Lec 19: Obfuscation

CS492E: Introduction to Software Security

Sang Kil Cha

Motivation

Can we make it difficult to reverse engineer binaries?

Obfuscation

The deliberate act of creating source or machine code that is difficult for humans to understand.

- Wikipedia

Why Obfuscate Binary Code?

- Digital Rights Management (DRM) or Copy Protection
  - Example: you have a secret algorithm in your software product, and you don't want to reveal it
- What about malicious uses?

Ultimate Copy Protection?

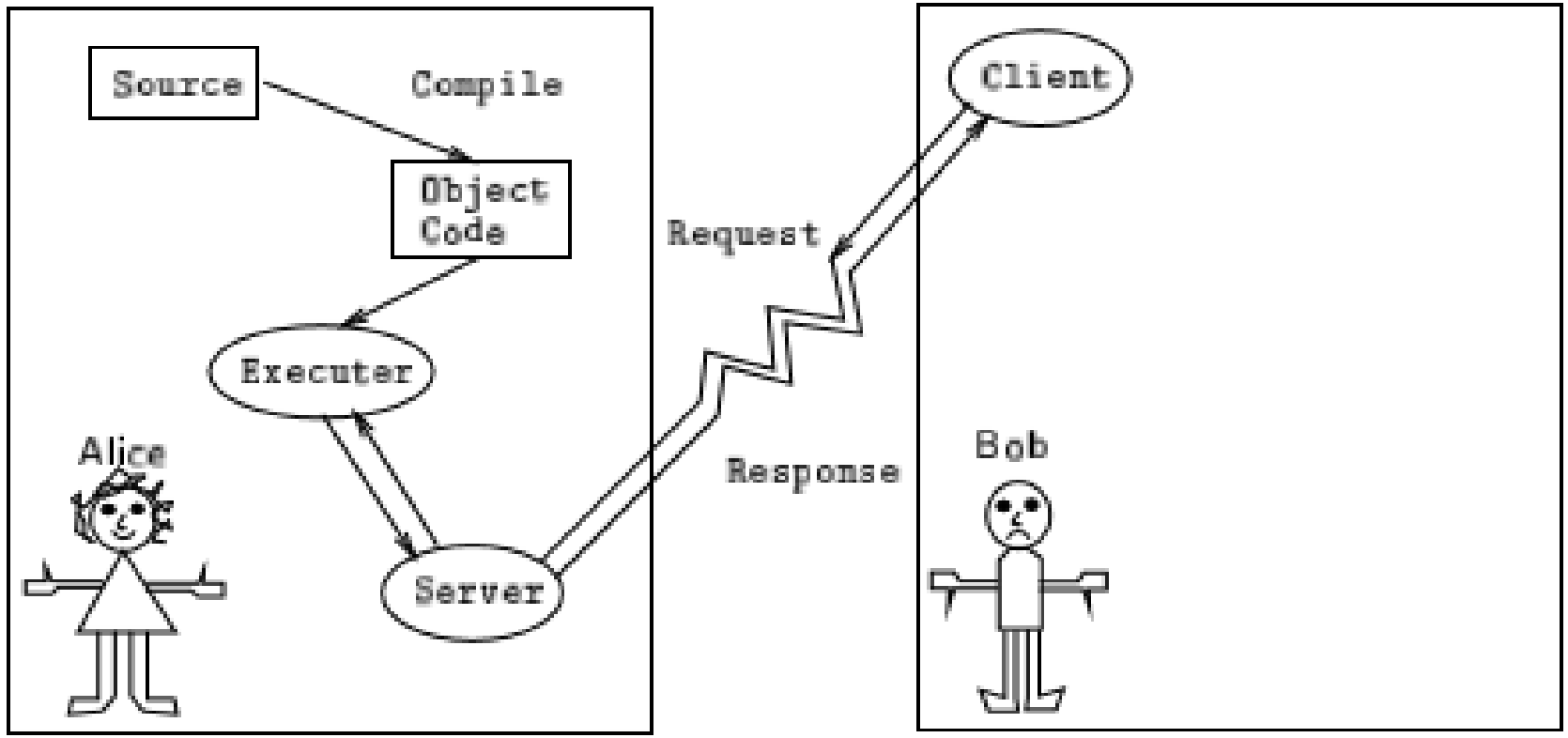


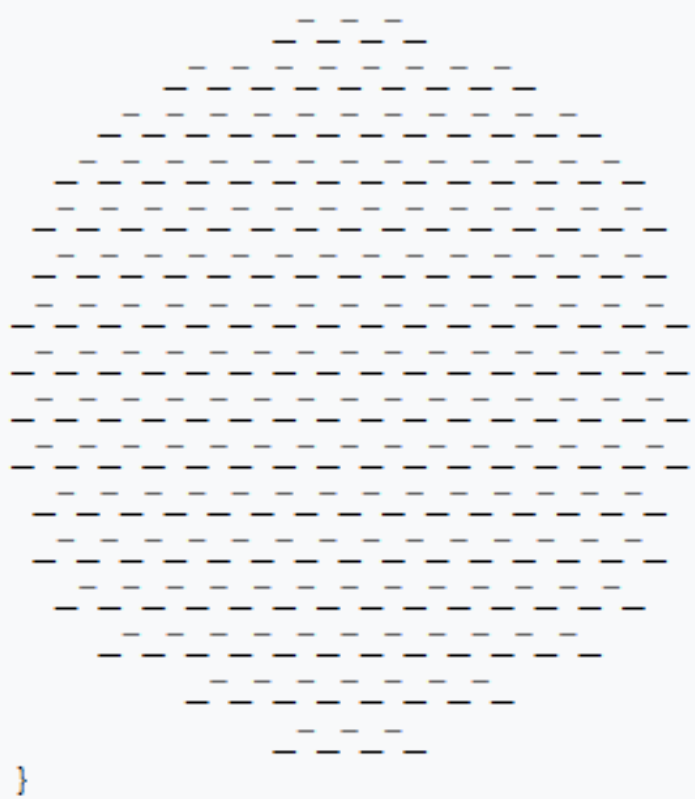
Figure from “A Taxonomy of Obfuscating Transformations”, Tech Report, University of Auckland, 1997

Why Not Obfuscate All the Time?

- Performance overhead
- Hard to debug / maintain

Traditional Obfuscation (Source Level)

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
```



```
}
```


Image take from https://en.wikipedia.org/wiki/International_Obfuscated_C_Code_Contest

Traditional Obfuscation (Source Level)

```
#include   /*recall-the#    /-good--old-#    /IOCCC-days!#    */<unistd.h>
 typedef  unsigned/*int*/  short U;U(main)  [32768],n,r[8];  __attribute__((
# define  R(x)       A(r[  7-(n       >>x&  7)],       (n>>  x>>3       )%8)
#define   C(x)       (U*)  ((/*             ||O|             -dpd
*/char*)  main       *(x)  )/*|             |CC|             ||*/
# define  A(v,       i)(i  ?i<2             ?C(v             ):i#
-4?v+=2,  C(i-       6?v-  2:v+       *C(v  -2))       :C(v  -=2)       :&v)
/*lian*/  constructor))U(  x)(){for(;;*r+=  2,*r+=!n?_exit(  write(2,"Illeg"
"al ins"   "truction ;-"    "(#n",24)),0:    n>>8==001?(      signed char
```

```
)n*2   :548==n>>    6&&usleep     /**/(10
)+n%  64==   4?0*  write  (r[7   /**/],C(
*C(*  r)),   *C(*  r+2)   )+4:  /**/ n>>9
==63   &&--r[7-n/   64%8]?n%+  /**/  64*-
2:0,         n>>6  ==47   ?*R( 0):n>>12==1?
*R(0  )=*R   (+6)  :n>>  12==+       14?*
R(0)   -=*R(2*3)    :0)n=*C(*        r);}
```

Binary Obfuscation

Fun Fact

Oftentimes, hand-written assembly code is already difficult to reverse ☺

Recursive vs. Linear-Sweep Disassembly

```
1  mov eax, [ebx]
2  call 100
3  .dword 0x42424242
4  mov edi, [eax]
5  test edi, edi
6  jne 1
7  pop eax
```

```
1  mov eax, [ebx]
2  call 104
3  inc edx
4  inc edx
5  inc edx
6  inc edx
7  mov edi, [eax]
8  test edi, edi
9  jne 1
10 pop eax
```

Disassembly Coverage?

(a) Correctly disassembled instructions.

Why Linear Sweep Works Well?

Self-Repairing Disassembly

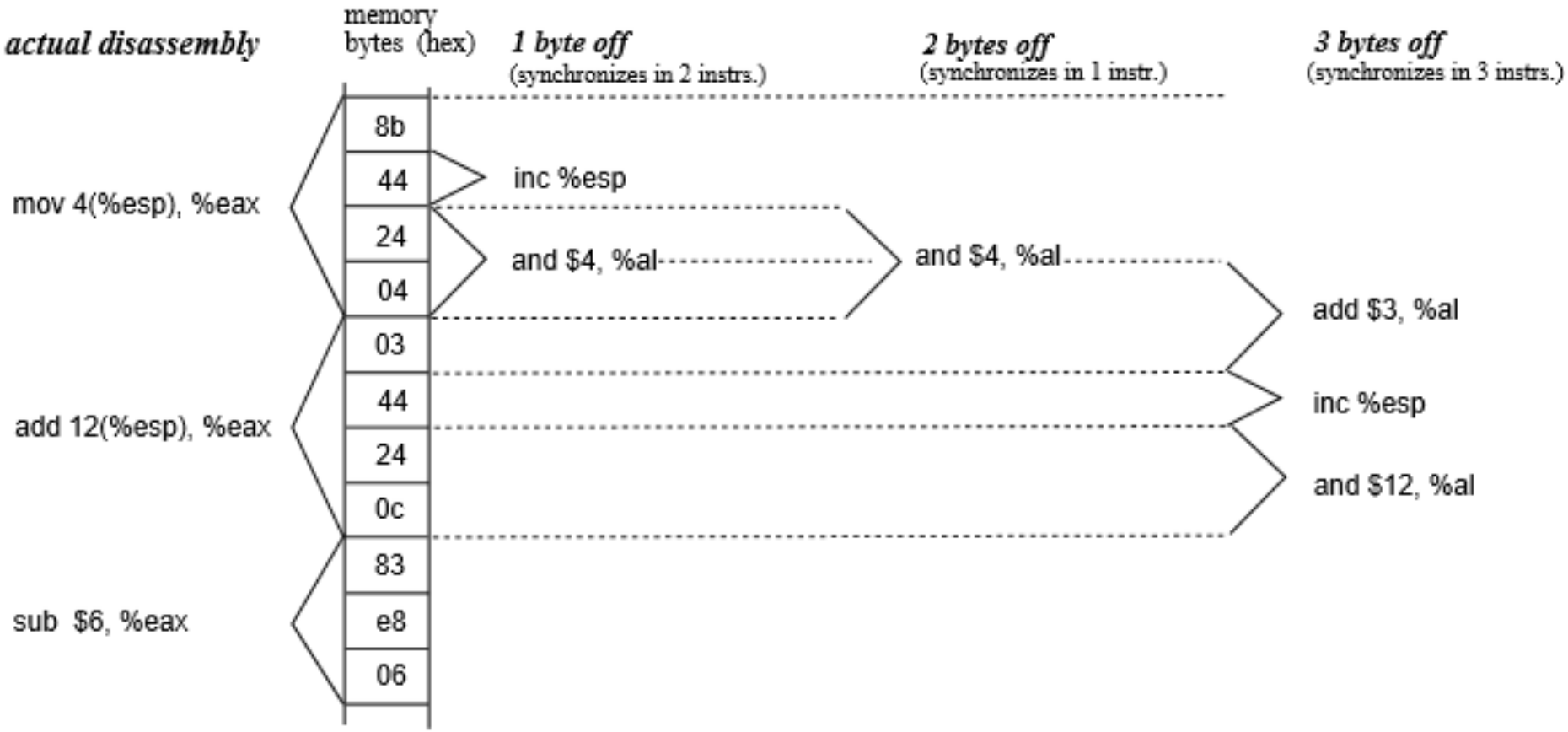


Image taken from “Obfuscation of Executable Code to Improve Resistance to Static Disassembly”, CCS 2003

Example

```
_start:
push ebx
xor ebx, ebx
je L0
.byte 0x11
L0:
pop ebx
ret
```

Example: Linear Sweep

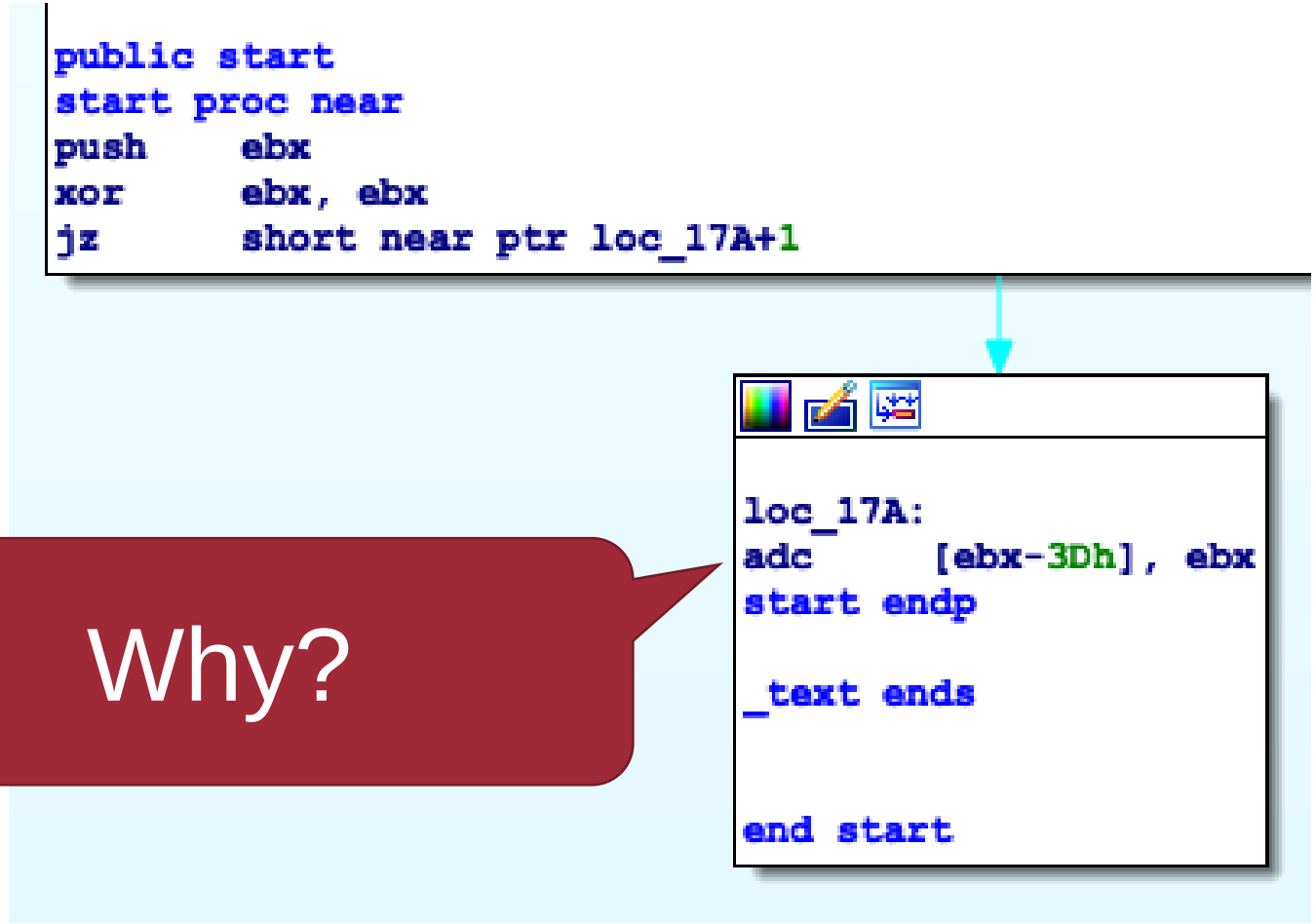
```
_start:
push ebx
xor ebx, ebx
je L0
.byte 0x11
L0:
pop ebx
ret
```

 # Output from OBJDUMP

```
00000175 <.text>:
 175: 53          push   ebx
 176: 31 db       xor    ebx,ebx
 178: 74 01       je     0x17b
 17a: 11 5b c3    adc    DWORD PTR [ebx-0x3d],ebx
```

Example: Recursive Descent

```
_start:
push ebx
xor ebx, ebx
je L0
.byte 0x11
L0:
pop ebx
ret
```



Disassembler Assumption

A sequence of bytes can only be represented in a single way

Obfuscation (1): Opaque Predicate

A variable in a program is ***opaque*** when it always has a fixed value, which is known a priori to the obfuscator, but is difficult to users (or deobfuscators) to deduce its value.

Opaque Predicates

```
int a = 5, b = 6;
int x = a + b;
if (b > 5) { ... /* dummy code */ }
if (rand() % 5 < a) { ... /* dummy code */ }
...
```

Collatz Conjecture (in 1937)

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

If we apply this function iteratively, we will always see 1 regardless of which positive integer is chosen initially.

Until now, we haven't found any counter example, but we also don't have any proof so far.

```
if ( collatz(i) == 1 ) { … }
```

Jump Table Spoofing

- Generalization of opaque predicates

- Convert a jump into an indirect jump through a jump table, where the index of the table is computed by an opaque expression that always evaluates to a single value

Obfuscation (2): Extended Loop

```
i = 1;
while (i < 100) {
  ...
  i++;
}
```

```
i = 1; j = 100;
while (i < 100
  && (j*j*(j+1)*(j+1) % 4 == 0)) {
  ...
  i++;
  j = j*i+3;
}
```

Obfuscation (3): Function Boundary Confusion

- `call` / `ret` instructions can be used for RIP-relative computations

- Difficult to decide whether a code block followed by `call` instruction is indeed a function entry or not

Knowing function boundaries is important for intra-procedural analyses or decompilation

Example Function

```
_start:
push ebx
xor ebx, ebx
call L0
L0:
pop ebx
add ebx, 0x6
push ebx
ret
pop ebx
ret
```

 # Output from OBJDUMP

```
00000175 <.text>:
 175: 53                push ebx
 176: 31 db             xor  ebx,ebx
 178: e8 00 00 00 00    call 0x17d
 17d: 5b                pop  ebx
 17e: 83 c3 06          add  ebx,0x6
 181: 53                push ebx
 182: c3                ret
 183: 5b                pop  ebx
 184: c3                ret
```

Confused Function Boundary

```
_start:
push ebx
xor ebx, ebx
call L0
L0:
pop ebx
add ebx, 0x6
push ebx
ret
pop ebx
ret
```

```
public start
start proc near
push    ebx
xor     ebx, ebx
call    $+5
pop     ebx
add     ebx, 6
push    ebx
retn
start endp ; sp-analysis failed
```

Conclusion

- Obfuscation is a way to protect software from reverse engineering.

- Finding a general way to de-obfuscate binaries is on-going research.

Questions?