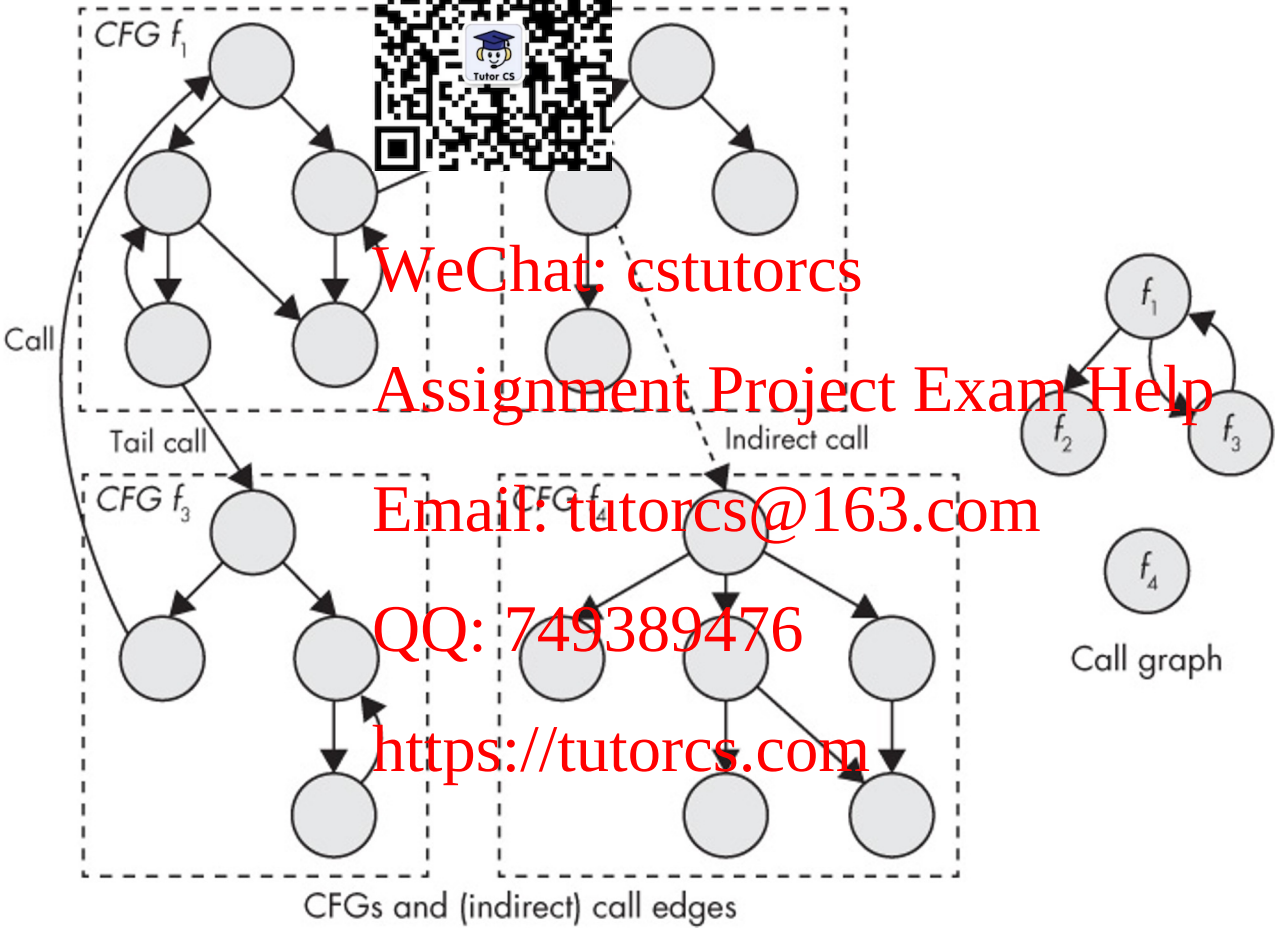
- Set of code blocks, ***basic blocks***.
  - The first instruction is the only entry point, and the last instruction is the only exit point.
- Blocks are connected by *branch* ***edges***, usually shown as arrows.
  - Edge from basic block A to B: last instruction in A may jump / fall through to the start of B.
- ***Call edges*** are **not** part of CFG.
- In practice, edges representing *indirect jumps* are often omitted.



Call Graphs

- Similar to CFGs, they show the relationship between call sites and functions rather than basic blocks.
- Indirect call is generally not shown in the graph.
- The address of a function may be stored in memory by an instruction.
  - That function is then called an *address-taken function*.
  - An address-taken function can potentially be called indirectly.
  - If a function's address is never taken and does not appear in any data sections, then we know it will never be called indirectly.

Call Graphs - Example



Object-Oriented Code

- Many of the binary analysis tools are targeting procedural languages like C.
  - Hence, mainly revolves around functions.
- But code can also be structured as classes / objects (OOP).
  - The exception handling mechanism can also be complex.
- In C++, a table of function pointers is used to resolve the actual function called.
  - Through indirect calls, hence harder to analyse.
- We do not go into the details of object-oriented code in this course.

Structuring Data

- Automatic data structure detection in stripped binaries is a notoriously difficult problem.
  - In general, disassemblers don't even attempt it.
  - Unless a reference to a data object is passed to a well-known function, then we can infer the data type.
  - Or maybe we can deduce based on the registers or the instructions used, e.g., floating-point registers.
- Some code can be compiled into similar instructions.
  - e.g., we can't differentiate between struct assignment and array element assignment.

Decompilation

- Reversing the compilation process.
- Attempt at guessing function signatures and local variables.
- Attempt at recovering control flows, e.g., if/else, loops, and function calls.
- Too error prone.
  - May not be reliable for further analysis.

Intermediate Representation

- Abstraction from I[illegible] machine languages.
  - e.g., Reverse E[illegible] Intermediate Language (REIL), LLVM bitcode, VEX IR (Valgrind).
- The goal is to automatically translate real machine code into an IR language that captures all of the machine code's semantics, but is much simpler to analyze.
  - e.g., REIL contains only 17 different instructions.
  - As opposed to doing analysis for each ISA, we can reuse analysis in the IR after lifting the ISAs to the same IR.
- Explicitly express all operations; no obscure side effects.
- Far less concise, can be unwieldy to read by humans.

Intermediate Representation - Example

Translating "add rax, r VEX IR:

```
IRSB {
    t0:Ity_I64 t1:Ity_I64 t2:Ity_I64 t3:Ity_I64
    00    |   ------ IMark(0x40339f, 3, 0) ------
    01    |   t2 = GET:I64(rax)
    02    |   t1 = GET:I64(rdx)
    03    |   t0 = Add64(t2,t1)
    04    |   PUT(cc_op) = 0x0000000000000004
    05    |   PUT(cc_dep1) = t2
    06    |   PUT(cc_dep2) = t1
    07    |   PUT(rax) = t0
    08    |   PUT(pc) = 0x00000000004033a2
    09    |   t3 = GET:I64(pc)
  NEXT: PUT(rip) = t3; Ijk_Boring
 }
```

Binary Analysis Properties

Binary Analysis Properties

- Broad classification of some properties of binary analysis.
  - Applicable to both control flow analysis and data flow analysis.
- What we will discuss:
  - Interprocedural and intraprocedural
  - Flow sensitivity
  - Context sensitivity

Interprocedural and Intraprocedural Analysis

- Intraprocedural: c[illegible]e code only within a single function at a time, [illegible]lyze CFG of each function in turn.
  - Maybe incomplete, e.g., a bug that is triggered by a specific combination of function calls.
- Interprocedural: consider an entire program as a whole, e.g., linking all the function CFGs together through call graph
  - Might take so long that the results won't matter anymore.
- Example: 10 functions with 10 if/else branches.
  - Intraprocedural will analyze $10 \times 1{,}024 = 10{,}240$ paths.
  - Interprocedural will analyze $1{,}024^{10} \sim 1.27 \times 10^{30}$ paths.

Example: Dead Code Elimination

- Intraprocedural analysis would not allow *code elimination* optimisation.
- For this we need interprocedural analysis to conclude the dead code is never reached.

```
#include <stdio.h>

static void
dead(int x)
{
  if(x == 5) {
    printf("Never reached\n");
  }
}
int
main(int argc, char *argv[])
{
  dead(4);
  return 0;
}
```

Flow Sensitivity

- Flow-sensitivity is the property that the analysis takes the order of the instructions into account.
- The analysis can be either flow-sensitive or flow-insensitive.
  - Flow-sensitive analysis is much more complex and also more computationally intensive than flow-insensitive analysis.
  - But it captures program behaviour better than its counterpart.
- Example:


```
x = unsigned_int(argv[0]) #  x ∈ [0,∞]
x = x + 5                 #  x ∈ [5,∞]
x = x + 10                #  x ∈ [15,∞]
```

Context Sensitivity

- Context sensitivity takes the order of function invocation into account.
  - Only meaningful in the context of interprocedural analysis.
- A different result for each possible path in the call graph.
  - Usually represented as a list of previously traversed functions.
- Due to computational complexity, generally the context is limited.

Context Sensitivity: Example

Control Flow Analysis: Loop Detection

- Detect loop in th
  - In the source co sy to spot loops via explicit keywords for loops such as `while` or `for`.
  - At the binary level, loops are implemented using jump instructions and harder to spot.
- Why loop detection?
  - Most of program's execution time is spent in loops, so they are interesting targets for optimisation.
  - Buffer overflow tends to occur in loops.

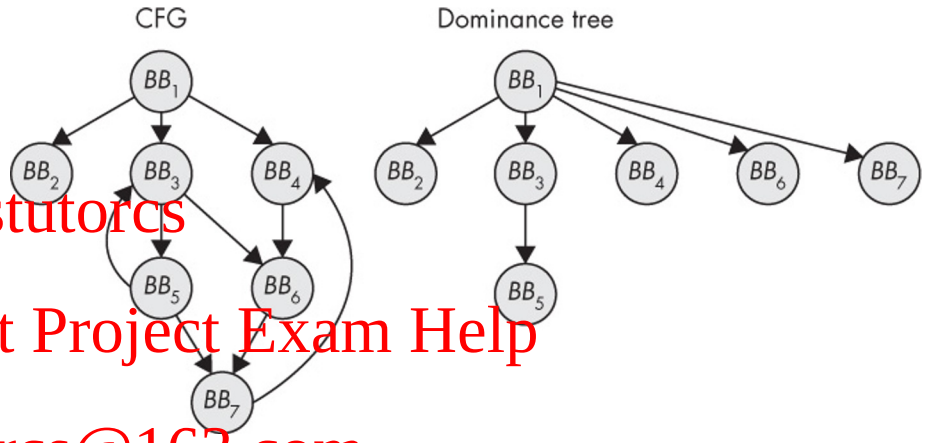
Control Flow Analysis: Loop Detection

- Compilers use *dominance trees* to determine a class of loops called *natural loops*.
- A dominance tree is constructed from a CFG.
  - A basic block A is said to dominate another basic block B if the only way to get to B from the entry point of the CFG is to go through A.
- A natural loop is induced by a back edge from a basic block B to A, where A dominates B.
  - This loop contains all basic blocks dominated by A from which there is a path to B.

Control Flow Analysis: Loop Detection

Example:

- BB_3 dominates BB_5 but not BB_6. Instead, BB_6 is dominated by BB_1.
- There's a natural loop spanning BB_3 and BB_5.
- The back edge from BB_7 to BB_4 induces a cycle, but not a natural loop.



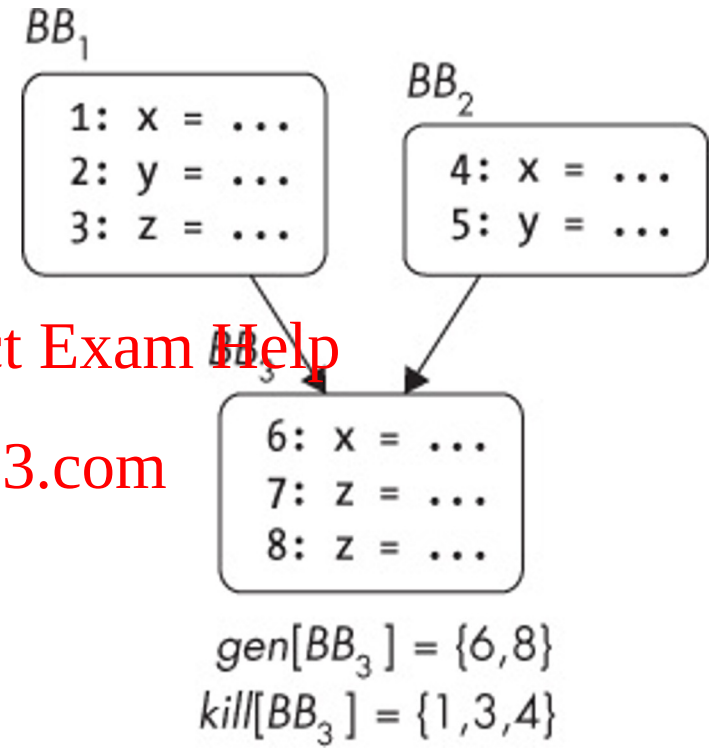
Control Flow Analysis: Cycle Detection

- Cycle detection is relatively easier compared to loop detection: just use depth-first search (DFS) on CFG.
- Dominance trees are not needed.
- Example: a DFS search to find cycles in the previous example.

```
 0:   [BB1]
 1:   [BB1,BB2]
 2:   [BB1]
 3:   [BB1,BB3]
 4:   [BB1,BB3,BB5]
 5:   [BB1,BB3,BB5,BB3]                *cycle found*
 6:   [BB1,BB3,BB5]
 7:   [BB1,BB3,BB5,BB7]
 8:   [BB1,BB3,BB5,BB7,BB4]
 9:   [BB1,BB3,BB5,BB7,BB4,BB6]
10:   [BB1,BB3,BB5,BB7,BB4,BB6,BB7]      *cycle found*
```

Data Flow Analysis: Reaching Definition Analysis

- "Which data definitions can reach this point in the program?"
  - Value assigned to a variable (or register, memory, etc.) can reach that point without being overwritten.
- Usually applied at CFG level – Individual basic blocks.
  - Which definitions a block *generates* and which it *kills*.



Data Flow Analysis: Reaching Definition Analysis

- The analysis starts with individual basic blocks.
- Local solutions for individual blocks are then combined to get a global solution
  - which definitions can reach the start of a basic block, and

$$in[B] = \bigcup_{p \in pred[B]} out[p]$$

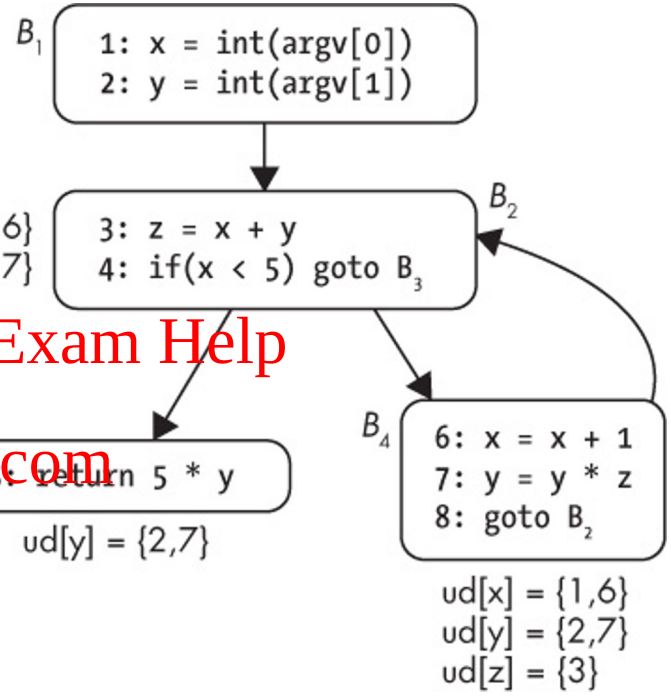
  - which can still be alive after the basic block.

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

- Since the definitions have mutual dependencies, it needs to be done iteratively until a stable state is reached.

Data Flow Analysis: Use-Def Chain

- When we encount[illegible]e of a variable in a pro[illegible]nt, this analysis will tell us where that variable may have been defined.
- Useful for decompilation, e.g., merging `cmp x, 5` and `jle` to `(x == 5)`.
- Or constant propagation.
- Quite straightforward to compute from the reaching-definition analysis.



Data Flow Analysis: Program Slicing

- Extract all instructions that contribute to the values of a chosen set of variables at a certain point in the program.
  - Also known as slicing criterion.
- Useful for debugging when we need to find out which parts of the code may be responsible for a bug, as well as reverse engineering.
- Computing slices can be complicated (still active research).
  - Have a look at angr (http://angr.io) which offers built-in slicing functionality.

Data Flow Analysis: Program Slicing

- Computed by tracking control and data flows to figure out program slices and remove them.
  - Because they make no difference to the eventual value of interest.
- Conditional guards (such as loop) are relevant when its body has relevant code.
- Originally it was a static analysis, but nowadays it's often applied to dynamic setting as well.
  - Because it produces smaller slices.


```
1:  x = int(argv[0])
2:  y = int(argv[1])
3:
4:  z = x +  y
5:  while(x  <  5) {
6:    x = x  + 1
7:    y = y  + 2
8:    z = z  + x
9:    z = z  + y
10:   z = z  *  5
11: }
12:
13: print(x)
14: print(y)
15: print(z)
```

Effects of Compiler Settings on Disassembly

- Optimized code is usually significantly harder to accurately disassemble (and therefore analyze) than non-optimized code.
- Corresponds less closely to the original source, e.g.,
  - `mul` and `div` instructions may be implemented using a series of bitshift and add operations.
  - Inlining of function calls to avoid the cost of call instructions.
  - Tail calls optimizations.
  - Additional padding bytes for memory alignment.
  - Unrolled loops.

Effects of Compiler Settings on Disassembly

- Data structure d[illegible] can be complicated by uses of the same base r[illegible] to index different arrays at the same time.
- Link-time optimization increases optimization surface and makes disassembly even more difficult.
- Binaries are increasingly compiled as position-independent code (PIC) to support ASLR.
- Binaries compiled with PIC are called position-independent executables (PIE).
  - Code and data are referenced relative to the program counter.

Summary

- Overview of inner workings of disassemblers.
- Overview of essential binary analysis techniques.
- Many techniques are intimately connected with how compilers work.
- Some of the dynamic analysis techniques (fuzzing and symbolic execution) will be covered in subsequent lectures.