CS451 – Software Analysis

Lecture 8
Disassembly and Binary Analysis Fundamentals (part 2)

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy

Structuring disassembled code and data

- No matter the techniques used for analysing a binary, it is useful to apply some structure
- Compared to high-level code, machine code is unstructured
- We can impose a structure which can benefit analysis
- Structure can be applied to both code and data

Structuring code

- Compartmentalization
  - Break the code in small logical connected parts, e.g., in *functions*
- Revealing control flow
  - Use control transfers to understand how different parts of code use other parts

Functions

- Most programming systems use functions to split the program's logic to a series of tasks
- Functions may not survive in machine code
  - For non-stripped binaries the start/end of each function is preserved
  - For stripped binaries, we need to identify the function boundaries with analysis: *function detection*
- Function signatures are used by most disassemblers
  - Scan the instruction stream for known patterns
  - Process target addresses of the call instruction
  - Scan for known prologues/epilogues (e.g., `leave; ret`)

Problems

- Compilers perform optimizations
  - Example, tail-call elimination
- Different compilers may use different signatures
- Some programming systems (e.g., Rust) have custom calling conventions

Non-optimized code

```
$ gcc -Wall tail-call.c -c -o tail-call.o
$ objdump  -d tail-call.o
Disassembly of section .text:
0000000000000000 <bar>:
   0: 55                     push   %rbp
   1: 48 89 e5               mov    %rsp,%rbp
   4: e8 00 00 00 00         callq  9 <bar+0x9>
   9: 5d                     pop    %rbp
   a: c3                     retq
000000000000000b <foo>:
   b: 55                     push   %rbp
   c: 48 89 e5               mov    %rsp,%rbp
   f: e8 00 00 00 00         callq  14 <foo+0x9>
  14: 5d                     pop    %rbp
  15: c3                     retq
0000000000000016 <main>:
  16: 55                     push   %rbp
  17: 48 89 e5               mov    %rsp,%rbp
  1a: 48 83 ec 10            sub    $0x10,%rsp
  1e: 89 7d fc               mov    %edi,-0x4(%rbp)
  21: 48 89 75 f0            mov    %rsi,-0x10(%rbp)
  25: e8 00 00 00 00         callq  2a <main+0x14>
  2a: c9                     leaveq
  2b: c3                     retq
```

```c
#include <stdlib.h>

int bar(void) {
    return rand();
}
int foo(void) {
    return bar();
}
int main(int argc, char *argv[]) {
    return foo();
}
```

Optimized code

```c
#include <stdlib.h>

int bar(void) {
    return rand();
}
int foo(void) {
    return bar();
}
int main(int argc, char *argv[]) {
    return foo();
}
```

```
$ gcc -Wall -O2 tail-call.c -c -o tail-call.o
$ objdump  -d tail-call.o

Disassembly of section .text:
0000000000000000 <bar>:
   0: e9 00 00 00 00       jmpq   5 <bar+0x5>
   5: 66 66 2e 0f 1f 84 00 data16 nopw %cs:0x0(%rax,%rax,1)
   c: 00 00 00 00

0000000000000010 <foo>:
  10: e9 00 00 00 00       jmpq   15 <foo+0x5>

Disassembly of section .text.startup:

0000000000000000 <main>:
   0: e9 00 00 00 00       jmpq   5 <main+0x5>
```

Optimized and stripped code

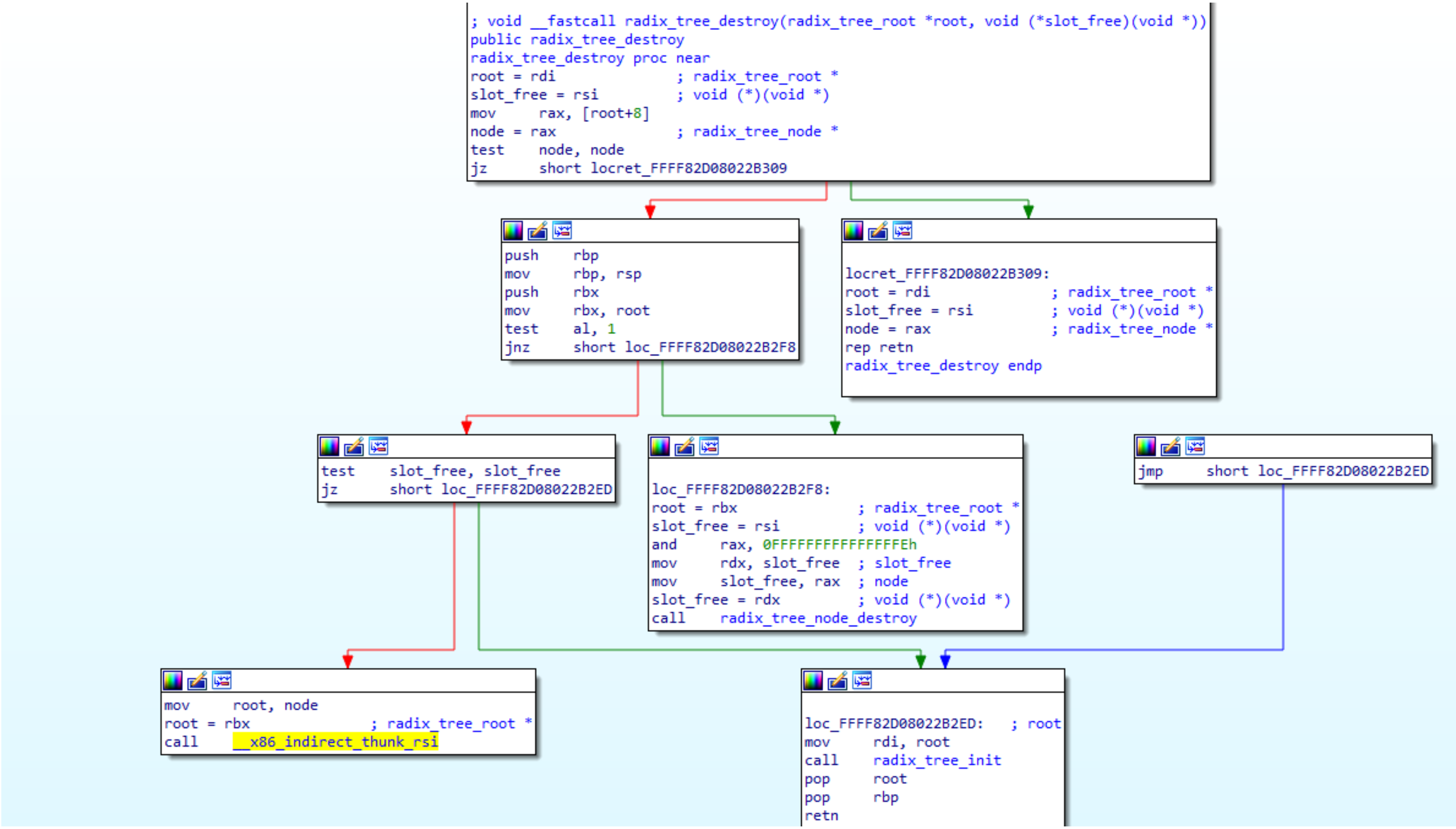
```
#include <stdlib.h>

int bar(void) {
    return rand();
}
int foo(void) {
    return bar();
}
int main(int argc, char *argv[]) {
    return foo();
}
```

```
$ objdump  -d tail-call.o
Disassembly of section .text:


0000000000000000 <.text>:
   0: e9 00 00 00 00       jmpq   0x5
   5: 66 66 2e 0f 1f 84 00 data16 nopw %cs:0x0(%rax,%rax,1)
   c: 00 00 00 00
  10: e9 00 00 00 00       jmpq   0x15

Disassembly of section .text.startup:

0000000000000000 <.text.startup>:
   0: e9 00 00 00 00       jmpq   0x5
```

Control-flow graphs

- A single function may be very complicated
  - Breaking to smaller blocks may be useful
- The control-flow graph (CFG) of a program can be computed by identifying basic blocks (BBs) that transfer control to other basic blocks
- This can be done at the machine-code level

Control-flow graph in IDA Pro



Call graphs

- Focused on the relationship between call sites and functions compared to CFGs that explore the control-flow between basic blocks
- Computation of a call graph is based on the function calls emitted by the machine code
- Sometimes it is hard to resolve indirect calls

Object-oriented code

- Machine code from compilers that utilize OO concepts can be complicated
- Exception handling is realized using indirect jumps
- Code is structured in objects, that contain code and data
  - Extracting class hierarchies in machine code is hard (see MARX: Uncovering Class Hierarchies in C++ Programs, Andre Pawloski, et all, in NDSS 2018)
- Virtual methods are dispatched using indirect jumps
  - Using pointers to VTables

Structuring data

- Data is much harder to be identified by disassemblers compared to code
- Sometimes it is possible
  - If the disassembler finds a call to send() can infer the types of the arguments, since send() has a known prototype
- Some primitive types can be inferred by the used registers
  - A floating-point register will contain a floating-point variable
  - `lodsb/stosb` manipulate parts of a string

Inferring data is hard

- Assignments of any type can produce the same machine code

```
ccf->user = pwrd->pwd_uid;
mov eax, DWORD PTR[rax+0x10]
mov DWORD PTR[rbx+0x60], eax
a[24] = b[4];
mov eax, DWORD PTR[rsi+0x10]
mov DWORD PTR[rdi+0x60], eax
```

Decompilation

- Decompilers attempt to reconstruct the high-level source from machine code
- The quality of the result is heavily related to the accuracy of the disassembly produced
- The code produced by decompilers is not very easy to read
  - Variable names are automatically chosen (v1, v2, f1(), f2(), etc.)

Intermediate representation

- Machine code is hard to be automatically analysed
  - Many instructions with complex semantics and side-effects (e.g., even a simple `add` will change the EFLAGS register)
- Sometimes it is useful to *lift* machine code to an intermediate representation (IR) form
  - LLVM (generic IR used by compilers), REIL and VEX IR (focused on reversing machine code)
- IR has a simpler instruction set and is more appropriate for automatic analysis
- Lifting machine code to IR is a difficult process

IR properties

- It is easier for an analysis to handle the semantics of a program expressed in IR
- It is harder for a human to read IR
  - Small set of simple instructions
  - Large sets of registers
  - Less concise, in general
- Performing the analysis at the IR level is done once
  - IR can then be transformed to any supported ISA (x86, ARM, etc.)

Binary analysis properties

- Interprocedural vs intraprocedural
  - Scope of analysis
- Flow sensitivity
  - Order in analyzed instructions is important
- Context sensitivity
  - Order of analyzed functions is important

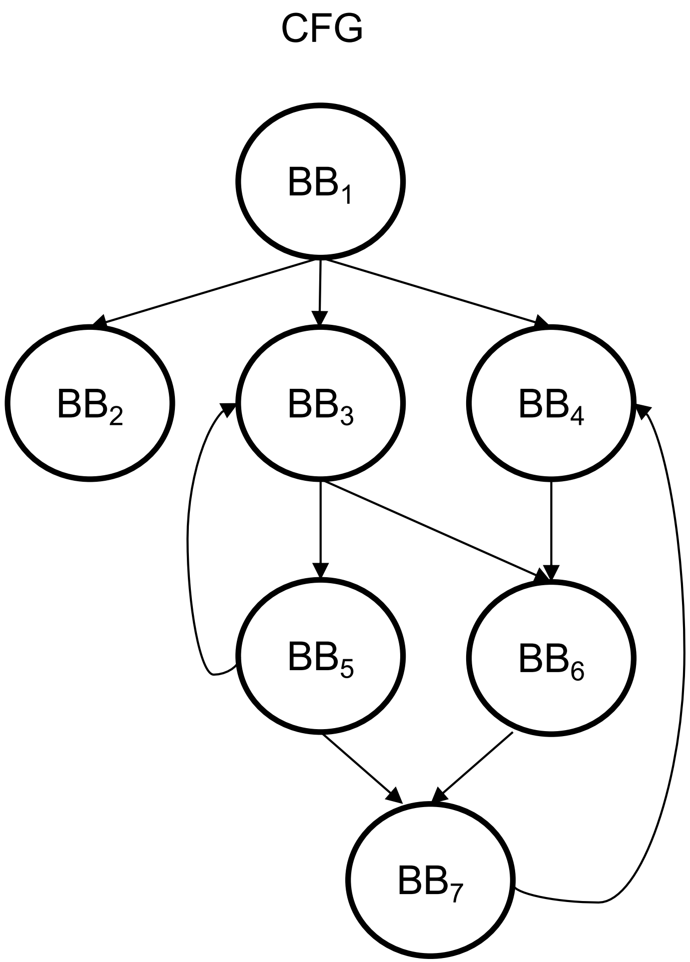
Interprocedural vs intraprocedural

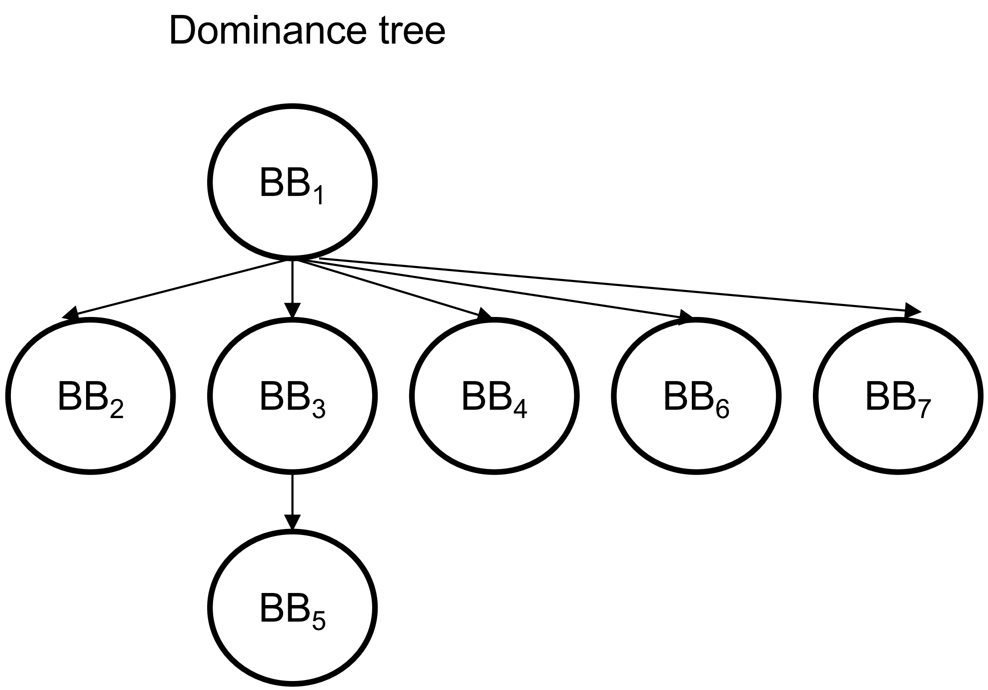
- Interprocedural analysis considers the entire program
  - Captures more complex interactions in the program
  - Can be infeasible for large programs
- Intraprocedural analysis considers a single function
  - Captures local interactions on a given function
  - The analysis is not complete, since functions usually interact with other functions

Control-flow analysis

- Loop detection
  - High-level code has very specific structures for constructing loops (`for {}`, `while {}`, etc .)
  - Machine code implements all loops using conditional branches
  - Loops are often the reason of a program's bottleneck, so identifying them is important
- Cycle detection
  - Programs may have a circular flow, not related to a natural loop, in particular
  - E.g., a function f1() may call f2(), and f2() may call f3(), and depending if a condition is met, f1() may be called again by f3()

Loop detection





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.

Natural loop: find a back edge from a basic block B to A, where A dominates B.

Cycle detection

- Compute the CFG
- Start a DFS from the entry node of the CFG
- Push each node that DFS is visiting in a stack
- Pop when the DFS backtracks
- If you push a node that is already in, then you have a circle

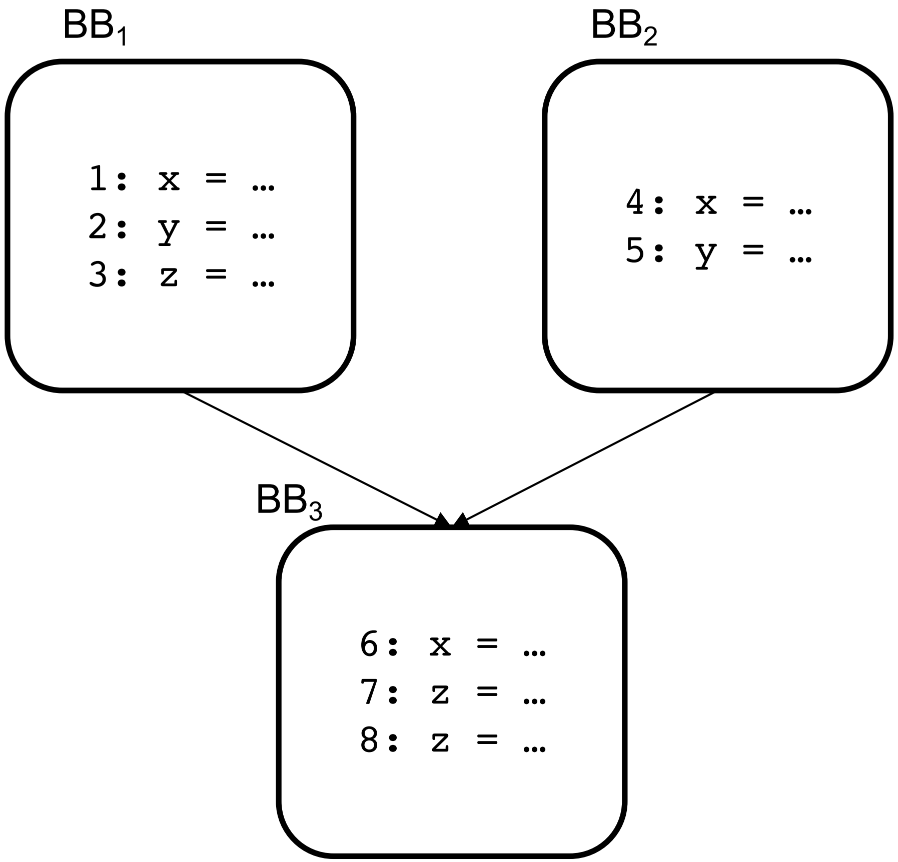
Example

- $[BB_1]$
- $[BB_1, BB_2]$
- $[BB_1]$
- $[BB_1, BB_3]$
- $[BB_1, BB_3, BB_5]$
- $[BB_1, BB_3, BB_5, BB_3]$ *cycle*
- …

Data-flow analysis

- Analysis may reason about data, as well
- Reaching definitions analysis
  - Which data definitions can reach this point in the program?
  - A value assigned to a variable (memory location, register) can reach a given point in the code, without being overwritten by another assignment along the way
- Use-def chains
  - Each time a variable is used, find the location of the related definition
- Program slicing
  - Find all instructions that contribute to the values of a set of variables at a certain point of a program

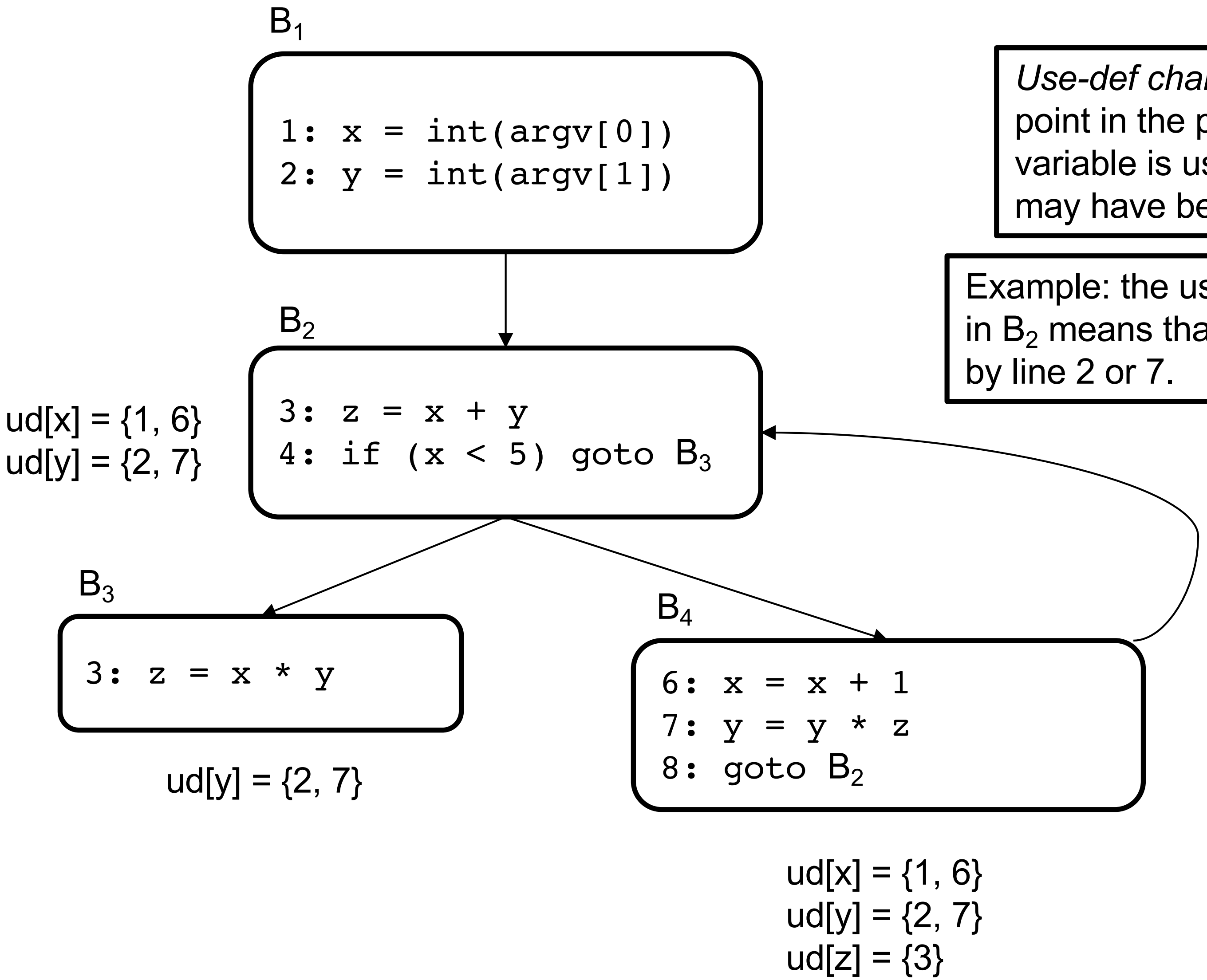
Reaching definitions



$gen[BB_3] = \{6, 8\}$
$kill[BB_3] = \{1, 3, 4\}$

For each basic block compute the definitions the block generates and kills.

Use-def chains



Use-def chains tell you, at each point in the program where a variable is used, where that variable may have been defined.

Example: the use-def chain ud[y] = {2, 7} in B_2 means that *y* has got its value either by line 2 or 7.

Program slicing

```
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)
```

Slicing is a data-flow technique that aims to extract all instructions that contribute to the values of a chosen set of variables at a certain point in the program (called the *slicing criterion*).

Example: using slicing to find the lines contributing to *y* in line 14.

Homework

- Reproduce slides 6, 7 and 8 with other test programs
  - Observe how an optimized program is disassembled using objdump, compared to the non optimized version
- Create a program with a natural loop and a cycle
  - Observe the disassembled machine code