Software Protection: How to Crack Programs, and **Defend Against Cracking** Lecture 3: Program Analysis Moscow State University, Spring 2014 Christian Collberg University of Arizona

Last week's lecture

 What form does the program take that that the adversary gets to attack?

Last week's lecture

- What form does the program take that that the adversary gets to attack?
- List some attack techniques!

In-Class Exercise

- Alice writes a program that she only wants Bob to execute 5 times.
- At the end of each run, the program writes a file .AliceSecretCount with the number of runs so far.
- At the beginning of each run, the program reads the file .AliceSecretCount and, if the number of runs so far is ≥ 5, it exits with an error message BAD BOB!.
- Draw a detailed attack tree with all attacks available to Bob!

Today's lecture

- Program analysis
- Control flow analysis
- Disassembly
- Occompilation
- Self-modifying code



Program Analysis



Defenders analyze their program to protect it!



Attackers analyze our program to modify it!

Program Analysis

- Attackers: need to analyze our program to modify it!
- Defenders: need to analyze our program to protect it!
- Two kinds of analyses:
- static analysis tools collect information about a program by studying its code;

 - Q dynamic analysis tools collect information from executing the program.





 control-flow graphs: representation of (possible) control-flow in functions.

- control-flow graphs: representation of (possible) control-flow in functions.
- call graphs: representation of (possible) function calls.

- control-flow graphs: representation of (possible) control-flow in functions.
- call graphs: representation of (possible) function calls.
- disassembly: turn raw executables into assembly code.

- control-flow graphs: representation of (possible) control-flow in functions.
- call graphs: representation of (possible) function calls.
- disassembly: turn raw executables into assembly code.
- decompilation: turn raw assembly code into source code.



Dynamic Analyses

debugging: what path does the program take?

Dynamic Analyses

- debugging: what path does the program take?
- tracing: which functions/system calls get executed?

Dynamic Analyses

- debugging: what path does the program take?
- tracing: which functions/system calls get executed?
- profiling: what gets executed the most?



Control-Flow Graphs

Control-flow Graphs (CFGs)

- A way to represent the possible flow of control inside a function.
- Nodes are called basic blocks.
- Each block consists of straight-line code ending (possibly) in a branch.
- An edge A → B: control could flow from A to B.

```
int modexp(int y, int x[])
           int w, int n)
   int R, L;
   int k = 0;
   int s = 1;
   while (k < w) {
      if (x[k] == 1)
         R = (s * y) % n;
      else
        R = s;
      s = R * R % n;
      L = R;
      k++;
   return L;
```

```
(1) k=0
(2) s=1
(3) if (k \ge w) goto (12)
(4) if (x[k]!=1) goto (7)
(5) R=(s*y)%n
(6) goto (8)
 (7) R=s
 (8) s=R∗R%n
(9) L=R
(10) k++
(11) goto (3)
(12) return L
```

The resulting graph



Step 1: Generate Three-Address Statements

 Compile the function into a sequence of simpler statements:

•
$$x = y + z$$

● goto L

- These are called three-address statements.
- Other representations are possible, for example expression trees.

Step 2: Build the graph

BUILDCFG(F):



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.



A basic block consists of the instructions from a leader up to, but not including, the next leader.



Add an edge $A \rightarrow B$ if A ends with a branch to B or can fall through to B.

In-Class Exercise I

```
int gcd(int x, int y) {
    int temp;
    while (true) {
        if (x%y == 0) break;
        temp = x%y;
        x = y;
        y = temp;
    }
}
```

- Turn this function into a sequence of three-address statements.
- Turn the sequence of simplified statements into a CFG.

In-Class Exercise II

(1)
$$X := 20$$

(2) **if** $X \ge 10$ **goto** (8)
(3) $X := X-1$
(4) $A[X] := 10$
(5) **if** $X <> 4$ **goto** (7)
(6) $X := X-2$
(7) **goto** (2)
(8) $Y := X+5$

Construct the corresponding CFG.



Call Graphs

Interprocedural control flow

- Interprocedural analysis 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() { h(); **void** (*p)() = &k; p(); }



In-Class Exercise

 Build the call graph for the Java program on the next slide.

```
class M {
   public void a () {System.out.println("hello");}
  public void b () {}
  public void c () {System.out.println("world!")}
class N extends M {
   public void a () {super.a();}
  public void b () {this.b(); this.c();}
  public void c () {}
class Main {
   public static void main (String args[]) {
       M x = (args.length > 0)? new M() : new N();
       x.a();
       N y = new N(); y.b();
```



Disassembly





Instruction Set

- On the next slide you will see an instruction set for a small architecture.
- All operators and operands are one byte long.
- Instructions can be 1-3 bytes long.

Instruction set 1

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

opcode	mnemonic	operands	semantics
8	load	$reg_1, (reg_2)$	$reg_1 \leftarrow [reg_2]$
9	loadi	reg,imm	$\textit{reg} \leftarrow \textit{imm}$
10	cmpi	reg,imm	compare <i>reg</i> and <i>imm</i> and set flags
11	add	<i>reg</i> ₁ , <i>reg</i> ₂	$reg_1 \leftarrow reg_1 + reg_2$
12	brge	offset	branch to $pc + offset$ if flags for \geq are set
13	breq	offset	branch to $pc + offset$ if flags for = are set
14	store	$(reg_1), reg_2$	$[reg_1] \leftarrow reg_2$

Disassembly — example



- 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,r0		
25:[8,2,1]	load r2,	(r1)		
28:[5,2]	jmpi	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			

```
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]
                1
        bra
```

# LINEAR SWE	EEP DISASS	SEMBLY
0: [6]	prologue	
1: [0,10]	call	10
3: [9,0,43]	loadi	r0,43
6: [1,0]	calli	rO
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,r0
25:[8,2,1]	load r2,	(r1)
28:[5,2]	jmpi	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	rl
58:[7]	ret	
59:[4,3]	bra	3
61:[4,1]	bra	1
\		

			C		
f0: # RECURS	SIVE TRAVI	ERSAL	32:[9,1,3]	loadi	r1,3
0: [6]	prologue		33:[4,7]	DIA	1
1: [0,10]	call	10	37:[9,1,4]	loadi	r1,4
3: [9.0.43]	loadi	r0.43	40:[4,2]	bra	2
6 · [1 0]	calli	r0	42:[7]	ret	
0. [1,0]	Calli	10	43:[6]	prologue	
0:[/]	rec		44:[9,0,3]	loadi	r0,3
			47:[7]	ret	
9: [0]	.byte	0			
			48.[6]	byte	6
f10:			10.[0]	byte	9
10:[6]	prologue			.byce	0
11:[9,0,1]	loadi	r0,1		.byte	1
14:[10,0,1]	cmpi	r0.1	51:[1]	.byte	1
17:[2.26]	bra	26	52:[7]	.byte	7
19.[9 1 30]	loadi	r1 30	53:[42]	.byte	42
22.[11 1 0]	add	r1,50	54:[2]	.byte	2
22:[11,1,0]	auu	L1,L0			
25:[8,2,1]	load r2	, (r1)	59:[4]	.bvte	4
28:[5,2]	Jmbī	r2	60:[3]	.bvte	3
30:[32]	.byte	32	61 • [4]	byte	- 4
31:[37]	.byte	37	62.[1]	buto	1
			/ ဎ∠・[⊥]	.byte	1

Exercise



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

opcode	mnemonic	operands	semantics
8	load	$reg_1, (reg_2)$	$reg_1 \leftarrow [reg_2]$
9	loadi	reg,imm	$\textit{reg} \leftarrow \textit{imm}$
10	cmpi	reg,imm	compare <i>reg</i> and <i>imm</i> and set flags
11	add	<i>reg</i> ₁ , <i>reg</i> ₂	$reg_1 \leftarrow reg_1 + reg_2$
12	brge	offset	branch to $pc + offset$ if flags for \geq are set
13	breq	offset	branch to $pc + offset$ if flags for = are set
14	store	$(reg_1), reg_2$	$[reg_1] \leftarrow reg_2$

 Variable length instruction sets overlapping instructions.

- Variable length instruction sets overlapping instructions.
- Mixing data and code misclassify data as instructions.

- 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!

- 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 fuctions — if no dedicated return instruction.

- Finding the end of fuctions if no dedicated return instruction.
- Handwritten assembly code won't conform to the standard calling conventions.

- Finding the end of fuctions 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.

- Finding the end of fuctions 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.



Decompilation













```
void foo() {
    x=1;
    while (x<10} {
        if (x<5)
            x++;
        else
            x+=2;
        printf();
    }
}</pre>
```

 Disassembly — first step of any decompiler!

- Disassembly first step of any decompiler!
- Target language assembly code may not correspond to any legal source code.

- Disassembly first step of any decompiler!
- Target language assembly code may not correspond to any legal source code.
- Standard library functions (call printf() ⇒ call foo96()).

- Disassembly first step of any decompiler!
- Target language assembly code may not correspond to any legal source code.
- Standard library functions (call printf() ⇒ call foo96()).
- Idioms of different compilers (xor r0, r0⇒r0=0).

• Artifacts of the target architecture (unnecessary jumps-to-jumps).

- Artifacts of the target architecture (unnecessary jumps-to-jumps).
- Structured control-flow from mess of machine code branches.

- Artifacts of the target architecture (unnecessary jumps-to-jumps).
- Structured control-flow from mess of machine code branches.
- Compiler optimizations undo loop unrolling, shifts and adds ⇒ original multiplication by a constant.

- Artifacts of the target architecture (unnecessary jumps-to-jumps).
- Structured control-flow from mess of machine code branches.
- Compiler optimizations undo loop unrolling, shifts and adds ⇒ original multiplication by a constant.
- Loads/stores ⇒ operations on arrays, records, pointers, and objects.



Self-Modifying Code

Abnormal Programs

- In a "normal" program, the code segment doesn't change.
- But, the programs in this course are not normal!
- Programs which change the code segment are called self-modifying.

Example

0:	[9,0,12]	loadi	r0,12
3:	[9,1,4]	loadi	r1,4
6:	[14,0,1]	store	(r0),r1
9:	[11,1,1]	add	r1,r1
12:	[3,4]	inc	r4
14:	[4,-5]	bra	-5
16:	[7]	ret	

Instruction set 1

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

Instruction set 2

opcode	mnemonic	operands	semantics
8	load	$reg_1, (reg_2)$	$reg_1 \leftarrow [reg_2]$
9	loadi	reg,imm	$\textit{reg} \leftarrow \textit{imm}$
10	cmpi	reg,imm	compare <i>reg</i> and <i>imm</i> and set flags
11	add	<i>reg</i> ₁ , <i>reg</i> ₂	$reg_1 \leftarrow reg_1 + reg_2$
12	brge	offset	branch to $pc + offset$ if flags for \geq are set
13	breq	offset	branch to $pc + offset$ if flags for = are set
14	store	$(reg_1), reg_2$	$[reg_1] \leftarrow reg_2$
Building the CFG is simple!



Building the CFG is simple!



- The CFG isn't even connected!
- The backwards branch at position 14 forms an infinite loop!

Look again!

0:	[9,0,12]	loadi	r0,12
3:	[9,1,4]	loadi	r1,4
6:	[14,0,1]	store	(r0),r1
9:	[11,1,1]	add	r1,r1
12:	[3,4]	inc	r4
14:	[4,-5]	bra	-5
16:	[7]	ret	

 The store instruction is writing the byte 4 to position 12, changing the inc r4 instruction into a bra 4!

Actual CFG

• The actual control flow graph looks like this:



Actual CFG

The actual control flow graph looks like this:



 If the codebytes are changing at runtime, a standard control flow graph isn't sufficient.

New CFG Model

- Add a codebyte data structure to the graph that represents all the different states each instruction can be in.
- Add conditions to the edges; only if the condition on an edge is true can control take that path.



New CFG Model...



New CFG Model...

- **The** add instruction comprises three bytes (11, 1, 1) at addresses 9-11.
- Code byte addresses are in pink and the code bytes themselves in blue.
- At location 12, two values can be stored, 3 and 4.
- The outgoing edges from add's basic block are conditional on what is stored at 12, either 3 or 4.

New CFG Model...

- Nice representation of a self-modifying function!
- 2 But, hard to build in practice.
- Omputer viruses are often self-modifying.
- We will see self-modifying protection algorithms later in the course.

Next week's lecture

Obfuscation algorithms Please check the website for important announcements:

www.cs.arizona.edu/~collberg/

Teaching/mgu/2014