Software Protection Research

ISSISP 2017 — Program Analysis

Christian Collberg

Department of Computer Science
University of Arizona

http://collberg.cs.arizona.edu

collberg@gmail.com

Supported by NSF grants 1525820 and 1318955 and by the private foundation that shall not be named
What is Program Analysis?

Control Flow Analysis

Discussion
What is Program Analysis?
int foo() {
    int x;
    int* y;
    printf(x+*y);
}

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?
• **Defenders:** need to analyze their program to protect it!

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

int foo() {
    ...
}

Tigress

Obfuscate

int foo'() {
    ...
}
• **Attackers**: need to analyze our program to modify it!

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

```c
int foo() {
    ...
}
```

- Extract Code!
- Discover Algorithms!
- Find Design!
- Find Keys!
- Modify Code!
Two kinds of analyses:

- **static analysis**: collect information about a program by studying its code;

- **dynamic analysis**: collect information from *executing* the program.
• **static analysis**: collect information about a program by studying its code

```c
int foo() {
    ...
}
```

I.e. we analyze the source or binary code of the program itself.

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?
**Dynamic analysis:** collect information from *executing* the program.

I.e. we analyze a *trace* of the program as it is running on some *particular input*.

- Who calls `foo`?
- Who does `foo` call?
- Is `x` ever initialized?
- Can `y` ever be null?
- What will `foo` print?
Static Analyses

- **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?
- **tracing**: which functions/system calls get executed?
- **profiling**: what gets executed the most?
Control Flow Analysis
Control-Flow Graph (CFG)

• A way to represent the possible flow of control inside a function.

• **Nodes**: called **basic blocks**. Each block consists of straight-line code ending (possibly) in a branch.

• **Edges**: An edge $A \rightarrow B$ means that control could flow from $A$ to $B$.

• There is one unique **entry node** and one unique **exit node**.
int foo() {
    printf("Boo!");
}
printf("Boo!");
int foo() {
    x=1;
    y=2;
    printf(x+y);
}

ENTRY

x=1;
y=2;
printf(x+y);

EXIT
int foo() {
    read(x);
    if (x>0) 
        printf(x);
}
int foo() {
    read(x);
    if (x>0)
        printf(x);
    else
        printf(x+1);
}

ENTRY

B0:

B1: x=1;
    if (x>0) goto B2

B2: printf(x);

B3: EXIT

B4: printf(x+1);
```c
int foo() {
    x=10;
    while (x>0){
        printf(x);
        x=x-1;
    }
}
```
1. Mark every instruction which can start a basic block as a leader:
   1. the first instruction
   2. a target of a branch
   3. any instruction following a conditional branch

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

3. Add an edge A→B if A ends with a branch to B or can fall through to B.
Exercise!

x ← 20;
while (X<10){
    X ← X-1;
    A[X] ← 10;
    if (X=4)
        X ← X-2;
}
Y ← X+5;

1: x←20
2: if x>=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

Convert to CFG!
First simplify!

Work with your friends!!!
Disassembly
• **Attackers**: prefer looking at assembly code than machine code

```c
int foo() {
    ...
}
```

`foo.c`

`foo.exe`

---

```
add r1,r2,r3
ld r2,[r3]
call bar
cmp r1,r4
bgt L2
```

Disassemble
Static Disassembly

objdump -d i/bin/ls | less
<table>
<thead>
<tr>
<th>Address</th>
<th>Code (bytes)</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xd78</td>
<td>55 48 89 e5 48 83 c7</td>
<td>push %rbp</td>
</tr>
<tr>
<td>0xd79</td>
<td>48 89 e5</td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td>0xd7c</td>
<td>48 83 c7 68</td>
<td>add $0x68,%rdi</td>
</tr>
<tr>
<td>0xd80</td>
<td>48 83 c6 68</td>
<td>add $0x68,%rsi</td>
</tr>
<tr>
<td>0xd84</td>
<td>5d</td>
<td>pop %rbp</td>
</tr>
<tr>
<td>0xd85</td>
<td>e9 26 38 00 00</td>
<td>jmpq 1000045b0</td>
</tr>
<tr>
<td>0xd8a</td>
<td>55</td>
<td>push %rbp</td>
</tr>
<tr>
<td>0xd8b</td>
<td>48 89 e5</td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td>0xd8e</td>
<td>48 8d 46 68</td>
<td>lea 0x68(%rsi),%rax</td>
</tr>
<tr>
<td>0xd92</td>
<td>48 8d 77 68</td>
<td>lea 0x68(%rdi),%rsi</td>
</tr>
<tr>
<td>0xd96</td>
<td>48 89 c7</td>
<td>mov %rax,%rdi</td>
</tr>
<tr>
<td>0xd99</td>
<td>5d</td>
<td>pop %rbp</td>
</tr>
<tr>
<td>0xd9a</td>
<td>e9 11 38 00 00</td>
<td>jmpq 1000045b0</td>
</tr>
<tr>
<td>0xd9f</td>
<td>55</td>
<td>push %rbp</td>
</tr>
</tbody>
</table>
• Disassembly is hard! And sometimes disassemblers get it wrong!

• In general, this is always the case: program analysis is more or less precise.
There are two general algorithm ideas for disassembly:

1. Linear Sweep Traversal
2. Recursive Traversal

At times, both with fail.

We typically add heuristics to improve precision.
1. `push %rbp`
2. `mov %rsp,%rbp`
3. `add $0x68,%rdi`
4. `add $0x68,%rsi`
5. `pop %rbp`
6. `jmpq 1000045b0`
7. `push %rbp`
1. **0xd78**: push %rbp
2. **0xd79**: mov %rsp,%rbp
3. **0xd7c**: add $0x68,%rdi
4. **0xd80**: add $0x68,%rsi
5. **0xd84**: pop %rbp
6. **0xd85**: jmpq 0x45b0
7. **0xd8a**: push %rbp

Recursive traversal disassembly

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xd8a</td>
</tr>
<tr>
<td>0x45b0</td>
</tr>
</tbody>
</table>
Exercise!

1. 0xd78: push %rbp
2. 0xd79: mov %rsp,%rbp
3. 0xd7c: add $0x68,%rdi
4. 0xd80: add $0x68,%rsi
5. 0xd84: pop %rbp
6. 0xd85: jmpq 0x45b0
7. 0xd8a: .byte 0x55
8. 0xd8b: mov %rdi,%rbp

0x55 ≡ push %rbp!!!

• How would a **linear sweep** disassembly handle this code?
Exercise!

1. 0xd78: push %rbp
2. 0xd79: mov %rsp,%rbp
3. 0xd7c: add $0x68,%rdi
4. 0xd80: add $0x68,%rsi
5. 0xd84: pop %rbp
6. 0xd85: jmpr %rdi
7. 0xd8b: mov %rdi,%rbp

Indirect jump!

• How would a **recursive traversal** disassembly handle this code?
Questions?