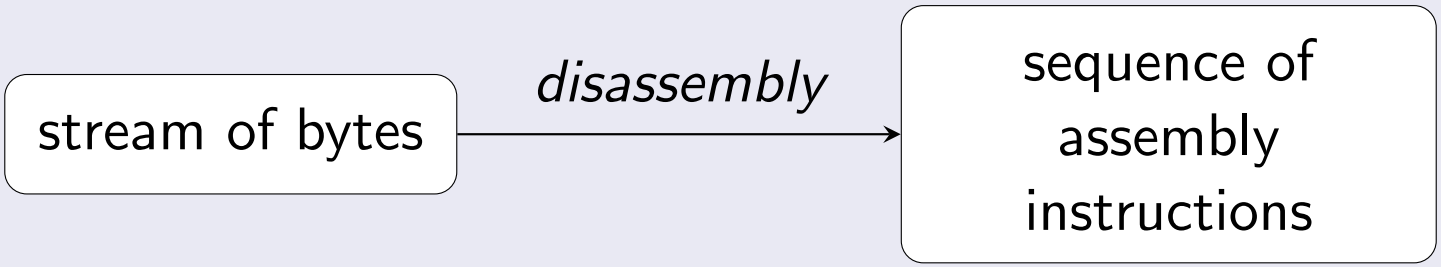
Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
Dipartimento di Informatica

Reverse engineering: *disassembly*

Lanzi Andrea <andrew@security.di.unimi.it>

A.A. 2016–2017

Cos'è?



Esempio

```
$ echo -ne "\x89\x04\x24\xc3" | ndisasm -u -
00000000  890424            mov [esp],eax
00000003  C3                ret
```

Indecidibilità della separazione tra *codice* e *dati*

Intel x86: 248/256 byte rappresentano l'inizio di un'istruzione valida

Esempio

. . . 0x81 0xc3 0x90 0x90 0x90 0x90 . . .

- se consideriamo il byte `0x81` come codice:

```
$ echo -ne \x81\xc3\x90\x90\x90\x90 | ndisasm -b32 -
00000000  81C390909090      add ebx,0x90909090
```

- se consideriamo il byte `0x81` come data e il byte `0xc3` come codice:

```
$ echo -ne \xc3\x90\x90\x90\x90 | ndisasm -b32 -
00000000  C3                ret
00000001  90                nop
00000002  90                nop
00000003  90                nop
00000004  90                nop
```

Disassembly
Problema: esplosione di complessità

```
 1  int main(int argc, char **argv)
 2  {
 3    int num;
 4
 5    scanf("%d", &num);
 6
 7    if(num == 1234)
 8      printf("ok\n");
 9    else
10      printf("no\n");
11
12    return 0;
13  }
```

Disassembly

Problema: esplosione di complessità

```
push   %ebp
mov    %esp,%ebp
sub    $0x8,%esp
call   8048304
call   8048360
call   8048490
leave
ret
pushl  0x80495dc
jmp    *0x80495e0
add    %al,(%eax)
jmp    *0x80495e4
push   $0x0
jmp    8048290
jmp    *0x80495e8
push   $0x8
jmp    8048290
jmp    *0x80495ec
push   $0x10
jmp    8048290
jmp    *0x80495f0
push   $0x18
jmp    8048290
xor    %ebp,%ebp
pop    %esi
mov    %esp,%ecx
and    $-16,%esp
push   %eax
push   %esp
push   %edx
push   $0x80483e0
push   $0x8048430
push   %ecx
push   %esi
push   $0x8048384
call   80482c0
hlt
nop
nop
push   %ebp
mov    %esp,%ebp
```

```
push   %ebx
sub    $0x4,%esp
call   8048310
pop    %ebx
add    $0x12c8,%ebx
mov    -4(%ebx),%edx
test   %edx,%edx
je     8048326
call   80482d0
pop    %eax
pop    %ebx
leave
ret
nop
...
push   %ebp
mov    %esp,%ebp
sub    $0x8,%esp
cmpb   $0x0,0x8049600
je     804834b
jmp    804835d
add    $0x4,%eax
mov    %eax,0x80495fc
call   *%edx
mov    0x80495fc,%eax
mov    (%eax),%edx
test   %edx,%edx
jne    8048341
movb   $0x1,0x8049600
leave
ret
nop
push   %ebp
mov    %esp,%ebp
sub    $0x8,%esp
mov    0x8049508,%eax
test   %eax,%eax
je     8048381
mov    $0x0,%eax
test   %eax,%eax
je     8048381
movl   $0x8049508,(%esp)
call   *%eax
```

```
leave
ret
nop
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
and    $-16,%esp
mov    $0x0,%eax
add    $0xf,%eax
add    $0xf,%eax
shr    $0x4,%eax
shl    $0x4,%eax
sub    %eax,%esp
lea    -4(%ebp),%eax
mov    %eax,0x4(%esp)
movl   $0x80484e8,(%esp)
call   80482b0
mov    -4(%ebp),%eax
cmp    $0x4d2,%eax
jne    80483cb
movl   $0x80484eb,(%esp)
call   80482a0
jmp    80483d7
movl   $0x80484ee,(%esp)
call   80482a0
mov    $0x0,%eax
leave
ret
nop
...
push   %ebp
mov    %esp,%ebp
push   %edi
push   %esi
push   %ebx
call   8048483
add    $0x11ed,%ebx
sub    $0xc,%esp
lea    -224(%ebx),%eax
lea    -224(%ebx),%edi
sub    %edi,%eax
sar    $0x2,%eax
```

```
lea    -1(%eax),%esi
cmp    $-1,%esi
je     8048419
lea    0x0(%esi),%esi
call   *(%edi,%esi,4)
dec    %esi
cmp    $-1,%esi
jne    8048410
lea    0x0(%esi),%esi
call   80484c4
add    $0xc,%esp
pop    %ebx
pop    %esi
pop    %edi
pop    %ebp
ret
lea    0x0(%esi),%esi
push   %ebp
mov    %esp,%ebp
push   %edi
push   %esi
push   %ebx
call   8048483
add    $0x119d,%ebx
sub    $0xc,%esp
call   8048278
lea    -224(%ebx),%eax
lea    -224(%ebx),%edx
sub    %edx,%eax
sar    $0x2,%eax
mov    %eax,-16(%ebp)
je     804847b
xor    %edi,%edi
mov    %edx,%esi
lea    0x0(%esi),%esi
lea    0x0(%edi),%edi
inc    %edi
call   *(%esi)
add    $0x4,%esi
cmp    %edi,-16(%ebp)
jne    8048470
```

```
add    $0xc,%esp
pop    %ebx
pop    %esi
pop    %edi
pop    %ebp
ret
mov    (%esp),%ebx
ret
nop
...
push   %ebp
mov    %esp,%ebp
push   %ebx
sub    $0x4,%esp
mov    0x80494f8,%eax
cmp    $-1,%eax
je     80484bd
mov    $0x80494f8,%ebx
lea    0x0(%esi),%esi
lea    0x0(%edi),%edi
call   *%eax
mov    -4(%ebx),%eax
sub    $0x4,%ebx
cmp    $-1,%eax
jne    80484b0
pop    %eax
pop    %ebx
pop    %ebp
ret
nop
...
push   %ebp
mov    %esp,%ebp
push   %ebx
sub    $0x4,%esp
call   80484d0
pop    %ebx
add    $0x1108,%ebx
call   8048330
pop    %ecx
pop    %ebx
leave
ret
```

Disassembly

Instruction set

- istruzioni a lunghezza fissa
  - ogni istruzione inizia ad un indirizzo multiplo della sua lunghezza
- istruzioni a lunghezza variabile
  - disassembly difficoltoso
  - il processo di disassembly può "desincronizzarsi"

Disassembly

Instruction set

- istruzioni a lunghezza fissa
  - ogni istruzione inizia ad un indirizzo multiplo della sua lunghezza
- istruzioni a lunghezza variabile
  - disassembly difficoltoso
  - il processo di disassembly può "desincronizzarsi"

Problemi

- interposizione di dati e istruzioni nello stesso address space (come distinguerli?)
- istruzioni di lunghezza variabile
- trasferimenti di controllo indiretti (function pointer, dynamic linking, jump table, ...)

Disassembly

Problemi[2]

Nella fase di compilazione si perdono:

- nomi delle variabili
- informazioni sui tipi
- concetto di "blocco"
- macro
- commenti

Problemi[3]

- riconoscere i limiti delle funzioni
- distinguere i parametri delle funzioni

Disassembly
Sintassi AT&T *vs* Intel

```
$ objdump -M att -d /bin/ls
...
push   %ebp
xor    %ecx,%ecx
mov    %esp,%ebp
sub    $0x8,%esp
mov    %ebx,(%esp)
mov    0x8(%ebp),%ebx
mov    %esi,0x4(%esp)
mov    0xc(%ebp),%esi
mov    (%ebx),%edx
mov    0x4(%ebx),%eax
xor    0x4(%esi),%eax
xor    (%esi),%edx
or     %edx,%eax
je     8049c60 <exit@plt+0x13c>
mov    %ecx,%eax
mov    (%esp),%ebx
mov    0x4(%esp),%esi
mov    %ebp,%esp
pop    %ebp
ret
```

```
$ objdump -M intel -d /bin/ls
...
push   ebp
xor    ecx,ecx
mov    ebp,esp
sub    esp,0x8
mov    DWORD PTR [esp],ebx
mov    ebx,DWORD PTR [ebp+0x8]
mov    DWORD PTR [esp+0x4],esi
mov    esi,DWORD PTR [ebp+0xc]
mov    edx,DWORD PTR [ebx]
mov    eax,DWORD PTR [ebx+0x4]
xor    eax,DWORD PTR [esi+0x4]
xor    edx,DWORD PTR [esi]
or     eax,edx
je     8049c60 <exit@plt+0x13c>
mov    eax,ecx
mov    ebx,DWORD PTR [esp]
mov    esi,DWORD PTR [esp+0x4]
mov    esp,ebp
pop    ebp
ret
```

Disassembly
Esempio

```
08048350 | B8 66 83 04 08          | mov $0x8048366,%eax
08048355 | FF D0                   | call *%eax
08048357 | C3                      | ret
08048358 | 48 65 6C 6C 6F 20 57    | (data)
0804835F | 6F 72 6C 64 21 0A 0D    | (data)
08048366 | BA 0E 00 00 00          | mov $0xe,%edx
0804836B | B9 58 83 04 08          | mov $0x8048358,%ecx
08048370 | BB 00 00 00 00          | mov $0x0,%ebx
08048375 | B8 04 00 00 00          | mov $0x4,%eax
0804837A | CD 80                   | int $0x80
0804837C | B8 00 00 00 00          | mov $0x0,%eax
08048381 | C3                      | ret
```

Tecniche utilizzate

- linear sweep disassembly
- recursive traversal disassembly

Linear sweep disassembly
(e.g., `objdump`)

- inizia al primo byte del segmento testo
- procedi decodificando un'istruzione dopo l'altra
  `$ objdump -D file`
- *assunzione*: istruzioni memorizzate in locazioni adiacenti

```
08048350 | B8 66 83 04 08    | mov $0x8048366,%eax
08048355 | FF D0             | call *%eax
08048357 | C3                | ret
08048358 | 48                | dec %eax
08048359 | 65                | gs
0804835A | 6C                | insb (%dx),%es:(%edi)
0804835B | 6C                | insb (%dx),%es:(%edi)
0804835C | 6F                | outsl %ds:(%esi),(%dx)
0804835D | 20 57 6F          | and %dl,0x6f(%edi)
08048360 | 72 6C             | jb 0x80483ce
08048362 | 64 21 0A          | and %ecx,%fs:(%edx)
08048365 | 0D BA 0E 00 00    | or $0xeba,%eax
0804836A | 00 B9 58 83 04 08 | add %bh,0x8(%ecx)
08048370 | BB 00 00 00 00    | mov $0x0,%ebx
08048375 | B8 04 00 00 00    | mov $0x4,%eax
0804837A | CD 80             | int $0x80
0804837C | B8 00 00 00 00    | mov $0x0,%eax
08048381 | C3                | ret
```

Linear sweep disassembly
(e.g., `objdump`)

- inizia al primo byte del segmento testo
- procedi decodificando un'istruzione dopo l'altra
  `$ objdump -D file`
- *assunzione*: istruzioni memorizzate in locazioni adiacenti

```
08048350 | B8 66 83 04 08       | mov $0x8048366,%eax
08048355 | FF D0                | call *%eax
08048357 | C3                   | ret
08048358 | 48                   | dec %eax
08048359 | 65                   | gs
0804835A | 6C                   | insb (%dx),%es
0804835B | 6C                   | insb (%dx),%es:(%edi)
0804835C | 6F                   | outsl %ds:(%esi),(%dx)
0804835D | 20 57 6F             | and %dl,0x6f(%edi)
08048360 | 72 6C                | jb 0x80483ce
08048362 | 64 21 0A             | and %ecx,%fs:(%edx)
08048365 | 0D BA 0E 00 00       | or $0xeba,%eax
0804836A | 00 B9 58 83 04 08    | add %bh,0x8(%ecx)
08048370 | BB 00 00 00 00       | mov $0x0,%ebx
08048375 | B8 04 00 00 00       | mov $0x4,%eax
0804837A | CD 80                | int $0x80
0804837C | B8 00 00 00 00       | mov $0x0,%eax
08048381 | C3                   | ret
```

IA-32 disassembly è *self-repairing*

Recursive traversal disassembly
(e.g., IDA Pro)

1. inizia dall'entry point
2. quando incontri un'istruzione di branch:
   - determina possibili successori
   - riprendi il disassemby a questi indirizzi

```
08048350 | B8 66 83 04 08 | mov $0x8048366,%eax
08048355 | FF D0          | call *%eax
08048357 | C3             | ret
```

Control flow graph

Cos'è?

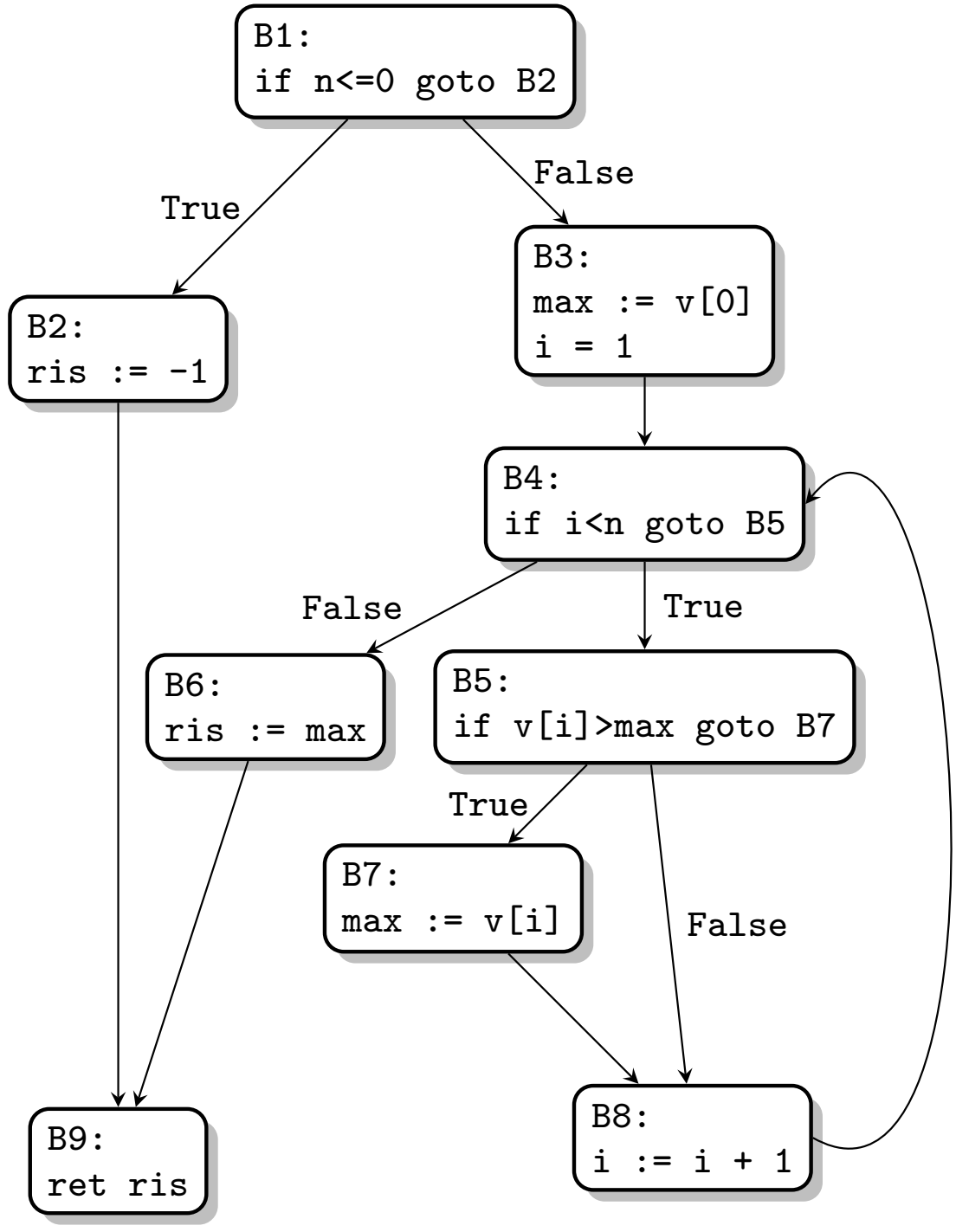
- struttura fondamentale in program analysis
- grafo orientato in cui:
  - **nodi**: *computazioni* (basic block)
  - **archi**: possibili *flussi di esecuzione*

Basic block

sequenza *non interrompibile* di istruzioni

- singolo entry point
- singolo exit point

```
 1  int find_max(int *v, int n)
 2  {
 3      unsigned int max, i;
 4      int ris;
 5
 6      if(n <= 0)
 7          ris = -1;
 8      else {
 9          max = v[0];
10          i = 1;
11          while(i < n) {
12              if(v[i] > max)
13                  max = v[i];
14              i = i + 1;
15          }
16          ris = max;
17      }
18      return ris;
19  }
```



Call graph

Cos'è?

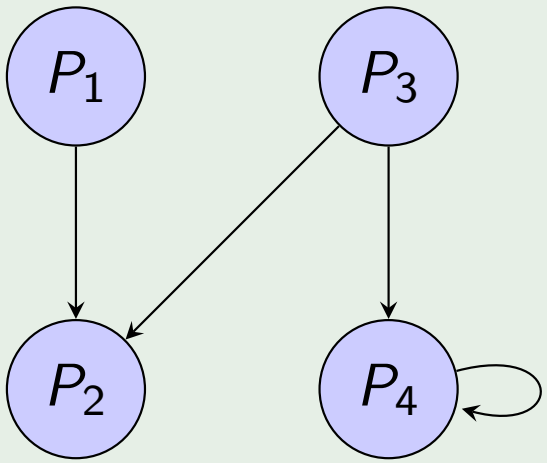
- grafo orientato $C = (R, F)$
  - i nodi rappresentano **procedure**
  - gli archi sono relazioni *chiamante-chiamato*
- se $(r_i, r_j) \in R$, nel corso di r_i si ha almeno una chiamata a r_j

Call graph

Cos'è?

- grafo orientato $C = (R, F)$
  - i nodi rappresentano **procedure**
  - gli archi sono relazioni *chiamante-chiamato*
- se $(r_i, r_j) \in R$, nel corso di r_i si ha almeno una chiamata a r_j

Esempio



Memorabilia

Regola #1 di IDA

Everything you see and you do, you are working on the database. Changes are NOT automatically reflected to the binary.

Regola #2 di IDA

There are plenty of things you can do. But there is no undo.

- `sub_xxxx`: a function at address `xxxx`
- `loc_xxxx`: an instruction at address `xxxx`
- `(byte|word|dword)_xxxx`: data at address `xxxx`
- `var_xx`: *local* variable at `EBP-xx`
- `arg_n`: *argument* at `EBP+8+n`

`https://www.hex-rays.com/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf`

Homework #3

`http://security.di.unimi.it/sicurezza1314/homeworks/homework3.tar.gz`

U

tilizzando IDA (!hexrays), *reversare* gli ELF scaricati. Ogni programma contiene una chiave che può essere individuata staticamente e confermata eseguendo il programma e fornendo in input la chiave.
Scrivere una *breve* relazione (pdf/txt), contenente, $\forall$ programma:

- la chiave individuata
- come l'avete individuata/estratta
- descrizione alto livello del programma

Bibliografia & Utilities

- G. Vigna
  Malware Detection, chapter Static Disassembly and Code Analysis
- C. Eagle
  The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler
- Intel 64 and IA-32 Architectures Software Developers Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z
- `http://ref.x86asm.net/coder32.html`