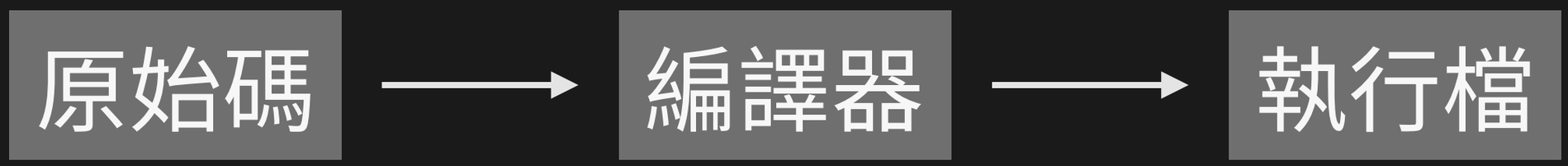
Reversing 101

從零開始的逆向工程

Inndy / inndy.tw@gmail.com

逆向工程

從成品到還原出配方的過程


原始碼
編譯器
執行檔


原始碼
執行檔

記憶體與數值

- 8 bits 稱作 1byte，寫成 16 進位是兩個數字
- 記憶體中每個 byte 都有一個編號，稱作位址（address）
- x86 的 CPU 使用 little-endian 編碼把數值存入記憶體
  - 0x0123456789ABCDEF（8 bytes 整數）
  - EF, CD, AB, 89, 67, 45, 23, 01

記憶體與變數

- 程式中的變數都會被分配到一個位址上
  - 變數的位址就是第一個 byte 所在的位址
    - 變數的型態表示你要怎麼去解讀這個位址的資料
- C語言的陣列使用連續的記憶體空間
- 字串其實就是 char 的陣列，通常以一個 '\0' 結尾

記憶體與變數

```c
#include <stdio.h>
#include <stdint.h>

uint64_t array[3] = {
    0x0123456789ABCDEF,
    0xAABBCCDD99887766,
    0x1111222233445566
};

int main()
{
    printf("&array = %p\n", &array);
    for(int i = 0; i < 3; i++)
        printf("&array[%d] = %p\n", i, &array[i]);
}
```

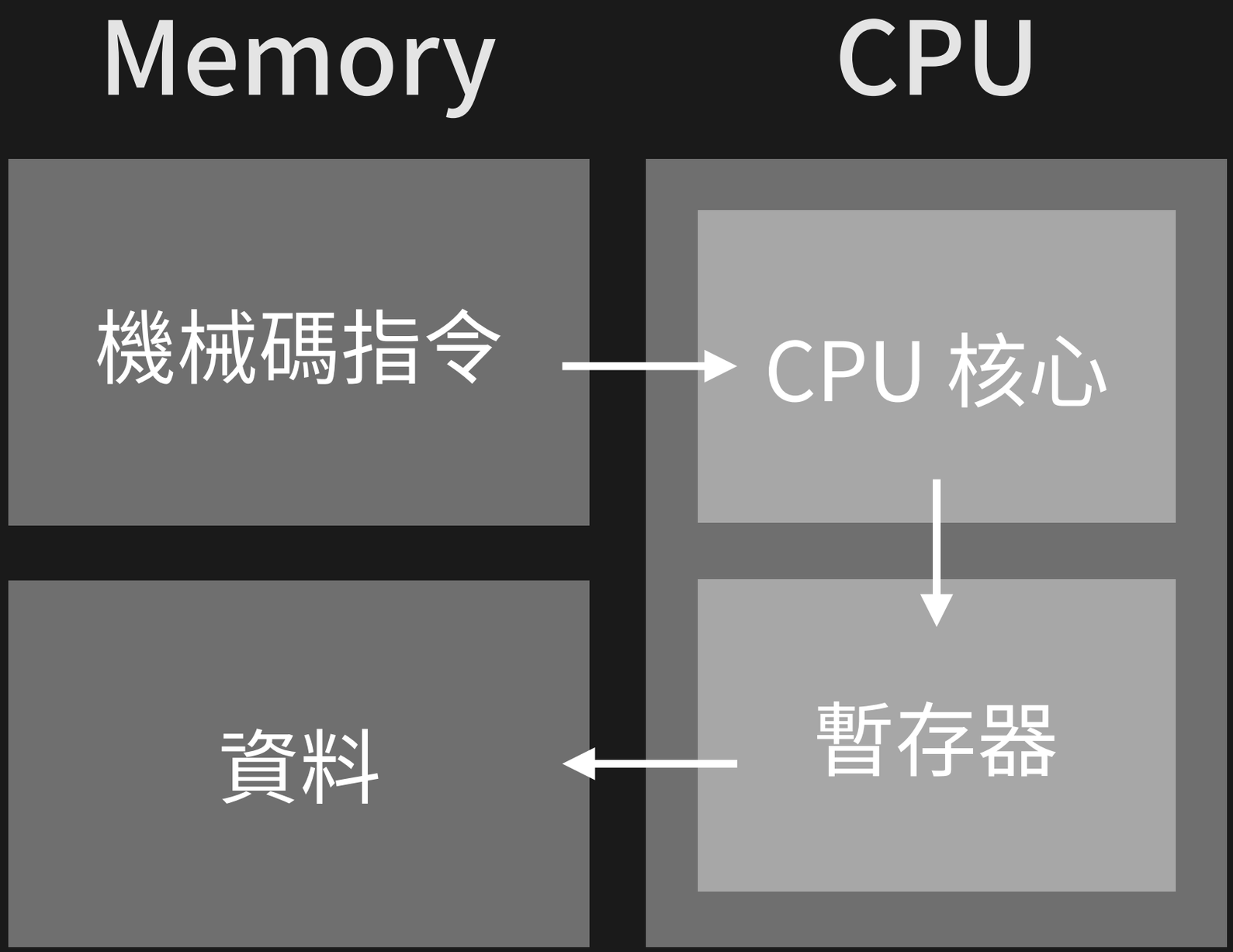
記憶體與變數

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000:																
0010:																
0020:	EF	CD	AB	89	67	45	23	01								
0030:																
0040:																
0050:																
0060:																
0070:																

記憶體與變數

- 指標就是指向變數第一個 byte 的 address
- 在 C 語言中，把指標加一表示指向下一個元素
  - 如果指標型別是 char * 才會剛好把位址加一

CPU 的運作



暫存器（Register）

- 通用：RAX, RBX, RCX, RDX, RDI, RSI, R8, R9…, R15
- R 開頭都是 8 bytes 整數暫存器
- 特殊暫存器：
  - RSP, RBP：指向堆疊區（Stack）
  - RIP：指向現在正在執行的指令位址
  - EFLAGS：紀錄比較運算結果以及其他狀態

暫存器（Register）

- 除了 R 開頭的暫存器，還有一些可以只取部分的用法：

```
                    AL
1111111122223344
                  = AX
1111111122223344
                ==== EAX
1111111122223344
============== RAX
1111111122223344
```

組合語言與機器碼

```
  位址     機械碼                    組合語言

0000000000401560 <main>:
  401560: 55                        push   rbp
  401561: 48 89 e5                  mov    rbp,rsp
  401564: 48 83 ec 20               sub    rsp,0x20
  401568: e8 a3 01 00 00            call   401710 <__main>
  40156d: 48 8d 0d 8c 2a 00 00      lea    rcx,[rip+0x2a8c]
  401574: e8 c7 16 00 00            call   402c40 <puts>
  401579: b8 00 00 00 00            mov    eax,0x0
  40157e: 48 83 c4 20               add    rsp,0x20
  401582: 5d                        pop    rbp
  401583: c3                        ret
```

組合語言入門

組合語言	C 語言
`push rbp`	`push(rbp);`
`mov rbp,rsp`	`rbp = rsp;`
`sub rsp,0x20`	`rsp -= 0x20;`
`call 401710`	`f_401710();`
`lea rcx,[rip+0x2a8c]`	`rcx = rip + 0x2a8c;`
`call 402c40`	`f_402c40();`
`mov eax,0x0`	`eax = 0;`
`add rsp,0x20`	`rsp += 0x20;`
`pop rbp`	`pop(&rbp);`
`ret`	`return;`

組合語言入門

組合語言	C 語言
`add rax, rbx`	`rax += rbx;`
`sub rbx, rsp`	`rbx -= rsp;`
`inc qword ptr [rax+4]`	`++ *(uint64_t*)(rax+4);`
`xor ecx, dword ptr [rax]`	`ecx ^= *(uint32_t*)rax;`
`mov rax, 1`	`rax = 1;`
`mov rdi, rdx`	`rdi = rdx;`
`mov byte ptr [rcx], al`	`*(char*)rcx = al;`
`mov ax, word ptr [rdi+3]`	`ax = *(short*)(rdi+3);`

組合語言入門

組合語言

```
push rax

pop rax
```

組合語言(?)

```
sub rsp, 8
mov qword ptr [rsp], rax

mov rax, qword ptr [rsp]
add rsp, 8
```

組合語言入門

組合語言	組合語言(?)
`jmp $addr`	`mov rip, $addr`
`jmp qword ptr [rax+8]`	`mov rip, qword ptr [rax+8]`
`call $func`	`push $NEXT_INSTRUCTION` `jmp $func`
`ret`	`pop rip`

條件跳躍

- cmp 指令後面會接著 j 系列的條件條約指令
- 根據上一次比較的結果決定要不要跳過去
  - je: Jump if Equal
  - jne: Jump if Not Equal
  - jg: Jump if Greater
  - jl: Jump if Lower

條件跳躍

組合語言

```
cmp rax, 5
je is_five
call not_five
jmp end
is_five:
call five
end:
```

C 語言

```
if (rax == 5)
  goto is_five;
not_five();
goto end;
is_five:
five();
end:
```

條件跳躍

組合語言

```
cmp rax, 5
je is_five
call not_five
jmp end
is_five:
call five
end:
```

C 語言

```
if (rax != 5)
{
  not_five();
} else {

  five();
}
```

條件跳躍

組合語言

```
cmp rax, 5
je is_five
jg greater_five
call not_five
jmp end
greater_five:
call six_or_more
is_five:
call five
end:
```

C 語言

```
if (rax == 5)
{
  five();
} else if (rax > 5)
{
  six_or_more();
  five();
} else {
  not_five();
}
```

條件跳躍

組合語言

```
mov rcx, 0
mov rax, 0
counter:
cmp rax, 0xa
jg end
inc rax
add rcx, rax
jmp counter
end:
```

C 語言

```
rcx = 0;
rax = 0;

while (rax <= 10)
{
  rax++;
  rcx += rax;
}
```

條件跳躍

- cmp $x, $y 其實會計算 $x - $y，然後把結果存下來
- 並不是直接存計算結果的數字
- 把「計算結果的狀態」存在 EFLAGS 暫存器內
- 有沒有 overflow？是否為負數？...

Function Call

- CALL / RET 指令成對使用
- 回傳值放在 RAX
- 返回的時候，利用堆疊的機制，知道跳回那裡
- 問題：Stack / Queue？

Stack vs Queue

Stack vs Queue

Credit to @arenamontanus on flickr.com

Function Call

Assembly Stack

```
> 0040: call 0080
  0045: add rax, 1
________________________
  0080: push 1
  0082: pop rax
  0083: inc rax
  0084: ret                    > ffff
```

Function Call

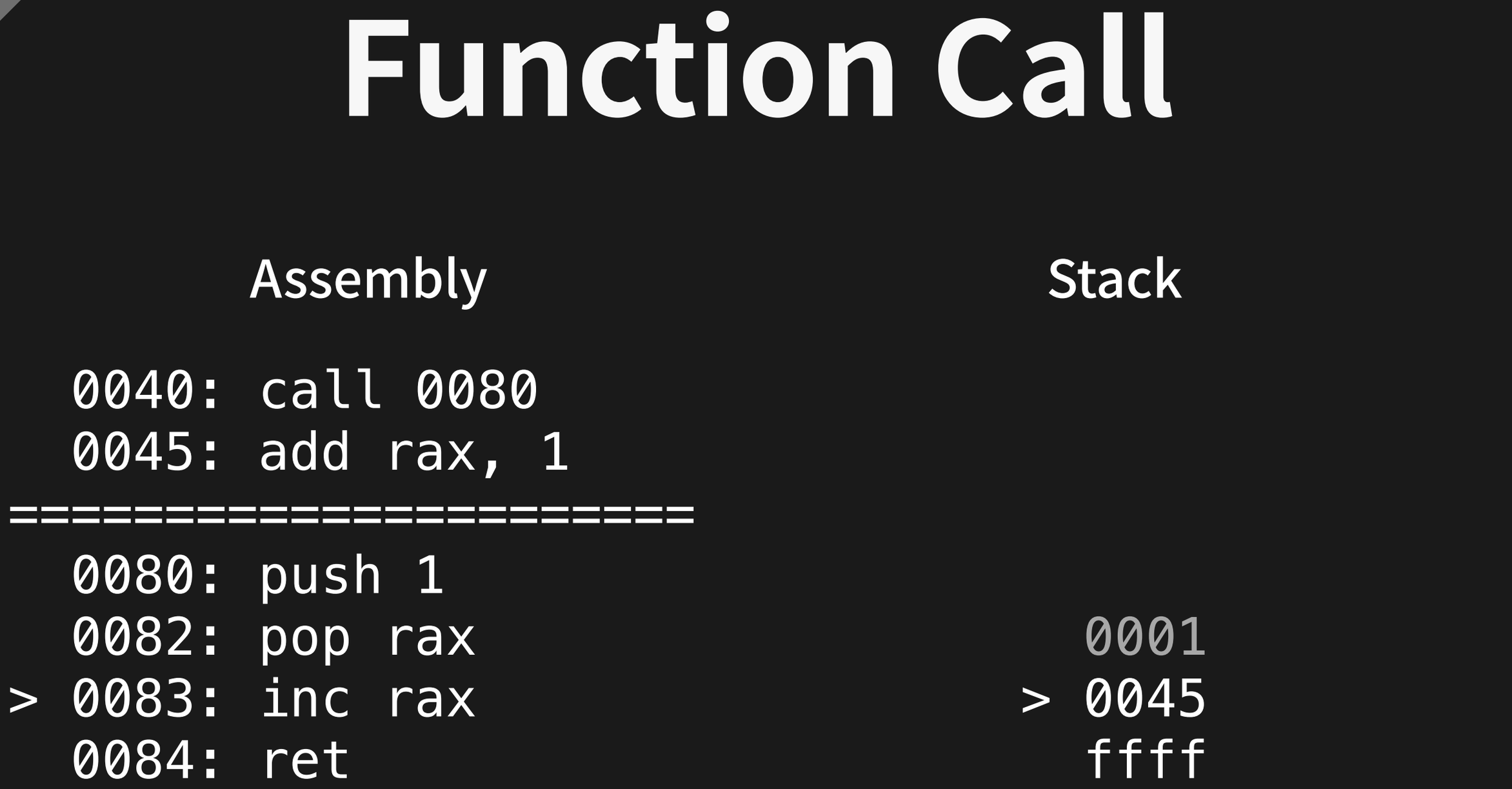
Assembly | Stack

```
  0040: call 0080
  0045: add rax, 1
________________________
> 0080: push 1
  0082: pop rax
  0083: inc rax                    > 0045
  0084: ret                          ffff
```

Function Call

Assembly	Stack
0040: call 0080	
0045: add rax, 1	
0080: push 1	
> 0082: pop rax	> 0001
0083: inc rax	0045
0084: ret	ffff

Function Call



Function Call

Assembly

```
  0040: call 0080
  0045: add rax, 1
________________________
  0080: push 1
  0082: pop rax
  0083: inc rax
> 0084: ret
```

Stack

```
0001
> 0045
  ffff
```

Function Call

Assembly

```
  0040: call 0080
> 0045: add rax, 1
______________________
  0080: push 1
  0082: pop rax
  0083: inc rax
  0084: ret
```

Stack

```
0001
0045
> ffff
```

Calling Convention

- 呼叫副程式的時候，參數要怎麼給？
  - 以下是 x86 64 bits 的狀況，32 bits 基本上都放堆疊
  - Windows：
    - 先放 RCX, RDX, R8, R9，其他依序放入堆疊
  - Linux & Mac：
    - 先放 RDI, RSI, RDX, RCX, R8, R9，其他依序放入堆疊

Calling Convention

C 語言

```
printf(
  "%d + %d = %d\n”,
1, 2, 3);
```

組合語言

```
mov rcx, $format
mov rdx, 1
mov r8, 2
mov r9, 3
call printf
```