

# Lec 20: Binary Analysis

CS492E: Introduction to Software Security

Sang Kil Cha

# Binary Analysis is Difficult

Not only automated analysis, but manual analysis is difficult.

What's the problem?

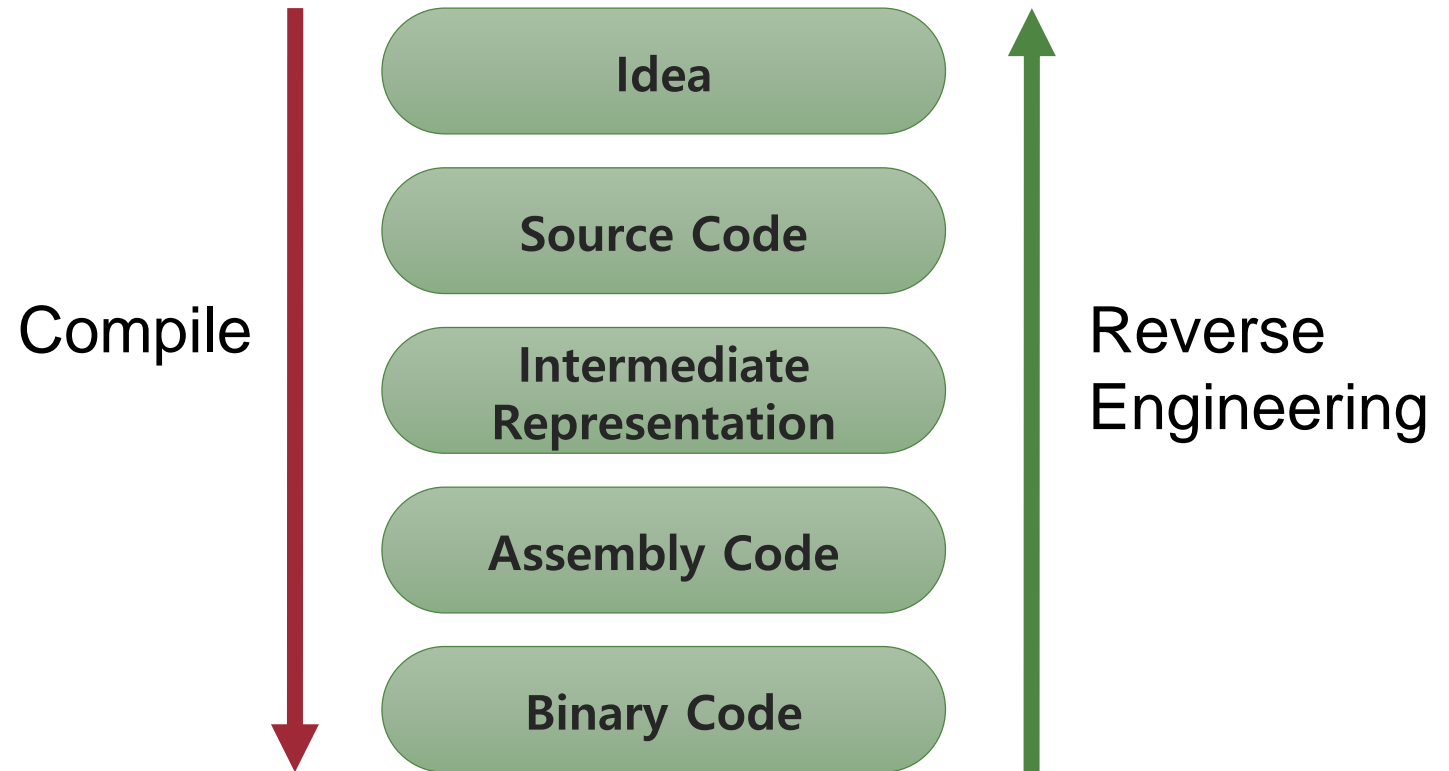
# No Program Abstraction!

```
4C 8B 47 08      mov     r8,qword ptr [rdi+8]
BA 02 00 00 00   mov     edx,2
48 8B 4F 20      mov     rcx,qword ptr [rdi+20h]
45 0F B7 08      movzx   r9d,word ptr [r8]
E8 54 16 00 00   call   00000001400026BC
48 8B 74 24 38   mov     rsi,qword ptr [rsp+38h]
8B C3           mov     eax,ebx
48 8B 5C 24 30   mov     rbx,qword ptr [rsp+30h]
48 83 C4 20      add     rsp,20h
5F             pop     rdi
C3             ret
48 8B C4         mov     rax,rsp
48 89 58 08      mov     qword ptr [rax+8],rbx
```

No types  
No variables  
No functions

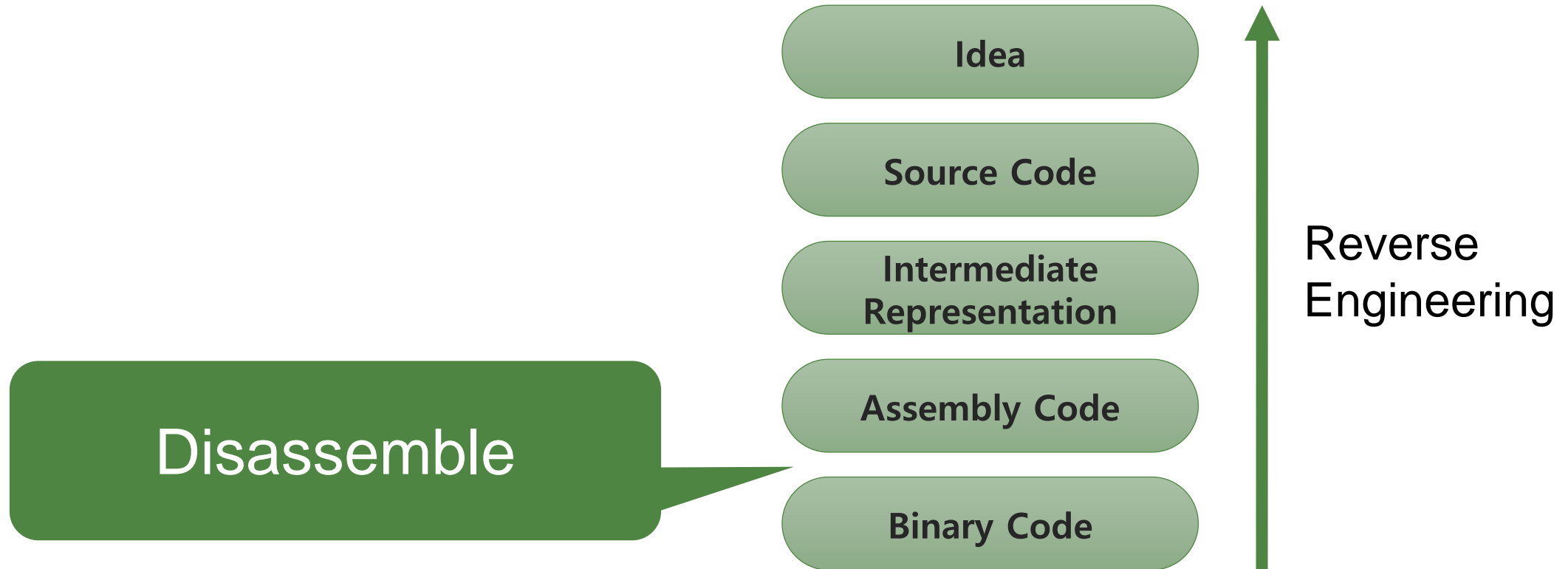
...

# Binary Analysis (= Reverse Engineering)



# Diassembly

# First Step: Disassembling Binary Code



# Recursive Descent Disassembly

1. Disassemble instruction one by one until reaching branch instructions
2. When there is a branch instruction, we examine the target address(es) of the branch instruction, and recursively disassemble from there.

# Figuring out Branch Target(s)

JMP EAX

CALL [EAX]

Can we statically decide what kind of values EAX can have?



# Simplest Example

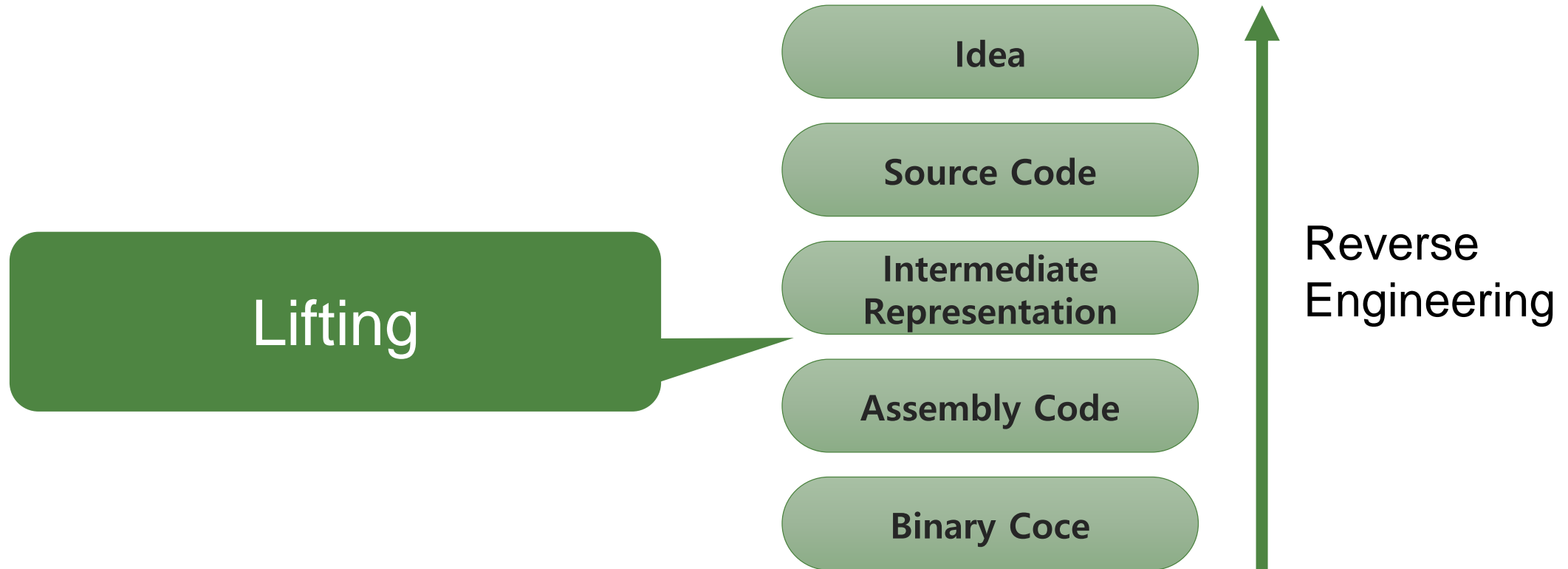
```
int main(int c, char** argv)
{
    switch (c)
    {
        case 1: counter += 20; break;
        case 2: counter += 33; break;
        case 3: counter += 62; break;
        case 4: counter += 15; break;
        case 5: counter += 416; break;
        case 6: counter += 3545; break;
        case 7: counter += 23; break;
        case 8: counter += 81; break;
    }
    return counter;
}
```

```
0000000000001130 <main>:
    1130: push   rbp
    1131: mov   rbp, rsp
    1134: mov   DWORD PTR [rbp-0x4], 0x0
    113b: mov   DWORD PTR [rbp-0x8], edi
    113e: mov   QWORD PTR [rbp-0x10], rsi
    1142: mov   eax, DWORD PTR [rbp-0x8]
    1145: add   eax, 0xffffffff
    1148: mov   ecx, eax
    114a: sub   eax, 0x7
    114d: mov   QWORD PTR [rbp-0x18], rcx
    1151: ja    122e <main+0xfe>
    1157: lea   rax, [rip+0xea6]
    115e: mov   rcx, QWORD PTR [rbp-0x18]
    1162: movsxd rdx, DWORD PTR [rax+rcx*4]
    1166: add   rdx, rax
    1169: jmp   rdx
    116b: lea   rax, [rip+0x2ebe]
    1172: mov   rcx, QWORD PTR [rax]
    1175: add   rcx, 0x14
```

...

# Lifting

# Second Step: Lifting



# Why IR?

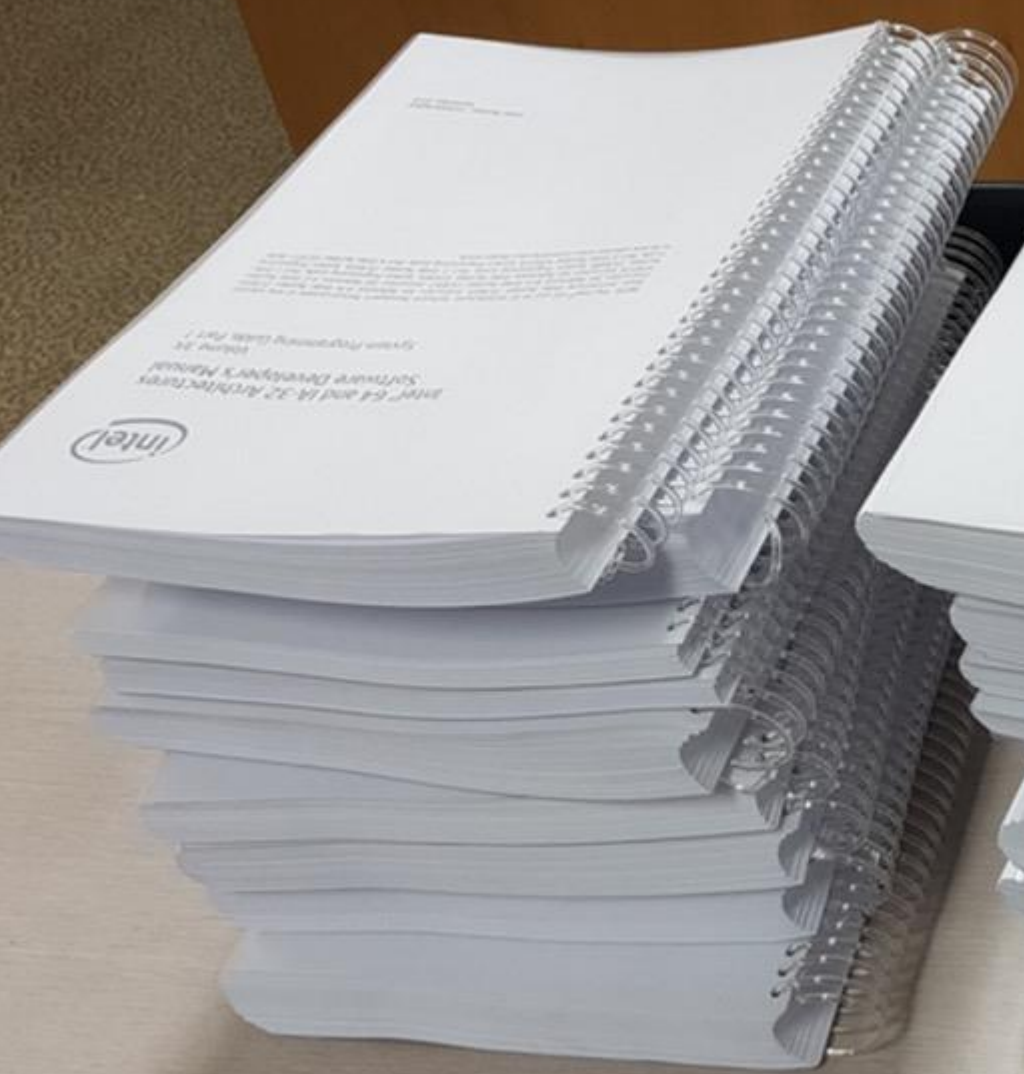
- Platform-neutral representation
- IR represents explicit semantics

# Lifting Example

add dword ptr [ecx], eax



```
T_0:i32 := EAX
T_1:i32 := [ECX]
T_2:i32 := (T_0:i32 + T_1:i32)
[ECX] := T_2:i32
CF := (T_2:i32 < T_0:i32)
OF := ((high:i1(T_0:i32) = high:i1(T_1:i32)) & (high:i1(T_0:i32) ^ high:i1(T_2:i32)))
AF := (((T_2:i32 ^ T_0:i32) ^ T_1:i32) & (0x1:i32 << 0x4:i32)) = (0x1:i32 << 0x4:i32)
SF := high:i1(T_2:i32)
ZF := (T_2:i32 = 0x0:i32)
T_3:i32 := (T_2:i32 ^ (T_2:i32 >> zext:i32(0x4:i8)))
T_4:i32 := ((T_2:i32 ^ (T_2:i32 >> zext:i32(0x4:i8))) ^ (T_3:i32 >> zext:i32(0x2:i8)))
PF := (~ low:i1((((T_2:i32 ^ (T_2:i32 >> zext:i32(0x4:i8))) ^ (T_3:i32 >> zext:i32(0x2:i8))) ^ (T_4:i32 >> zext:i32(0x1:i8))))))
```



# Example: PUSH

## Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).  
The address size is used only when referencing a source operand in memory.
- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).  
The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Core and Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FF /6	PUSH <i>r/m16</i>	M	Valid	Valid	Push <i>r/m16</i> .
FF /6	PUSH <i>r/m32</i>	M	N.E.	Valid	Push <i>r/m32</i> .
FF /6	PUSH <i>r/m64</i>	M	Valid	N.E.	Push <i>r/m64</i> .
50+ <i>rw</i>	PUSH <i>r16</i>	0	Valid	Valid	Push <i>r16</i> .
50+ <i>rd</i>	PUSH <i>r32</i>	0	N.E.	Valid	Push <i>r32</i> .
50+ <i>rd</i>	PUSH <i>r64</i>	0	Valid	N.E.	Push <i>r64</i> .
6A <i>ib</i>	PUSH <i>imm8</i>	1	Valid	Valid	Push <i>imm8</i> .

If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Core and Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

## Operation

(\* See Description section for possible sign-extension or zero-extension of source operand and for \*)  
(\* a case in which the size of the memory store may be smaller than the instruction's operand size \*)

IF StackAddrSize = 64

THEN

IF OperandSize = 64

THEN

RSP ← RSP - 8;  
Memory[SS:RSP] ← SRC; (\* push quadword \*)

ELSE IF OperandSize = 32

THEN

RSP ← RSP - 4;  
Memory[SS:RSP] ← SRC; (\* push dword \*)

ELSE (\* OperandSize = 16 \*)

RSP ← RSP - 2;  
Memory[SS:RSP] ← SRC; (\* push word \*)

FI;

ELSE IF StackAddrSize = 32

THEN

IF OperandSize = 64

THEN

ESP ← ESP - 8;  
Memory[SS:ESP] ← SRC; (\* push quadword \*)

ELSE IF OperandSize = 32

THEN

ESP ← ESP - 4;  
Memory[SS:ESP] ← SRC; (\* push dword \*)

FI;

IF OperandSize = 32

THEN

SP ← SP - 4;  
Memory[SS:SP] ← SRC; (\* push dword \*)

ELSE (\* OperandSize = 16 \*)

SP ← SP - 2;  
Memory[SS:SP] ← SRC; (\* push word \*)

FI;

FI;

# Example: BSF (Pseudo Code)

```
if ( source == 0 ) {  
    ZF = 0;  
    destination = undefined;  
}  
else {  
    ZF = 0;  
    T = 0;  
    while (Bit(source, T) == 0) {  
        T = T + 1;  
        destination = T;  
    }  
}
```



# IR is Complex and Error-Prone!

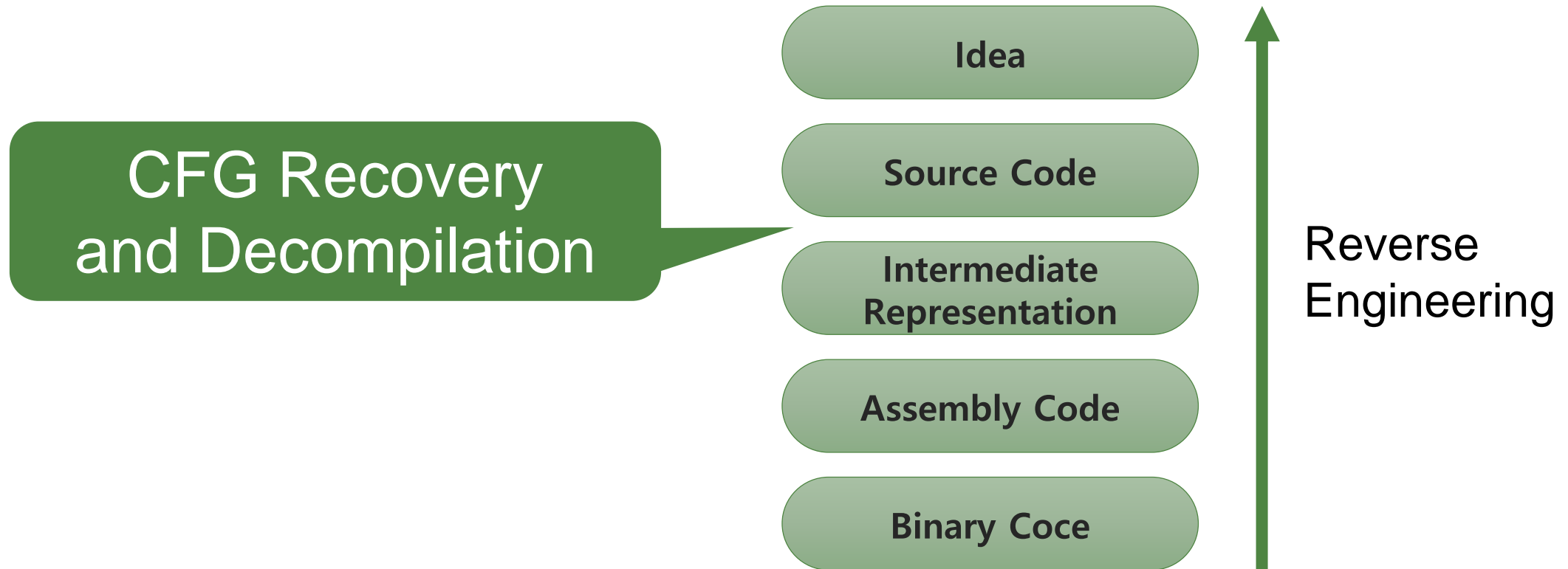
Human is writing the lifter!

# What Happens When IR is Incorrect?

- CVE-2009-2267, CVE-2009-1542
  - Security vulnerabilities
- QEMU failed to load a Linux kernel due to an IR bug
  - <http://lists.gnu.org/archive/html/qemu-devel/2017-01/msg03062.html>

# CFG Recovery & More

# Third Step: CFG Recovery & More



# Problem

- Recursive disassembly includes CFG recovery, but perfect disassembly is infeasible.
- Knowing the function entry points remains problematic.

# Call Target = Function?

- **False positives:** call targets may not be a function entry point
- **False negatives:** regular jump targets can be a function entry point

# Example: False Positives

```
11a0:      55                push   ebp
11a1:      89 e5            mov    ebp, esp
11a3:      50              push   eax
11a4:      e8 00 00 00 00   call  11a9
11a9:      58              pop    eax
11aa:      81 c0 57 2e 00 00 add    eax, 0x2e57
11b0:      31 c9            xor    ecx, ecx
```

...

# Example: False Negatives

...

```
c30a0:      31 f6                xor     esi,esi
c30a2:      eb 0c                jmp     c30b0
```

000000000000c30b0 <\_bfd\_generic\_read\_ar\_hdr\_mag>:

```
c30b0:      41 57                push   r15
c30b2:      41 56                push   r14
```

...



# Any Solution?

- Function entry points often have specific patterns
  - But not all of them follow the patterns
- PC getters have specific patterns
  - Inlined assembly code?

# Partitioned Functions

```
000000000007d70 <move_fd.part.0>:
 7d70:      55          push   rbp
 7d71:      89 fd       mov    ebp,edi
 7d73:      e8 28 bc ff call   39a0 <dup2@plt>
 7d78:      89 ef       mov    edi,ebp
 7d7a:      5d          pop    rbp
 7d7b:      e9 f0 bc ff jmp    3a70 <close@plt>
...
00000000000ac00 <open_input_files>:
...
 ae82:      45 85 f6    test   r14d,r14d
 ae85:      74 0a       je     ae91
 ae87:      31 f6       xor    esi,esi
 ae89:      44 89 f7    mov    edi,r14d
 ae8c:      e8 df ce ff call   7d70 <move_fd.part.0>
...
```

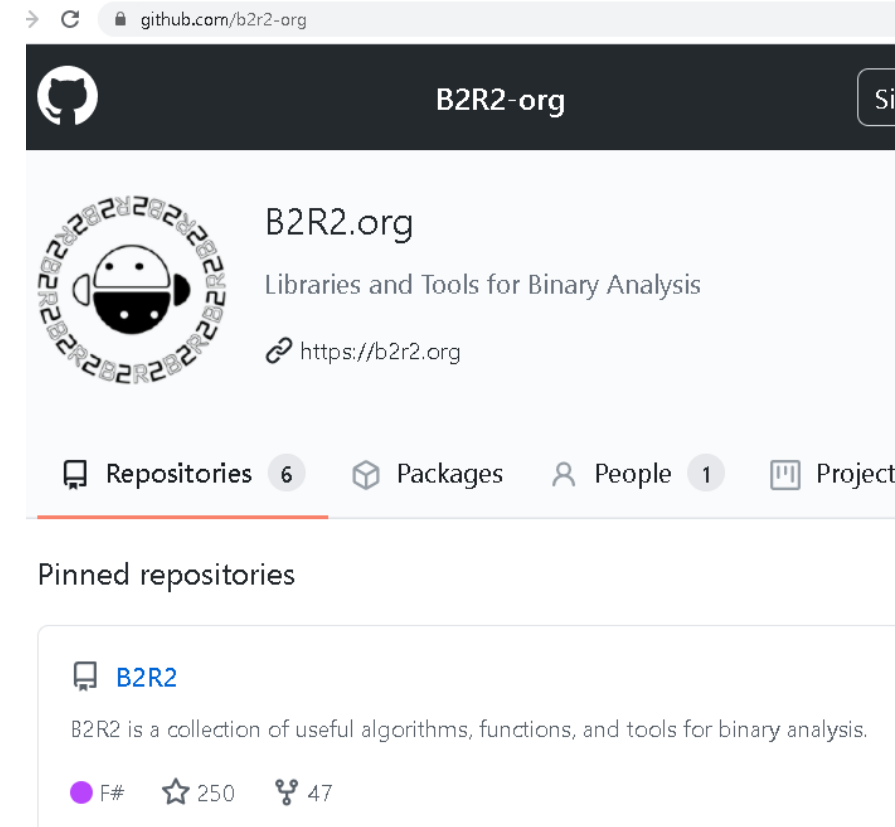
```
static void
move_fd (int oldfd, int newfd)
{
    if (oldfd != newfd)
    {
        dup2 (oldfd, newfd);
        close (oldfd);
    }
}
```

# Decompile?

- Value-Set Analysis (VSA)
  - Where are the variables?
- Type inference
  - Can we recover variable types?
- Structure Analysis
  - Can we recover high-level control flow structures?

# B2R2: the Next Generation Binary Analysis Framework

- Binary analysis platform developed in KAIST
- Won the best paper award in NDSS BAR 2019
- <https://github.com/B2R2-org/B2R2>



The screenshot shows the GitHub organization page for B2R2-org. The page header includes the GitHub logo and the organization name "B2R2-org". Below the header, there is a profile picture of a robot head surrounded by binary code, the organization name "B2R2.org", and the description "Libraries and Tools for Binary Analysis". A link to the website "https://b2r2.org" is also present. The page features navigation tabs for "Repositories" (6), "Packages", "People" (1), and "Projects". Under the "Pinned repositories" section, the "B2R2" repository is highlighted, with a description: "B2R2 is a collection of useful algorithms, functions, and tools for binary analysis." The repository statistics show it is written in F#, has 250 stars, and 47 forks.

# Conclusion

- Binary analysis is largely unsolved.
- There are many on-going research projects in every step of binary analysis.

# Questions?