Man-At-The-End Attacks and Defenses I

Lecture #4

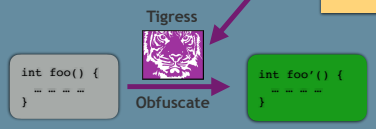
Christian Collberg
University of Arizona

Program Analysis

```
int foo() {
  int x;
  int* y;
  printf(x+*y);
}
```

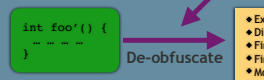
- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

- *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?

- *Attackers*: need to analyze our program to modify it!



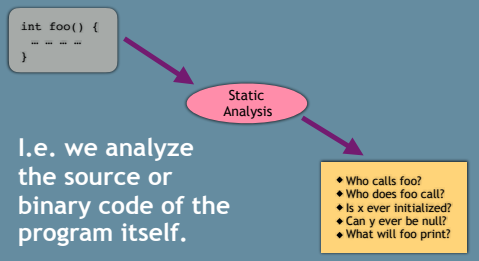
- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

- Extract Code!
- Discover Algorithms!
- Find Design!
- Find Keys!
- Modify Code!

Two kinds of analyses:

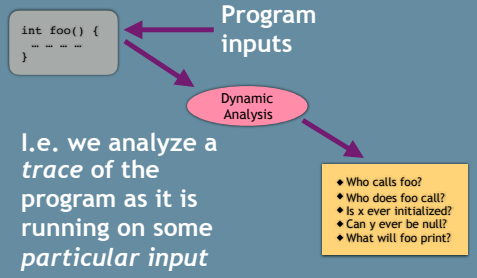
- **static analysis:** collect information about a program by studying its code;
- **dynamic analysis:** collect information from *executing* the program.

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



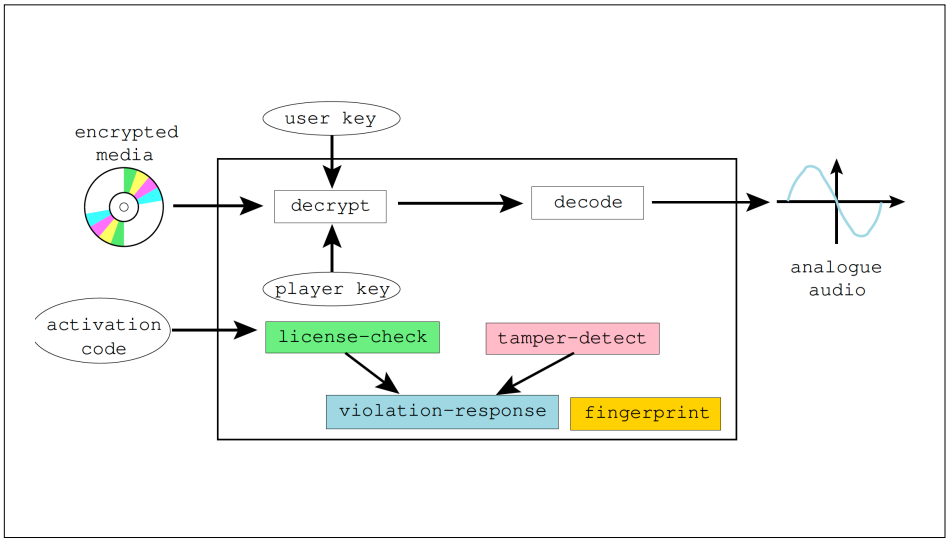
I.e. we analyze the source or binary code of the program itself.

- **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*

AttaoHing




```
typedef unsigned int uint32;
typedef char* caddr_t;
typedef uint32* waddr_t;

uint32 the_player_key = 0xbabeca75;
FILE* audio;

void play(uint32 user_key, uint32 encrypted_media[], int media_len) {
   int code;
   int i;
   struct timeval tv;
   for(i=0;i<media_len;i++) {
      uint32 key = user_key ^ the_player_key;
      uint32 decrypted = key ^ encrypted_media[i];
      if (time(0) > 1221011472) {
        *((int*)NULL)=99;
      }
      float decoded = (float)decrypted;
      fprintf(audio,"%f\n",decoded); fflush(audio);
   }
}

void player_main (uint32 argc, char *argv[]) {
   uint32 user_key = atoi(argv[1]);
   int i;
   uint32 encrypted_media[100];

   for(i=2; i<argc; i++)
      encrypted_media[i-2] = atoi(argv[i]);
   int media_len = argc-2;

   play(user_key,encrypted_media,media_len);
}

int main (int argc, char *argv[]) {
   audio = fopen("audio", "w");
   player_main(argc,argv);
   return 0;
}
```

```
void play(uint32 user_key,
          uint32 encrypted_media[], int media_len) {
   int code;
   int i;
   struct timeval tv;
   for(i=0;i<media_len;i++) {
      uint32 key = user_key ^ the_player_key;
      uint32 decrypted = key ^ encrypted_media[i];
      if (time(0) > 1221011472) {
        *((int*)NULL)=99;
      }
      float decoded = (float)decrypted;
      fprintf(audio,"%f\n",decoded); fflush(audio);
   }
}
```

Dynamic Attacks with gdb

Useful Commands

- Print dynamic symbols: `objdump -T player2`
- Disassemble: `objdump -d player2 | head`
- Find strings in the program: `strings player2`
- The bytes of the executable: `od -a player2`
- Trace library calls: `ltrace -i -e printf player2`
- Trace system calls: `strace -i -e write player2`

1. To start gdb:

```
gdb -write -silent \
   --args player2 0xca7ca115 1000
```

2. Search for a string in an executable:

```
(gdb) find startaddr, +length, "string"
(gdb) find startaddr, stopaddr, "string"
```

1. Breakpoints:

```
(gdb) break *0x......
(gdb) hbreak *0x......
```

`hbreak` sets a hardware breakpoint which doesn't modify the executable itself.

2. Watchpoints:

```
(gdb) rwatch *0x......
(gdb) awatch *0x......
```

1. To disassemble instructions:

```
(gdb) disass startaddress endaddress
(gdb) x/3i address
(gdb) x/i $pc
```

2. To examine data (x=hex,s=string, d=decimal, b=byte,...):

```
(gdb) x/x address
(gdb) x/s address
(gdb) x/d address
(gdb) x/b address
```

3. Print register values:

```
(gdb) info registers
```

1. Examine the callstack:

```
(gdb) where
(gdb) bt          -- same as where
(gdb) up          -- previous frame
(gdb) down        -- next frame
```

2. Step one instruction at a time:

```
(gdb) display/i $pc
(gdb) stepi     -- step one instruction
(gdb) nexti     -- step over function calls
```

3. Modify a value in memory:

```
(gdb) set {unsigned char}address = value
(gdb) set {int}address = value
```

Cracking an executable proceedes in these steps:

1. find the right address in the executable,
2. find what the new instruction should be,
3. modify the instruction in memory,
4. save the changes to the executable file.

Start the program to allow patching:

```
> gdb -write -q player1
```

Make the patch and exit:

```
(gdb) set {unsigned char} 0x804856f = 0x7f
(gdb) quit
```

```
> player 0xca7ca115 1 2 3 4
Please enter activation code: 42
expired!
Segmentation fault
```

- Find the assembly code equivalent of

  `if (time(0) > some value)···`

- Replace it with

  `if (time(0) <= some value)···`


```
> player 0xca7ca115 1 2 3 4
Please enter activation code: 42
expired!
Segmentation fault
```

- Find the assembly code equivalent of

  `if (time(0) > some value)···`

- Replace it with

  `if (time(0) <= some value)···`

```
(gdb) break time
Breakpoint 1 at 0x400680
(gdb) run
Please enter activation code: 42
Breakpoint 1, 0x400680 in time()
(gdb) where 2
#0  0x400680 in time
#1  0x4008b6 in ??
(gdb) up
#1  0x4008b6 in ??
(gdb) disassemble $pc-5 $pc+7
0x4008b1    callq   0x400680
0x4008b6    cmp     $0x48c72810,%rax
0x4008bc    jle     0x4008c8
```

CCCC	Name	Means
0000	O	overflow
0001	NO	Not overflow
0010	C/B/NAE	Carry, below, not above nor equal
0011	NC/AE/NB	Not carry, above or equal, not below
0100	E/Z	Equal, zero
0101	NE/NZ	Not equal, not zero
0110	BE/NA	Below or equal, not above
0111	A/NBE	Above, not below nor equal
1000	S	Sign (negative)
1001	NS	Not sign
1010	P/PE	Parity, parity even
1011	NP/PO	Not parity, parity odd
1100	L/NGE	Less, not greater nor equal
1101	GE/NL	Greater or equal, not less
1110	LE/NG	Less or equal, not greater
1111	G/NLE	Greater, not less nor equal

Patch the executable:

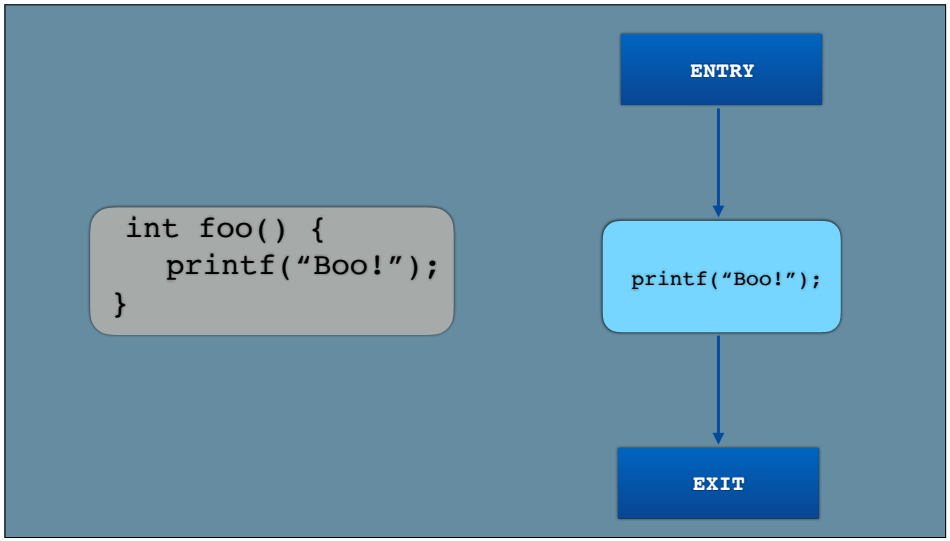
- replace the `jle` with a `jg` (x86 opcode 0x7f)

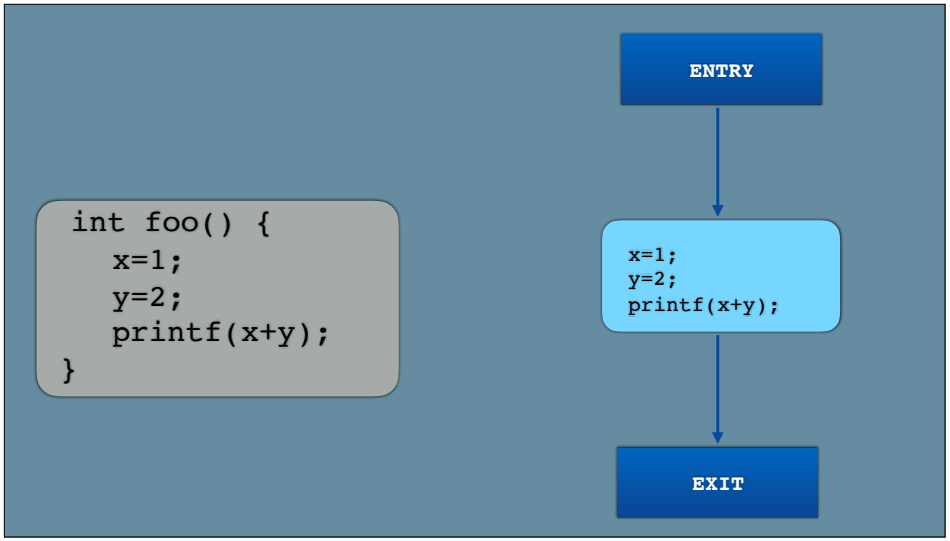
```
(gdb) set {unsigned char}0x4008bc = 0x7f
(gdb) disassemble 0x4008bc 0x4008be
0x4008bc    jg      0x4008c8
```

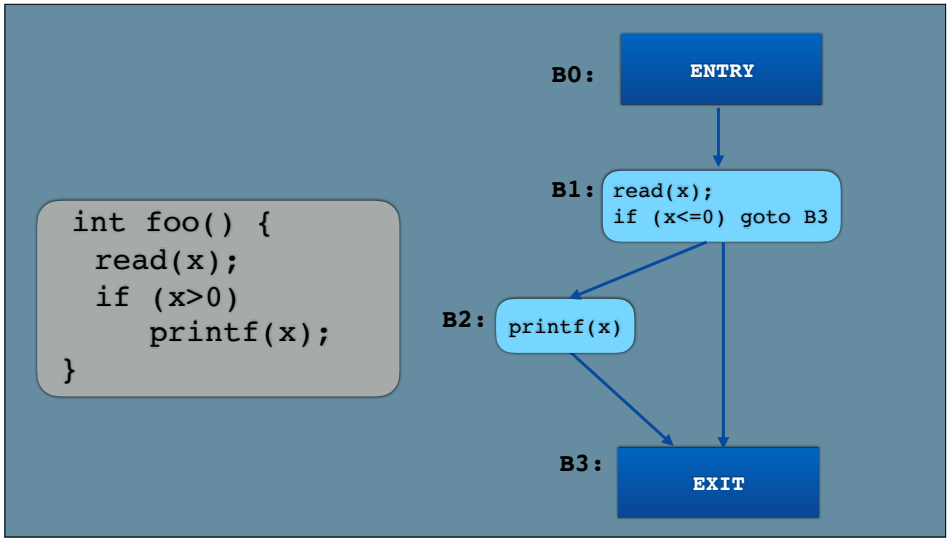
Control Flow Analysis

Control-Flow Graph (CFG)

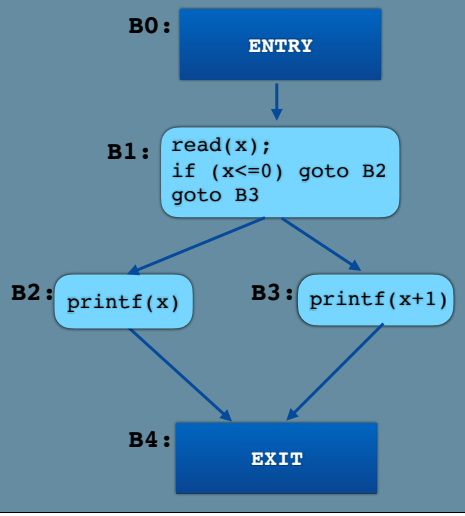
- A way to represent the possible flow of control inside a function.
- **Nodes**: called **basic blocks.** Each block consists of straight-line code ending (possibly) in a branch.
- **Edges**: An edge A → B means that control could flow from A to B.
- There is one unique **entry node** and one unique **exit node**.



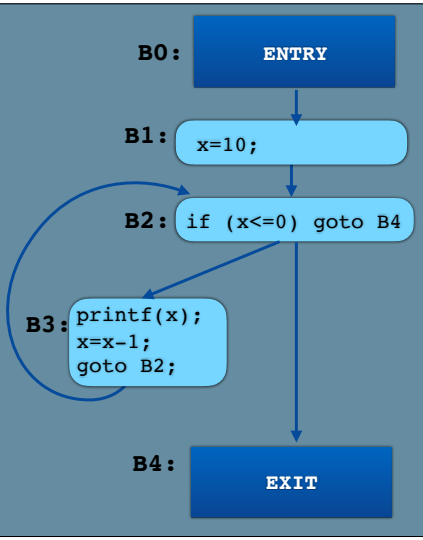




```
int foo() {
  read(x);
  if (x>0)
    printf(x);
  else
    printf(x+1);
}
```



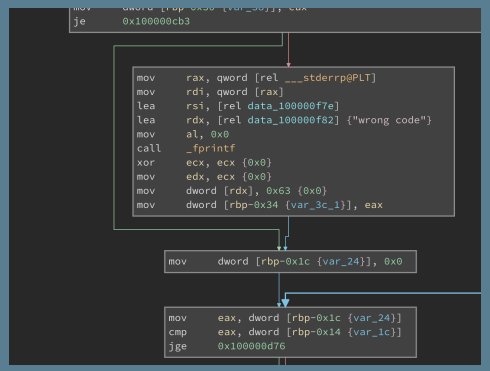

```
int foo() {
  x=10;
  while (x>0){
    printf(x);
    x=x-1;
  }
}
```



Exercise!

Work with your friends!!!

- Go to cloud.binary.ninja
- Load some binary file
- Click around!



Exercise!

Work with your friends!!!

Draw the CFG for this function!

```
X ← 20;
while (X<10){
  if (X%2==0)
     X←X+1;
};
```

Exercise!

Work with your friends!!!

Draw the CFG for this function!

```
read(x);
if (x%2==0) {
   while (x<10){
      printf(x);
      x++;
   }
}
```

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!

Work with your friends!!!

Convert to CFG!

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

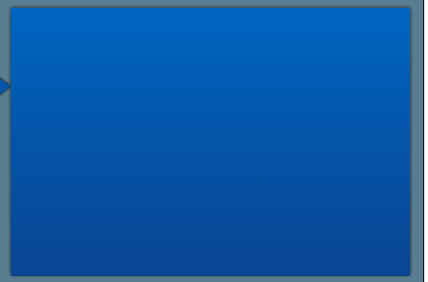
First simplify!

```
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
```

Step 1: Compute Leaders

```
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
```

1. Compute **leaders**:
   1. the **first instruction**
   2. a **target of a branch**
   3. any **instruction following a conditional branch**



Static Attacks with Ghidra

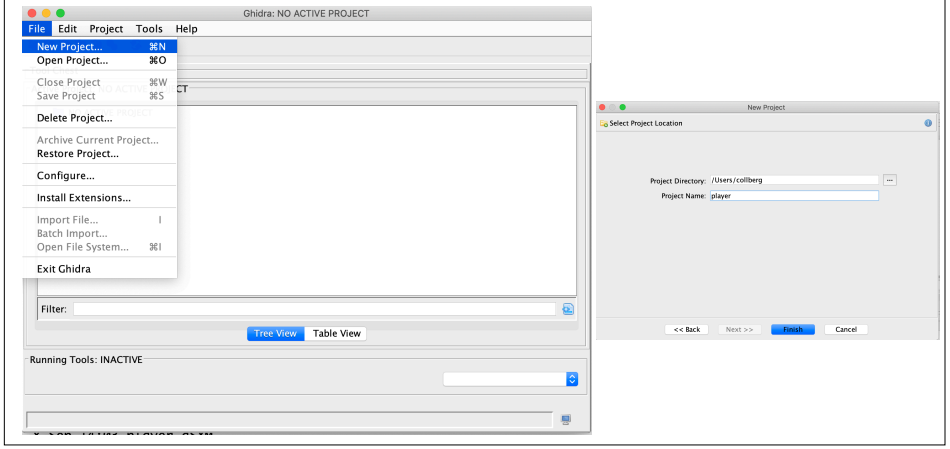
- Download Ghidra: https://ghidra-sre.org
- The installation instructions are here:
  https://ghidra-sre.org/InstallationGuide.html#Platforms
- You will first have to install the Java JDK:
  https://www.oracle.com/java/technologies/jdk12-downloads.html
- I had to add the following lines to my .bashrc file (this is for MacOS):

```
export JAVA_HOME="/Library/Java/JavaVirtualMachines/jdk-12.0.2.jdk/Contents/Home"
export PATH="$PATH:$JAVA_HOME"
```

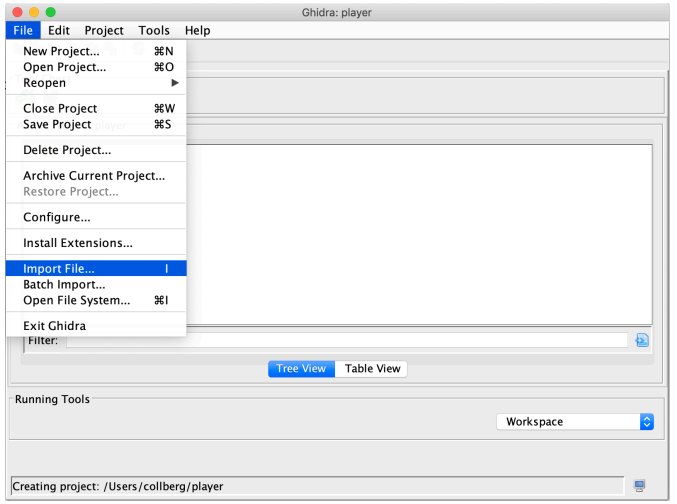
- Then run Ghidra from within the directory downloaded:

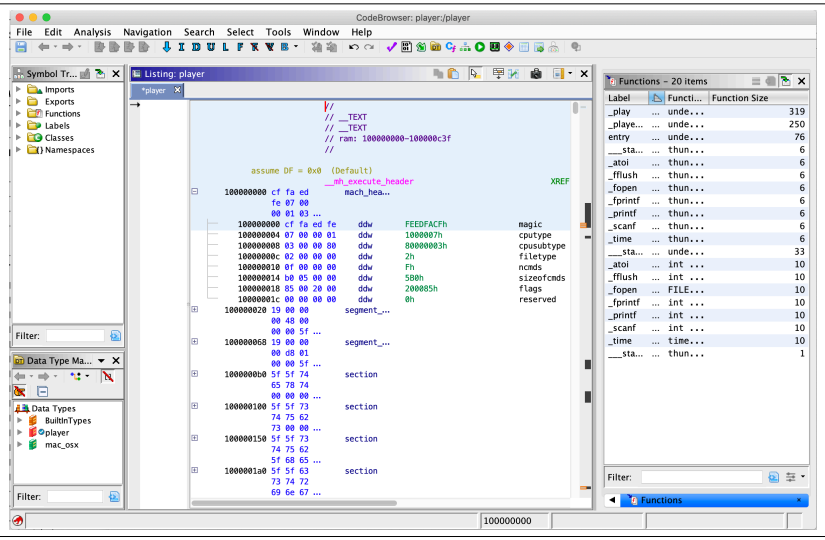
```
> cd ghidra_9.0.4
> ghidraRun
```

- Create a new project

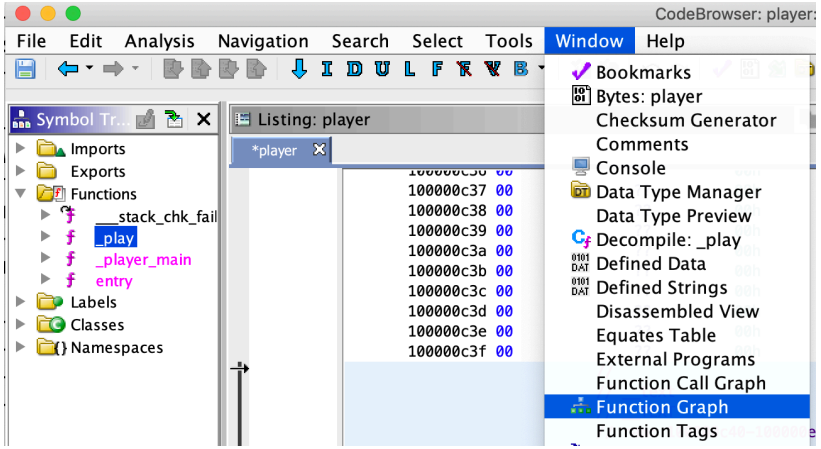


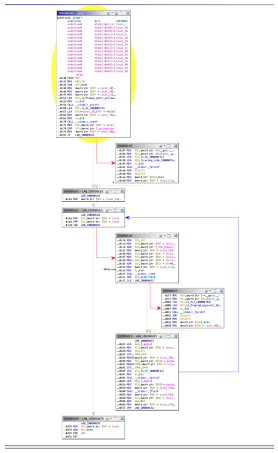
- Load the binary file you want to analyze





- Select a function and create its CFG





- Locate the branch you think you need to edit.

```
...0ce5 MOV   AL,0x0
...0ce7 CALL  __stubs::_time
...0cec CMP   EAX,0x48c72810
...0cf1 JLE   LAB_100000d23
```

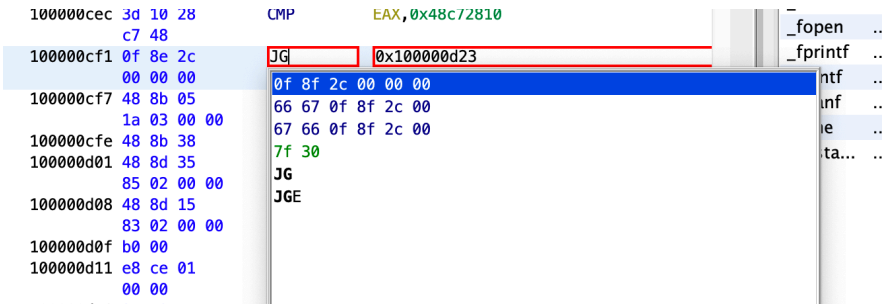
```
100000cf7
...0cf7 MOV   RAX,qwo
...0cfe MOV   RDI,qwo
...0d01 LEA   RSI.[s
```

- Select “Patch Instruction”

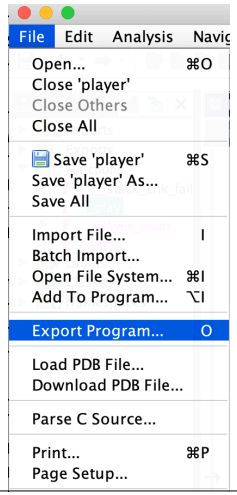
```
100000ce2 89 45 dc      MOV     dword ptr [RBP + local_2c],EAX
100000ce5 b0 00         MO
100000ce7 e8 0a 02      CA
          00 00
100000cec 3d 10 28      CM
          c7 48
100000cf1 0f 8e 2c      JL
          00 00 00
100000cf7 48 8b 05      MO                                  rp]
          1a 03 00 00
100000cfe 48 8b 38      MO                                  _stde
100000d01 48 8d 35      LE
          85 02 00 00
100000d08 48 8d 15      LE                                  2]
          83 02 00 00
100000d0f b0 00         MO
100000d11 e8 ce 01      CA
          00 00
100000d16 31 c9         XO
100000d18 89 ca         MO
```

Bookmark... ⌘D
Clear Code Bytes C
Clear With Options...
Clear Flow and Repair...
Copy ⌘C
Copy Special...
Paste ⌘V
Comments ▶
Instruction Info...
Modify Instruction Flow...
Patch Instruction ⇧⌘G
Processor Manual...
Processor Options...

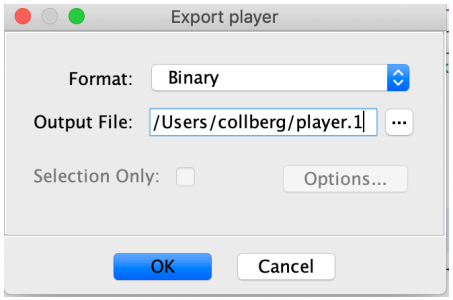
- Change JLE (jump less than or equal) to the inverse JG (jump greater).
- Note that 8e is changed to 8f!



CCCC	Name	Means
0000	O	overflow
0001	NO	Not overflow
0010	C/B/NAE	Carry, below, not above nor equal
0011	NC/AE/NB	Not carry, above or equal, not below
0100	E/Z	Equal, zero
0101	NE/NZ	Not equal, not zero
0110	BE/NA	Below or equal, not above
0111	A/NBE	Above, not below nor equal
1000	S	Sign (negative)
1001	NS	Not sign
1010	P/PE	Parity, parity even
1011	NP/PO	Not parity, parity odd
1100	L/NGE	Less, not greater nor equal
1101	GE/NL	Greater or equal, not less
1110	LE/NG	Less or equal, not greater
1111	G/NLE	Greater, not less nor equal



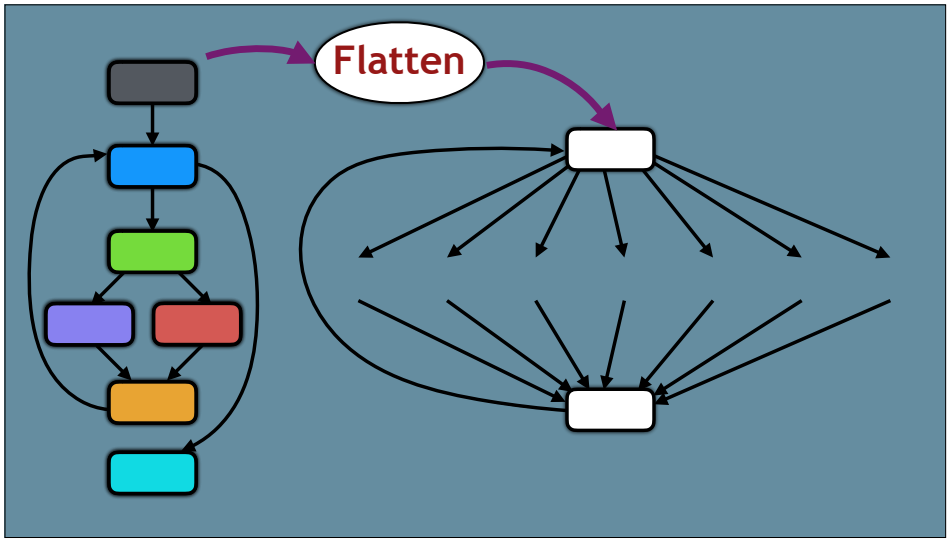
- Export the edited binary to a new file.



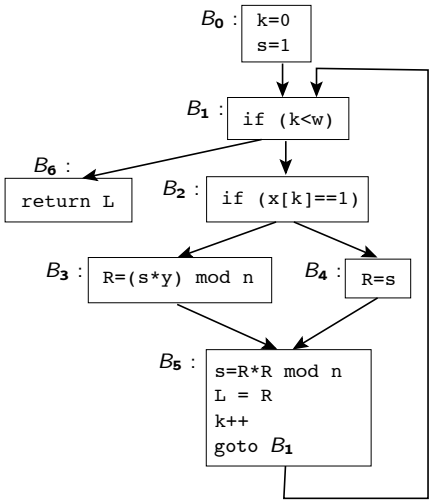
- Now run the original and edited file!

```
bash-3.2$ ~/player 1 2 3 4
Please enter activation code: 42
Program expired!!
Segmentation fault: 11
bash-3.2$
bash-3.2$ ~/player.1.bin 1 2 3 4
Please enter activation code: 42
bash-3.2$
```

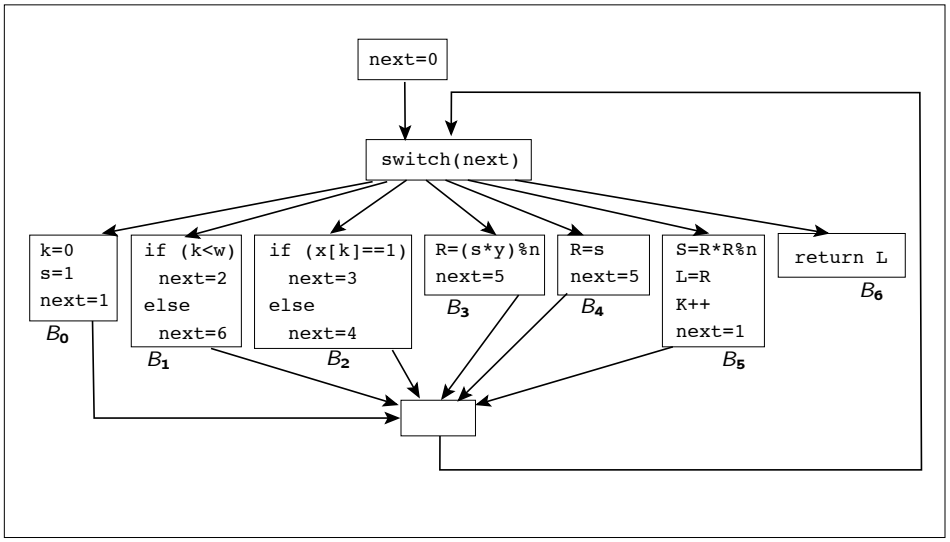
Obfuscation
—
Control Flow Flattening




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

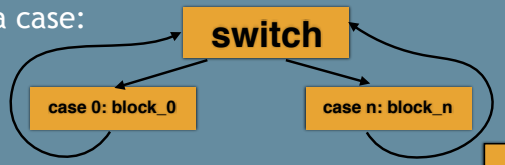



```
int modexp(int y, int x[], int w, int n) {
int R, L, k, s;
   int next=0;
   for(;;)
      switch(next) {
         case 0 : k=0; s=1; next=1; break;
         case 1 : if (k<w) next=2; else next=6; break;
         case 2 : if (x[k]==1) next=3; else next=4; break;
         case 3 : R=(s*y)%n; next=5; break;
         case 4 : R=s; next=5; break;
         case 5 : s=R*R%n; L=R; k++; next=1; break;
         case 6 : return L;
   }
}
```



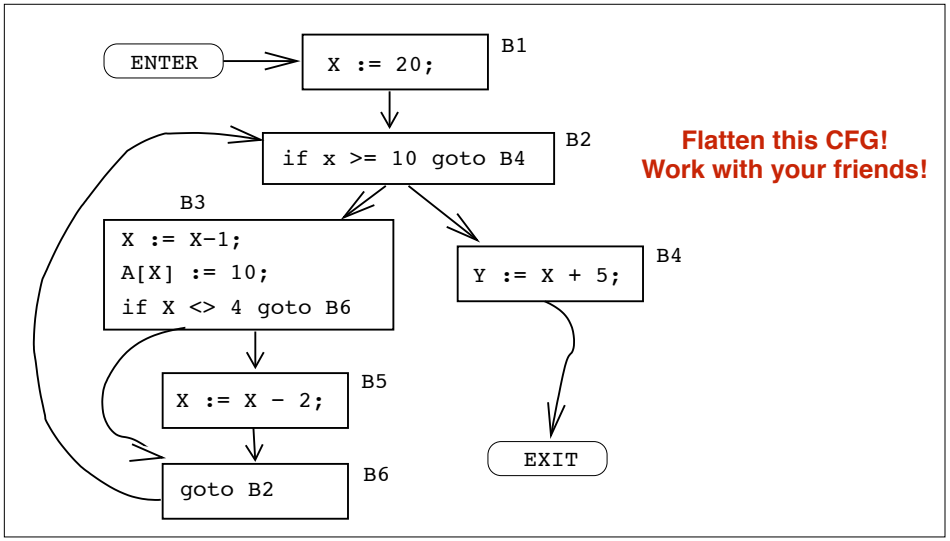
Flattening Algorithm

1. Construct the CFG
2. Add a new variable **int next=0;**
3. Create a switch inside an infinite loop, where every basic block is a case:



4. Add code to update the **next** variable:

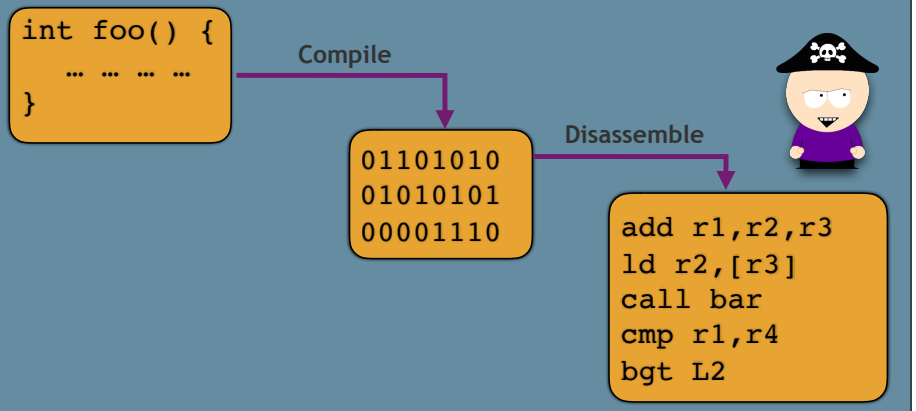
```
case n: {
  if (expression)
    next = ...
  else
    next = ...
}
```

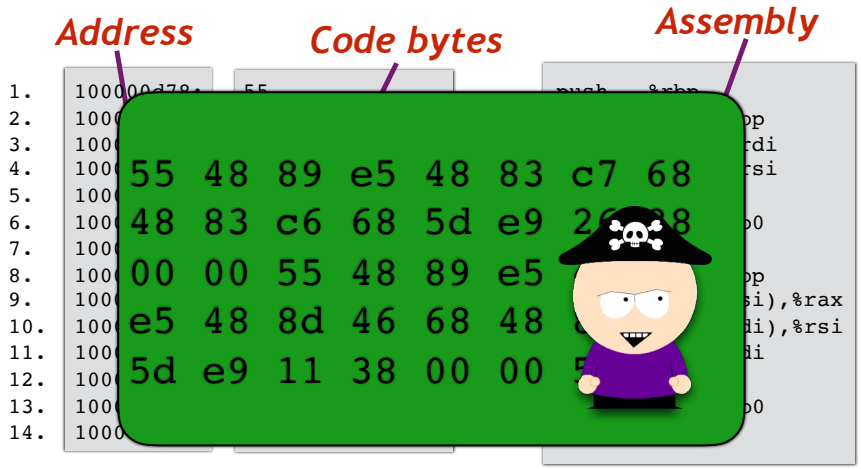


Flatten this CFG!
Work with your friends!

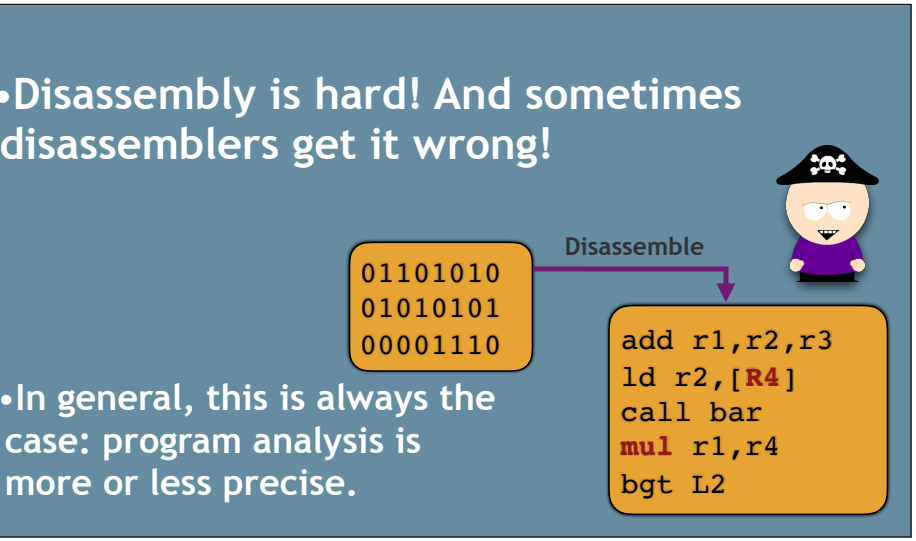
Disassembly and Anti Disassembly

- *Attackers: prefer looking at assembly code than machine code*

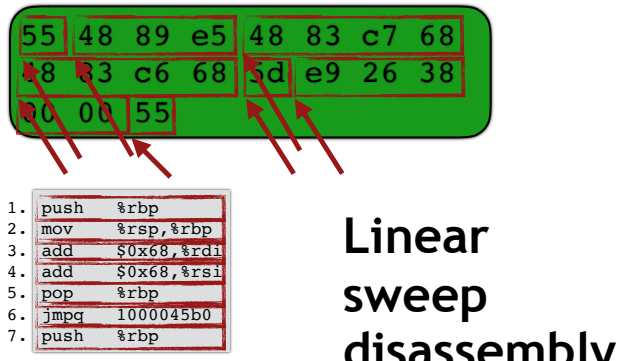




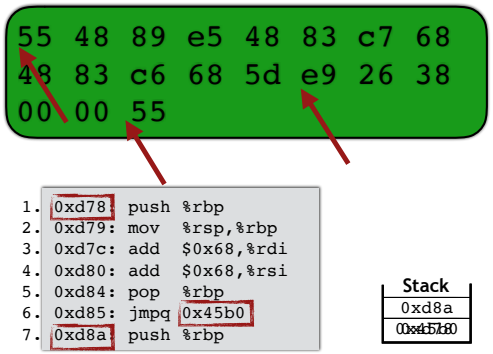
- Disassembly is hard! And sometimes disassemblers get it wrong!



- In general, this is always the case: program analysis is more or less precise.



Linear sweep disassembly



Recursive traversal disassembly

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?

Insert Bogus Dead Code

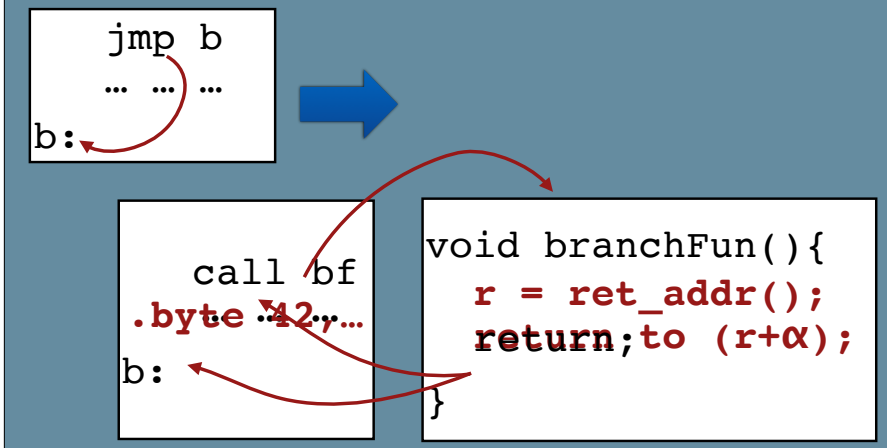
- Insert unreachable bogus instructions:

```
if (PF)
 asm(".byte 0x55 0x23 0xff…");
```

- This kind of lightweight obfuscation is common in malware.

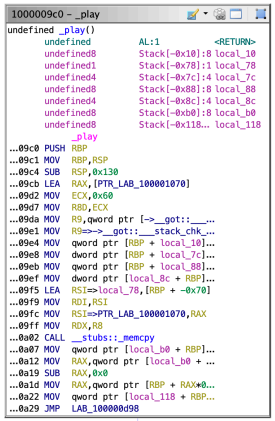
```
uint32 Skypes_hash_function () {
  addr_t addr =(addr_t)((uint32)addr ^(uint32)addr);
  addr = (addr_t)((uint32) addr + 0 x688E5C);
  uint32 hash = 0x320E83 ^ 0x1C4C4 ;
  int bound = hash + 0 xFFCC5AFD ;

  do {
     uint32 data =*((addr_t)((uint32)addr + 0x10));
     goto b1; asm volatile (".byte 0x19"); b1:
     hash = hash ⊕ data ; addr -= 1; bound --;
  } while (bound !=0);

  goto b2;
   asm volatile (".byte 0x73");
  b2:
  goto b3;
   asm volatile (".word 0xC8528417,…");
  b3:
  hash -= 0x4C49F346; return hash;
}
```

Branch Functions

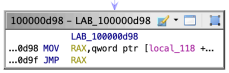


How does Ghidra do with Branch Functions and Control Flow Flattening?

```
tigress --Environment=x86_64:Darwin:Gcc:4.6 \
   --Transform=InitBranchFuns \
      --InitBranchFunsCount=1 \
      --InitBranchFunsObfuscate=false \
   --Transform=AntiBranchAnalysis \
      --AntiBranchAnalysisKinds=branchFuns \
      --AntiBranchAnalysisObfuscateBranchFunCall=false \
      --Functions=play \
   --out=player-bf.c player.c
gcc player-bf.c -o player-bf
```



```
1000009c0 - _play
undefined _play()
      undefined         AL:1           <RETURN>
      undefined8        Stack[-0x10]:8 local_10
      undefined1        Stack[-0x78]:1 local_78
      undefined4        Stack[-0x7c]:4 local_7c
      undefined8        Stack[-0x88]:8 local_88
      undefined4        Stack[-0x8c]:4 local_8c
      undefined8        Stack[-0xb0]:8 local_b0
      undefined8        Stack[-0x118... local_118
        _play
...09c0 PUSH  RBP
...09c1 MOV   RBP,RSP
...09c4 SUB   RSP,0x130
...09cb LEA   RAX,[PTR_LAB_100001070]
...09d2 MOV   ECX,0x60
...09d7 MOV   R8D,ECX
...09da MOV   R9,qword ptr [->__got::___...
...09e1 MOV   R9=>->__got::___stack_chk_...
...09e4 MOV   qword ptr [RBP + local_10]...
...09e8 MOV   dword ptr [RBP + local_7c]...
...09eb MOV   qword ptr [RBP + local_88]...
...09ef MOV   dword ptr [local_8c + RBP]...
...09f5 LEA   RSI=>local_78,[RBP + -0x70]
...09f9 MOV   RDI,RSI
...09fc MOV   RSI=>PTR_LAB_100001070,RAX
...09ff MOV   RDX,R8
...0a02 CALL  __stubs::_memcpy
...0a07 MOV   qword ptr [local_b0 + RBP]...
...0a12 MOV   RAX,qword ptr [local_b0 + ...
...0a19 SUB   RAX,0x0
...0a1d MOV   RAX,qword ptr [RBP + RAX*0...
...0a22 MOV   qword ptr [local_118 + RBP...
...0a29 JMP   LAB_100000d98
```



```
100000d98 - LAB_100000d98
                LAB_100000d98
...0d98 MOV   RAX,qword ptr [local_118 +...
...0d9f JMP   RAX
```

- Output from Ghidra!

```
1
2  void _play(void)
3
4  {
5    undefined local_78 [88];
6    code *UNRECOVERED_JUMPTABLE;
7    undefined8 local_10;
8
9    local_10 = *(undefined8 *)___stack_chk_guard;
10   _memcpy(local_78,&PTR_LAB_100001070,0x60);
11                     /* WARNING: Could not recover jumptable at 0x000100000d9f. Too many branches */
12                     /* WARNING: Treating indirect jump as call */
13   (*UNRECOVERED_JUMPTABLE)();
14   return;
15 }
16
```

Exercise!

```
************************************************************
* 7) Branch Functions
************************************************************
tigress --Environment=x86_64:Linux:Gcc:4.6 \
   --Transform=InitBranchFuns \
      --InitBranchFunsCount=1 \
   --Transform=AntiBranchAnalysis \
      --AntiBranchAnalysisKinds=branchFuns \
      --Functions=fib \
   --out=fib7.c fib.c
```