

- **Attackers**: need to analyze our program to modify it!
- **Defenders**: need to analyze our program to protect it!
- Two kinds of analyses:
 - 1 *static analysis tools* collect information about a program by studying its code;
 - 2 *dynamic analysis tools* collect information from executing the program.

1/22

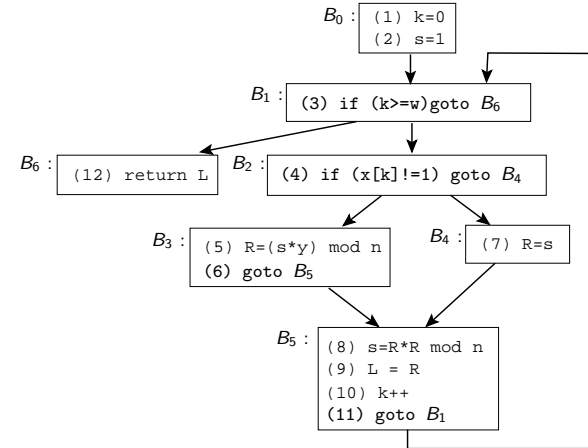
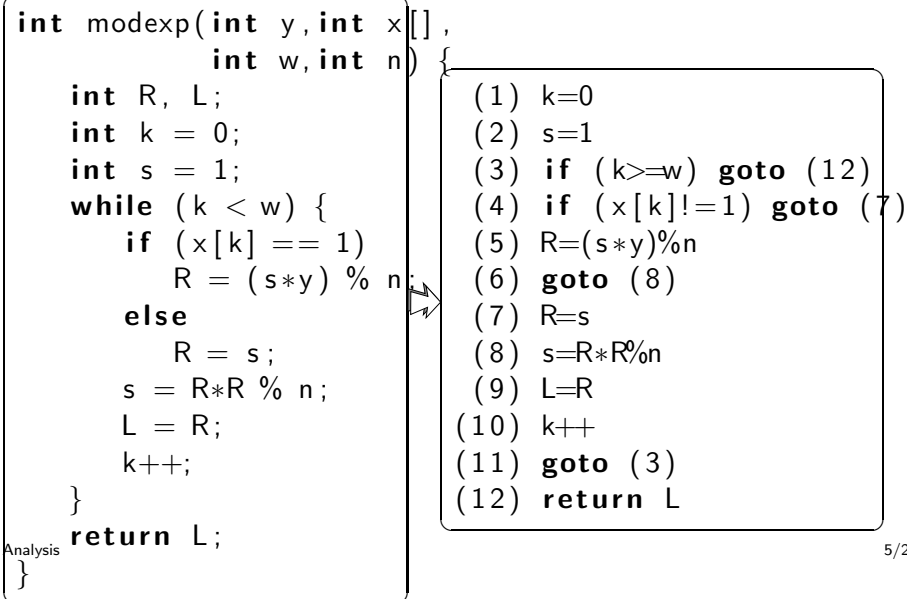
- **control-flow graphs**: representation of functions.
- **call graphs**: representation of (possible) function calls.
- **debugging**: what path does the program take?
- **tracing**: which functions/system calls get executed?
- **profiling**: what gets executed the most?
- **disassembly**: turn raw executables into assembly code.
- **decompilation**: turn raw assembly code into source code.

2/22

- 1 **Static Analysis**
 - Control-flow analysis
- 2 **Reconstituting source**
 - Disassembly

- A way to represent **functions**.
- Nodes are called **basic blocks**.
- Each block consists of straight-line code ending (possibly) in a branch.
- An edge $A \rightarrow B$: control could flow from A to B .

The resulting graph



Interprocedural control flow

BUILDCFG(F):

- 1 Mark every instruction which can start a basic block as a *leader*:
 - the first instruction is a leader;
 - any target of a branch is a leader;
 - the instruction following a conditional branch is a leader.
- 2 A basic block consists of the instructions from a leader up to, but not including, the next leader.
- 3 Add an edge $A \rightarrow B$ if A ends with a branch to B or can fall through to B . \square

- **Interprocedural analysis** also considers flow of information between functions.
- **Call graphs** are a way to represent **possible function calls**.
- Each **node** represents a function.
- An edge $A \rightarrow B$: A might call B .

Building call-graphs

```

void h ();

void f () {
    h ();
}

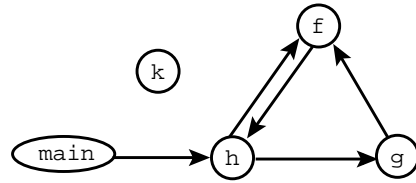
void g () {
    f ();
}

void h () {
    f ();
    g ();
}

void k () {}

int main () {

```

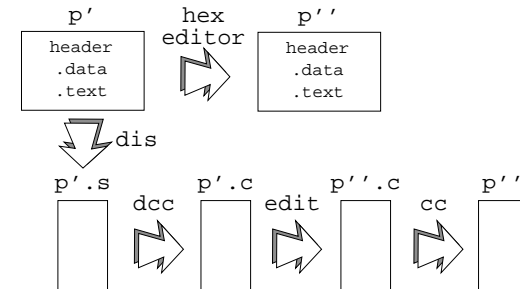
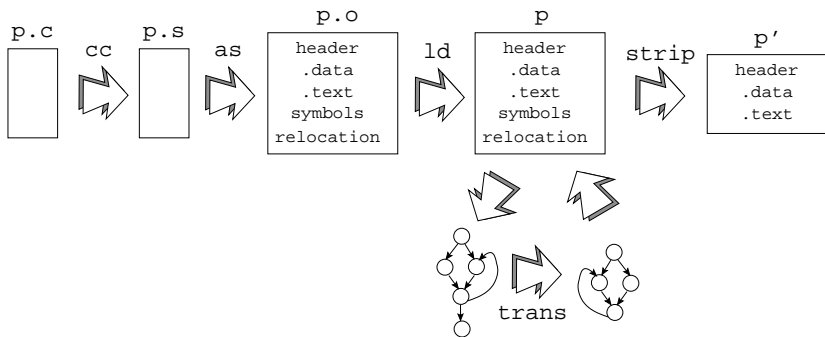


Outline

- 1 Static Analysis
 - Control-flow analysis
- 2 Reconstituting source
 - Disassembly

Reconstituting source

Attacking stripped binary code



Why is disassembly hard?

- **Variable length instruction sets** — overlapping instructions.
- **Mixing data and code** — misclassify data as instructions.
- **Indirect jumps** — must assume that *any* location could be the start of an instruction!
- Find the beginning of functions if all calls are indirect.
- Finding the end of functions — if no dedicated return instruction.
- **Handwritten assembly code** — won't conform to the standard calling conventions.
- **code compression** — the code of two functions may overlap.
- **Self-modifying code**.

Instruction set 1

opcode	mnemonic	operands	semantics
0	call	<i>addr</i>	function call to <i>addr</i>
1	calli	<i>reg</i>	function call to address in <i>reg</i>
2	brg	<i>offset</i>	branch to $pc + offset$ if flags for $>$ are set
3	inc	<i>reg</i>	$reg \leftarrow reg + 1$
4	bra	<i>offset</i>	branch to $pc + offset$
5	jmp	<i>reg</i>	jump to address in <i>reg</i>
6	prologue		beginning of function
7	ret		return from function

- Instruction set for a small architecture.
- All operators and operands are one byte long.
- Instructions can be 1-3 bytes long.

Instruction set 2

opcode	mnemonic	operands	semantics
8	load	$reg_1, (reg_2)$	$reg_1 \leftarrow [reg_2]$
9	loadi	reg, imm	$reg \leftarrow imm$
10	cmpi	reg, imm	compare <i>reg</i> and <i>imm</i> and set flags
11	add	reg_1, reg_2	$reg_1 \leftarrow reg_1 + reg_2$
12	brge	<i>offset</i>	branch to $pc + offset$ if flags for \geq are set
13	brq	<i>offset</i>	branch to $pc + offset$ if flags for $=$ are set
14	store	$(reg_1), reg_2$	$[reg_1] \leftarrow reg_2$

Disassembly — example

```
6 0 10 9 0 43 1 0 7 0 6 9 0 1 10 0 1 2 2 6 9
1 30 11 1 0 8 2 1 5 2 32 37 9 1 3 4 7 9 1 4
4 2 7 6 9 0 3 7 6 9 0 1 7 42 2 4 3 1 7 4
3 4 1
```

- Next few slides show the results of different disassembly algorithms.
- Correctly disassembled regions are in pink.

```

main: # ORIGINAL PROGRAM
0: [6]      prologue
1: [0,10]   call    foo
3: [9,0,43] loadi   r0,43
6: [1,0]    calli   r0
8: [7]      ret
9: [0]      .align  2
foo:
10:[6]     prologue
11:[9,0,1] loadi   r0,1
14:[10,0,1] cmpi   r0,1
17:[2,26]  brg    26
19:[9,1,30] loadi  r1,30
22:[11,1,0] add    r1,0
25:[8,2,1] load  r2,(r1)
28:[5,2]   jmp    r2
30:[32]    .byte  32
31:[37]    .byte  37
32:[9,1,3] loadi  r1,3
35:[4,7]   bra    7
37:[9,1,4] loadi  r1,4
40:[4,2]   bra    2

```

```

bar:
43:[6]     prologue
44:[9,0,3] loadi   r0,3
47:[7]     ret
baz:
48:[6]     prologue
49:[9,0,1] loadi   r0,1
52:[7]     ret
life:
53:[42]    .byte  42
fred:
54:[2,4]   brg    4
56:[3,1]   inc    r1
58:[7]     ret
59:[4,3]   bra    3
61:[4,1]   bra    1

```

```

# LINEAR SWEEP DISASSEMBLY
0: [6]      prologue
1: [0,10]   call    10
3: [9,0,43] loadi   r0,43
6: [1,0]    calli   r0
8: [7]      ret
9: [0,6]    call    6
11:[9,0,1] loadi   r0,1
14:[10,0,1] cmpi   r0,1
17:[2,26]  brg    26
19:[9,1,30] loadi  r1,30
22:[11,1,0] add    r1,0
25:[8,2,1] load  r2,(r1)
28:[5,2]   jmp    r2
30:[32]    ILLEGAL 32
31:[37]    ILLEGAL 37
32:[9,1,3] loadi  r1,3
35:[4,7]   bra    7
37:[9,1,4] loadi  r1,4
40:[4,2]   bra    2
42:[7]     ret

```

```

43:[6]     prologue
44:[9,0,3] loadi   r0,3
47:[7]     ret
48:[6]     prologue
49:[9,0,1] loadi   r0,1
52:[7]     ret
53:[42]    ILLEGAL 42
54:[2,4]   brg    4
56:[3,1]   inc    r1
58:[7]     ret
59:[4,3]   bra    3
61:[4,1]   bra    1

```

```

f0: # RECURSIVE TRAVERSAL
0: [6]      prologue
1: [0,10]   call    10
3: [9,0,43] loadi   r0,43
6: [1,0]    calli   r0
8: [7]      ret
9: [0]      .byte  0
f10:
10:[6]     prologue
11:[9,0,1] loadi   r0,1
14:[10,0,1] cmpi   r0,1
17:[2,26]  brg    26
19:[9,1,30] loadi  r1,30
22:[11,1,0] add    r1,0
25:[8,2,1] load  r2,(r1)
28:[5,2]   jmp    r2
30:[32]    .byte  32
31:[37]    .byte  37

```

```

32:[9,1,3] loadi  r1,3
35:[4,7]   bra    7
37:[9,1,4] loadi  r1,4
40:[4,2]   bra    2
42:[7]     ret
43:[6]     prologue
44:[9,0,3] loadi  r0,3
47:[7]     ret
48:[6]     .byte  6
49:[9]     .byte  9
50:[0]     .byte  0
51:[1]     .byte  1
52:[7]     .byte  7
53:[42]    .byte  42
54:[2]     .byte  2
59:[4]     .byte  4
60:[3]     .byte  3
61:[4]     .byte  4
62:[1]     .byte  1

```

Algorithm REHM

- Extends the standard recursive traversal algorithm with a collection of heuristics to increase precision.
- First, **follow all branches** and returns a set of function start addresses and a set of decoded addresses.
- Next, try to decode any remaining undecoded bytes by looking for **prologue instructions** that could start a function.
- Next, try to build a **reasonable control flow graph** from the remaining undecoded bytes.
- Reasonable CFG: “there are no jumps into the middle of another instruction and the resulting function contains at least two control transfer instruction.”

Algorithm REHM

f0: # HARRIS/MILLER			
0: [6]	prologue		
1: [0,10]	call	10	
3: [9,0,43]	loadi	r0	43
6: [1,0]	calli	r0	
8: [7]	ret		
9: [0]	.byte	0	
f10:			
10: [6]	prologue		
11: [9,0,1]	loadi	r0	1
14: [10,0,1]	cmpi	r0	1
17: [2,26]	brg	26	
19: [9,1,30]	loadi	r1	30
22: [11,1,0]	add	r1, r0	
25: [8,2,1]	load	r2, (r1)	
28: [5,2]	jmp	r2	
30: [32]	.byte	32	
31: [37]	.byte	37	
32: [9,1,3]	loadi	r1	3
35: [4,7]	bra	7	
37: [9,1,4]	loadi	r1	4
40: [4,2]	bra	2	
42: [7]	ret		
f43:			
43: [6]	prologue		
44: [9,0,3]	loadi	r0	3
47: [7]	ret		
f48:			
48: [6]	prologue		
49: [9,0,1]	loadi	r0	1
52: [7]	ret		
53: [42]	.byte	42	
f54:			
54: [2,4]	brg	4	
56: [3,1]	inc	r1	
58: [7]	ret		
59: [4]	.byte	4	
60: [3]	.byte	3	
61: [4]	.byte	4	
62: [1]	.byte	1	

- Function f43 is only called indirectly, function f48 isn't called at all — the disassembler still finds them by searching for their prologue instructions.
- The disassembler next starts at location 53, realizes that 42 isn't a valid opcode, moves to location 54, builds a valid CFG.
- The algorithm recovered 95.6% of all functions over a set of Windows and Linux programs.