

LigthBranch : Binary fuzzing with snapshot-assisted-driven comparison branches analysis

Kijong Son
KISA

About me

- Kijong Son
- Security researcher @ KISA
- Penetration testing Instructor
 - Teaching courses
- Past Experiences
 - Penetration tester for 10+ years
 - Bug bounty program management
- Focusing on vulnerability and exploitation research

Agenda

- Motivation
- Introducing LightBranch
- Snapshot mechanism for input generation
- How we analyze comparison branches
- DEMO

Motivation

- Fuzzer tend to get stuck in the input validation code.
- Need to generate feedback information to guide fuzzer
- Time consuming to manually make a input dictionary.
 - Some mutation-based fuzzer supports user-supplied dictionaries
 - But In order to make a dictionary, It requires manual effort
- Automatic valid input generation for fuzzing

Random | Feedback

american fuzzy lop 2.52b (8000)

process timing	overall results
run time : 0 days, 1 hrs, 0 min, 0 sec	cycles done : 20.9k
last new path : none yet (odd, check syntax!)	total paths : 1
last uniq crash : none seen yet	uniq crashes : 0
last uniq hang : none seen yet	uniq hangs : 0
cycle progress	map coverage
now processing : 0 (0.00%)	map density : 0.05% / 0.05%
paths timed out : 0 (0.00%)	count coverage : 1.00 bits/tuple
stage progress	findings in depth
now trying : havoc	favored paths : 1 (100.00%)
stage execs : 210/256 (82.03%)	new edges on : 1 (100.00%)
total execs : 5.35M	total crashes : 0 (0 unique)
exec speed : 1503/sec	total tmouts : 7 (5 unique)
fuzzing strategy yields	path geometry
bit flips : 0/32, 0/31, 0/29	levels : 1
byte flips : 0/4, 0/3, 0/1	pending : 0
arithmetics : 0/224, 0/0	pend fav : 0
known ints : 0/22, 0/83, ...	own finds : 0
dictionary : 0/0, 0/0, 0/0	imported : 0
havoc : 0/5.35M, 0/0	stability : 100.00%
trim : 66.67%/2, 0.00%	

^C [cpu000:201%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

```
#include<stdio.h>

void vuln(char *buf) {
    char arr[64] = {0};
    strcpy(arr,buf);
    return;
}

void main() {
    char buf[1337] = {0};
    char *str = "findme";
    read(0,buf,sizeof(buf));
    if(!strcmp(str,buf,6)) {
        printf("correct! go to vuln function!\n");
        vuln(buf);
    }
    return;
}
```

american fuzzy lop 2.52b (8000)

process timing	overall results
run time : 0 days, 0 hrs, 0 min, 34 sec	cycles done : 23
last new path : 0 days, 0 hrs, 0 min, 34 sec	total paths : 2
last uniq crash : 0 days, 0 hrs, 0 min, 16 sec	uniq crashes : 1
last uniq hang : 0 days, 0 hrs, 0 min, 26 sec	uniq hangs : 1
cycle progress	map coverage
now processing : 0 (0.00%)	map density : 0.05% / 0.07%
paths timed out : 0 (0.00%)	count coverage : 1.00 bits/tuple
stage progress	findings in depth
now trying : splice 13	favored paths : 2 (100.00%)
stage execs : 20/32 (62.50%)	new edges on : 2 (100.00%)
total execs : 35.8k	total crashes : 1 (1 unique)
exec speed : 1060/sec	total tmouts : 4 (1 unique)
fuzzing strategy yields	path geometry
bit flips : 0/96, 0/94, 0/90	levels : 2
byte flips : 0/12, 0/10, 0/6	pending : 0
arithmetics : 0/671, 0/0	pend fav : 0
known ints : 0/70, 0/27, ...	own finds : 1
dictionary : 0/2, 1/12, 0/0	imported : 0
havoc : 0/13.6k, 1/20.5k	stability : 100.00%
trim : 45.45%/4, 0.00%	

^C [cpu001:264%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

Random mutation

Feedback guided mutation

Comparison branch

Interesting inputs

- Pre-defined Inputs that are required by program
 - Option, Command
 - File format
 - Protocol spec
- They tends to be compared at the front end of a program
- Play a big role in detecting new path during fuzzing

Input generation

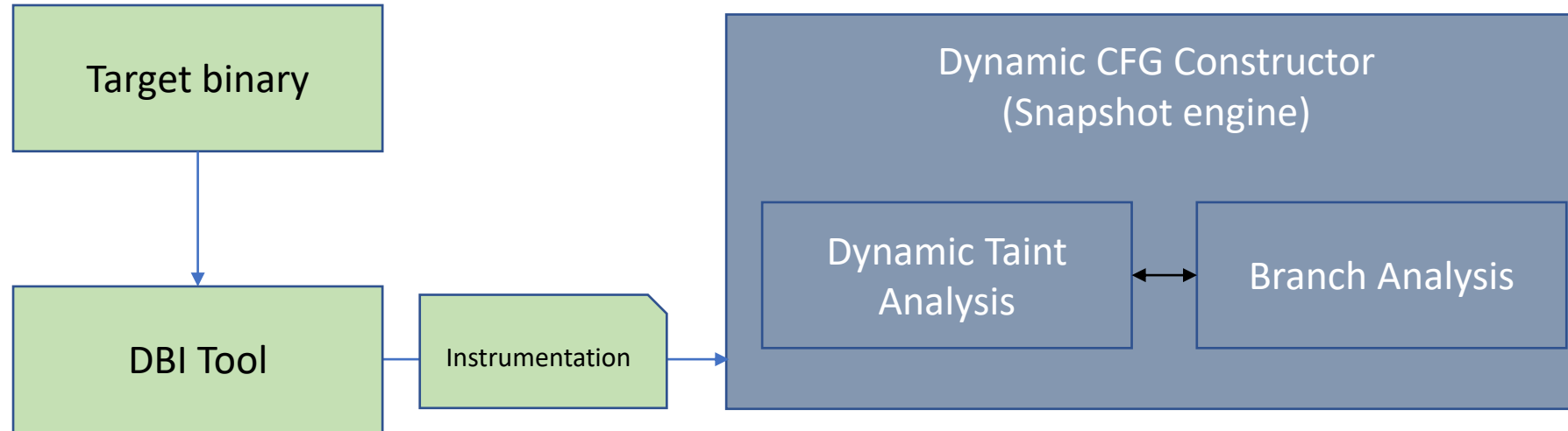
- Make a dictionary file
- Symbolic / Concolic execution
- Collect seed templates from web crawling
- Static/Dynamic binary or source code analysis

Our approach

Input learning with snapshot based comparison branches analysis

LightBranch Design

- LightBranch consists of three major components
 - (1) Dynamic CFG Constructor, (2) Taint analysis, (3) Branch analysis



Why snapshot?

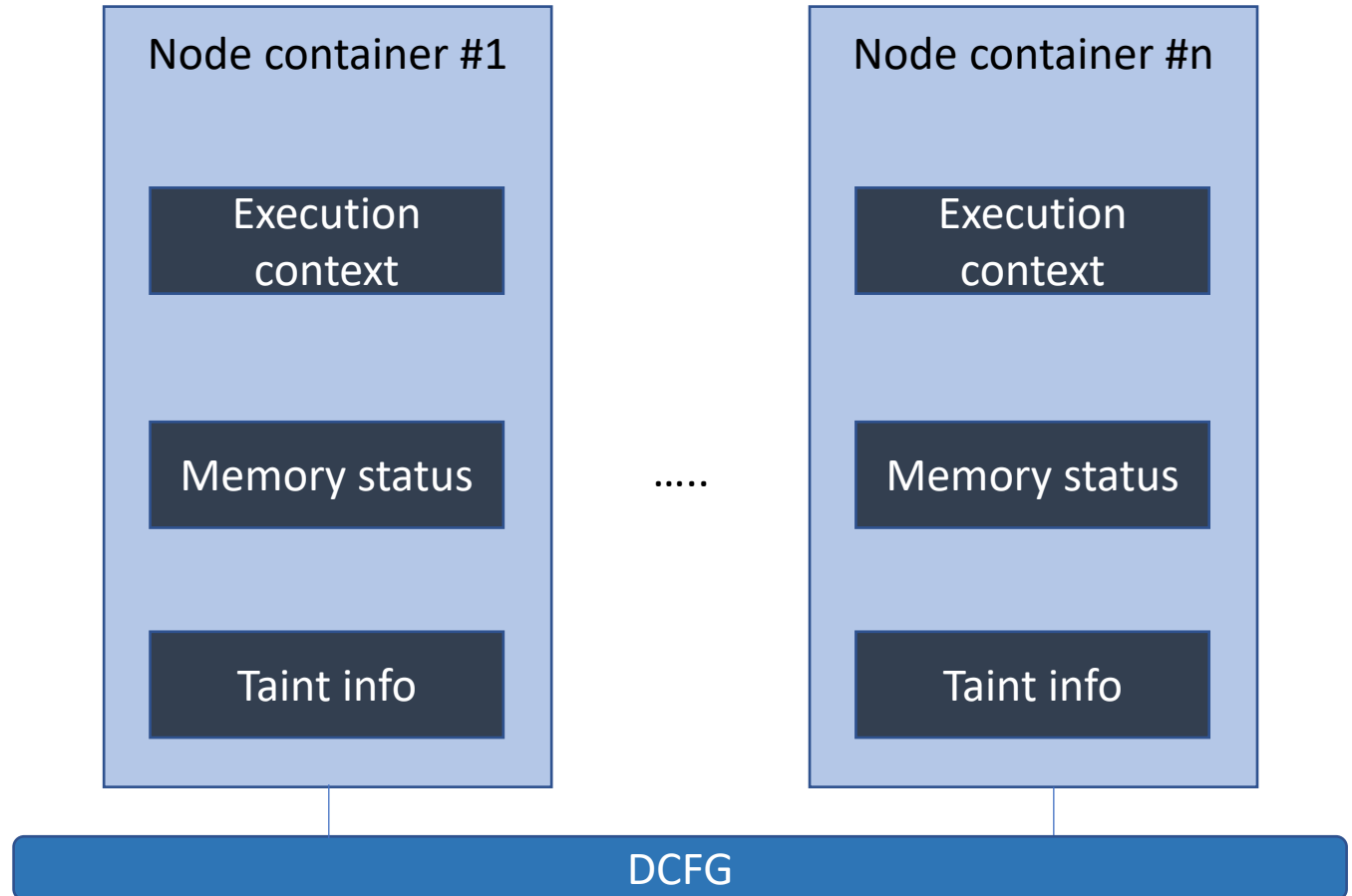
- Skip over unnecessary process startup code
- Execute both directions of conditional branch
- Extend taint propagation coverage
- More access to comparison branch with in-memory processing

Snapshotting with Dynamic CFG

- Generate dynamic control flow graph nodes
- Only conditional branch's basic block is treated as node
- Each node represents a snapshot. It has a snapshot information
- Restoring a snapshot by referencing graph nodes
- Managing snapshot and restore scheduling

Node container internals

- Snapshot repository
 - Execution Context
 - Memory Status
 - Taint Information

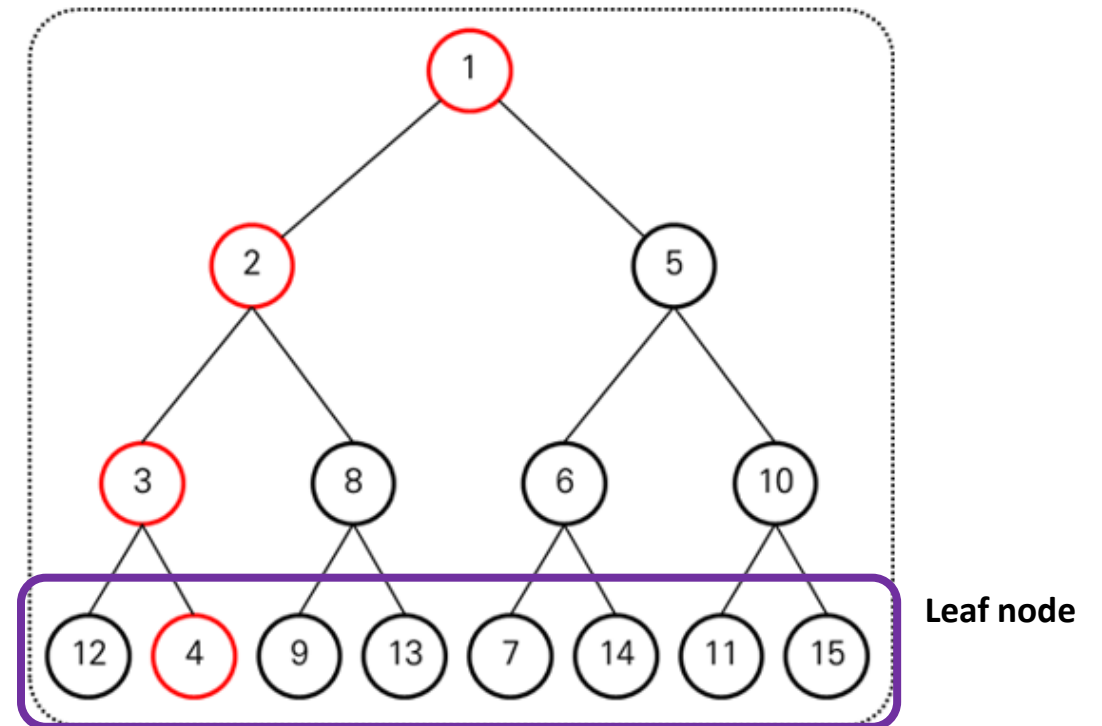


Snapshot creation flow

- Instrument head instruction of conditional branch's basic block
- Take a snapshot of runtime state of conditional branch
- Create node container to save snapshot information
- Manage all snapshots with CFG tree
- Restore snapshot under predefined conditions

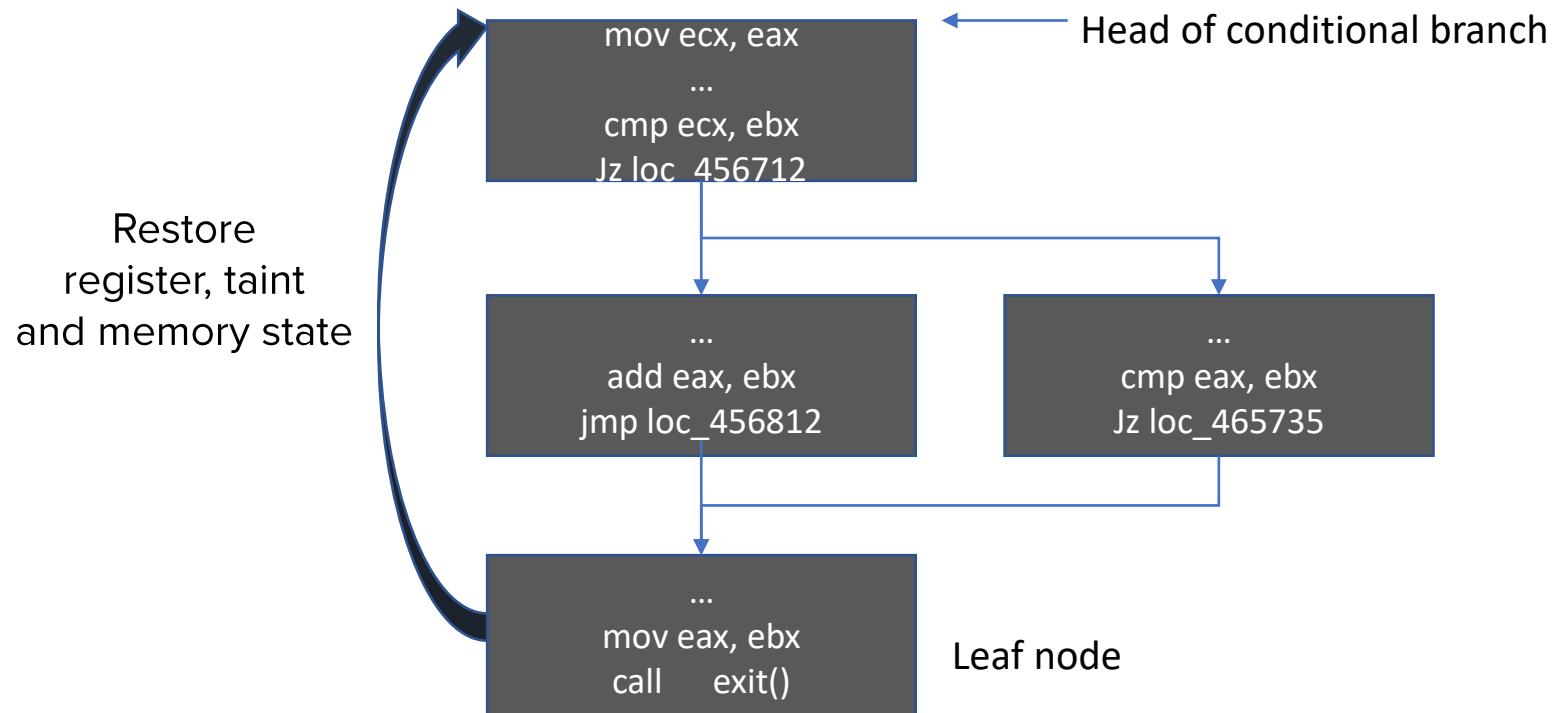
Restore snapshot

- The key idea for restoring snapshot is to detect leaf node.
- Leaf node that doesn't have child node
 - The end address of main function
 - Program exit functions are called
 - Exception signals are generated
 - Invalid instructions



Restore location

- Where is destination address for restoring?
 - The head of conditional branch's basic block



Snapshot rules

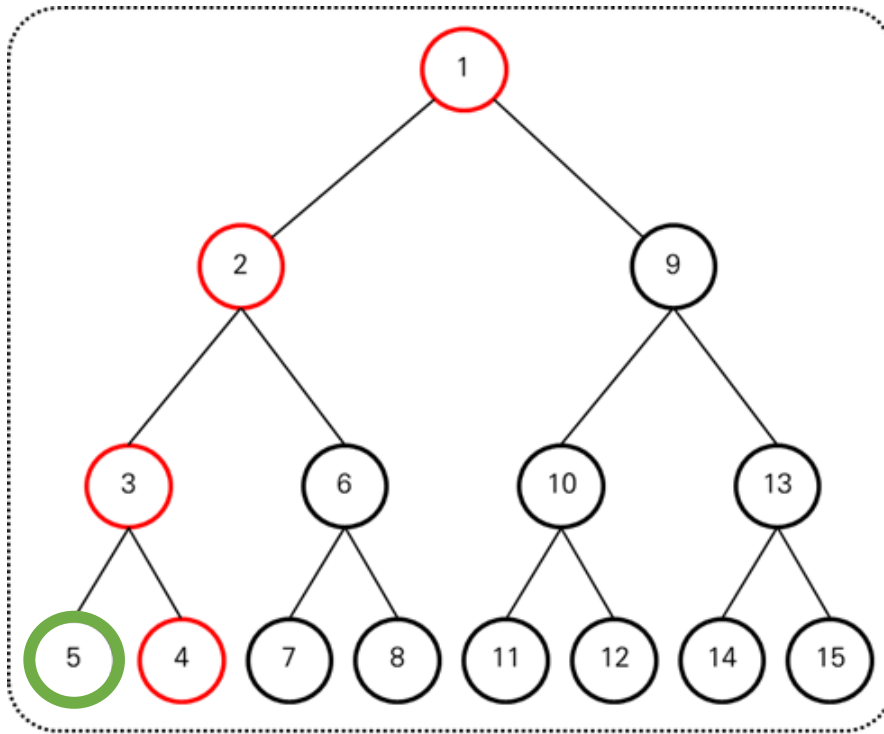
- Doesn't take a snapshot for first basic block right after restoring
- The restored node is deleted from the node list
- (optional) Set depth of the deepest node level
- (optional) Allow the redundant snapshot mode

Memory snapshot

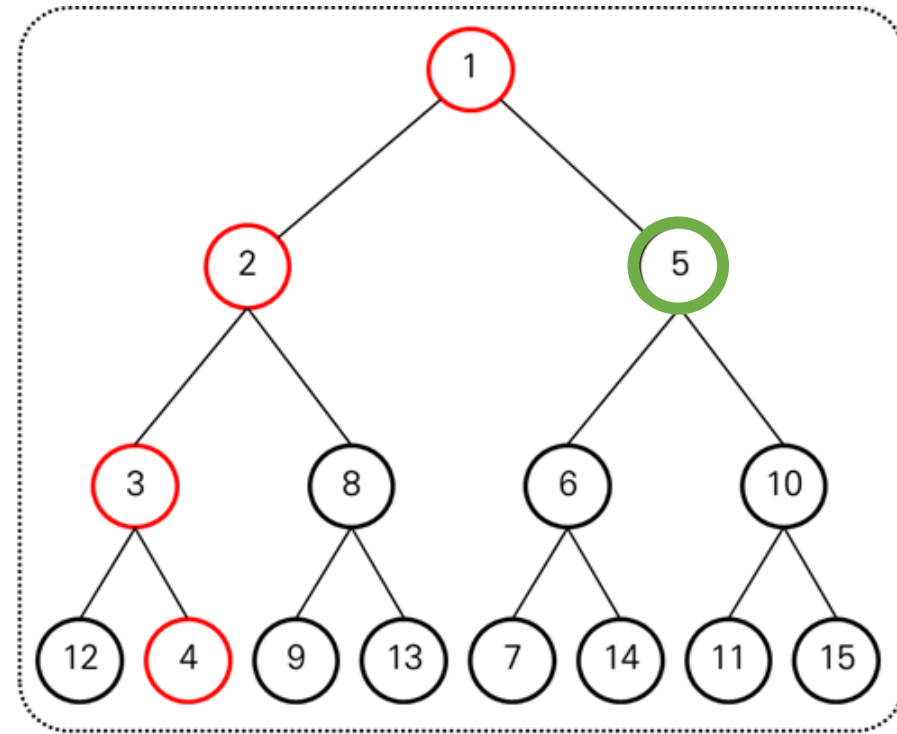
- Instrument the memory-writing instructions on trace level
- Preserve the original value of memory before writing
 - From the beginning of each conditional branch to right before being restored
- Save memory snapshot on each node container
- Memory snapshot rule
 - If a value is written to the same address multiple times, record only first original value in same node

Tree traversal for restore

- There are 2 cases of snapshot tree traversal



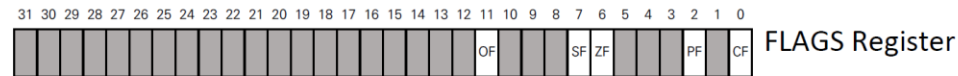
Bottom Up Restore



Top Down Restore

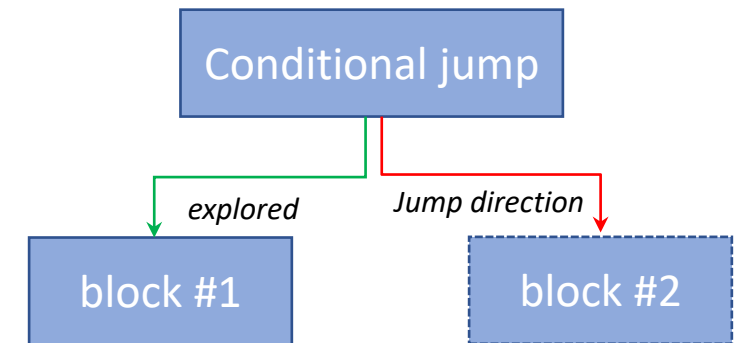
Control flow hijacking

- Check current flag register and then determine the jump direction



Instructions	Flags
JO	OF = 1
JNO	OF = 0
JS	SF = 1
JNS	SF = 0
JE JZ	ZF = 1
JNE JNZ	ZF = 0
JB JNAE JC	CF = 1
JNB JAE JNC	CF = 0
JBE JNA	CF = 1 or ZF = 1

Instructions	Flags
JA JNBE	CF = 0 and ZF = 0
JL JNGE	SF <> OF
JGE JNL	SF = OF
JLE JNG	ZF = 1 or SF <> OF
JG JNLE	ZF = 0 and SF = OF
JP JPE	PF = 1
JNP JPO	PF = 0
JCXZ JECXZ	%CX = 0 %ECX = 0



Validation check

- Read or Write Memory access
 - Collect address ranges from /proc/[PID]/maps file
 - Update address ranges because of dynamic memory allocation
 - Check invalid memory access
- Indirect call address
 - Get a register value and check if address is in code sections
- Null point access
- Double free and invalid free pointer

Snapshot for loop body

- Loop detection
 - Backward jump to address
 - Also check if jump address is greater than function's start address
- (optional) Set loop iteration threshold to escape loop
 - To avoid unnecessary loop iteration
 - Count the number of execution times of backward jump
 - Restore snapshot if the threshold is reached

Comparison branch

- Compare with two operands and then jump somewhere

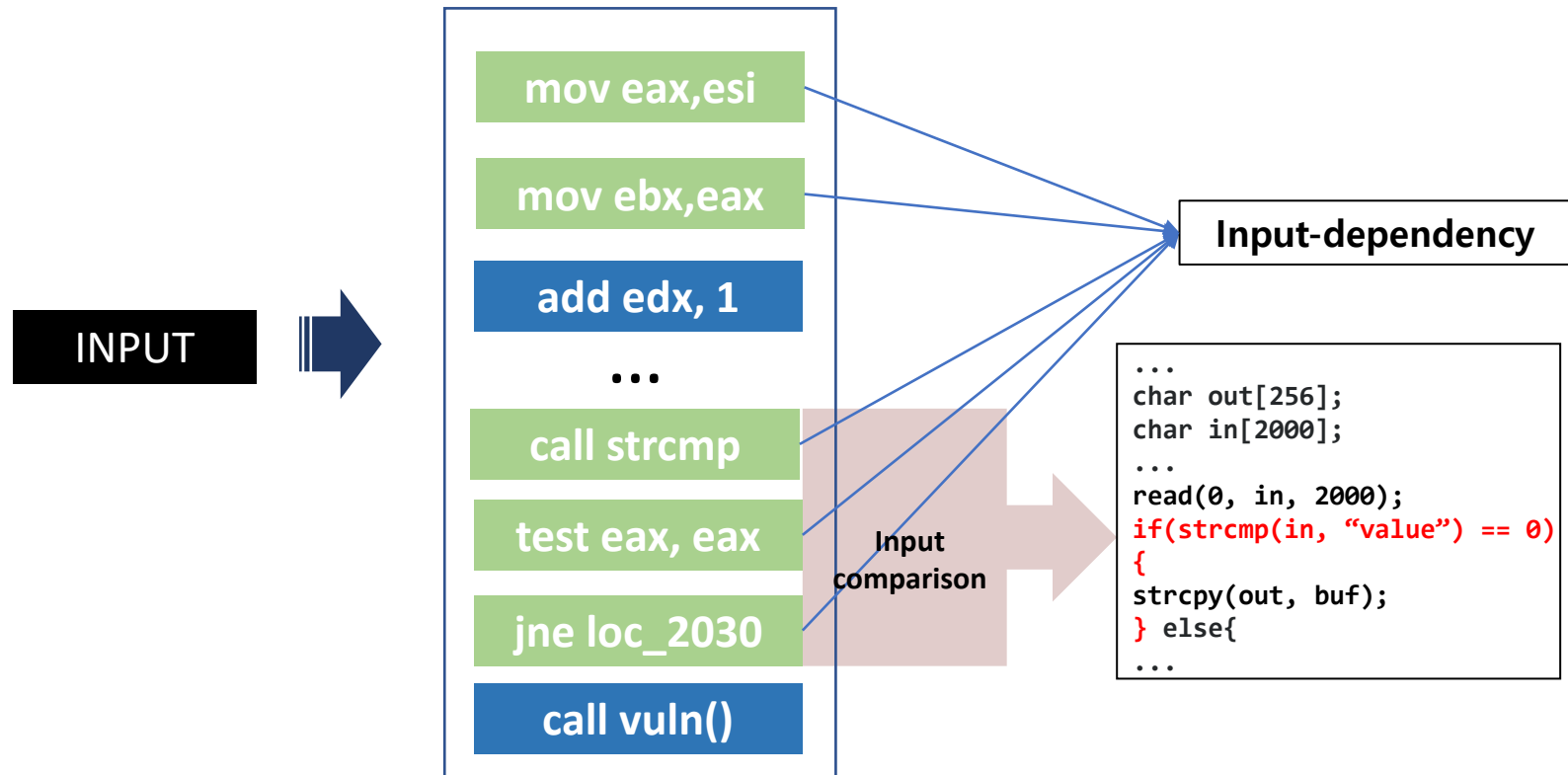
```
if(comparison_is_true) {  
    do_something;  
}  
else {  
    do_something;  
}
```

```
switch(expression) {  
    case x:  
        do_something  
    case y:  
        do_something  
    default:  
        do_something;  
}
```

- Use Cases : Single branch, Nested branches, Branch in the loop

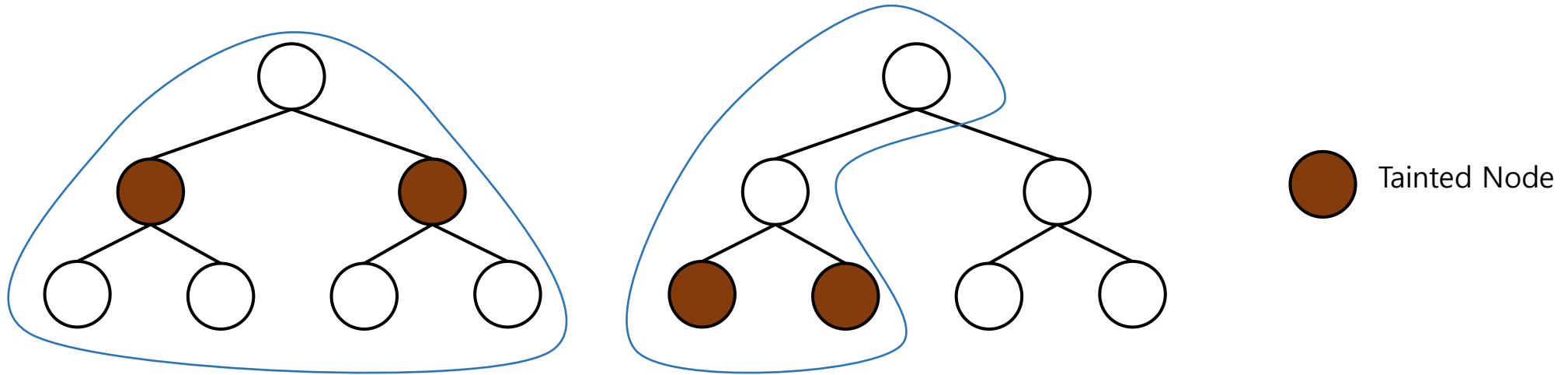
Input-dependency branch

- Dynamic taint propagation



Marking tainted node

- Tainted node in snapshot tree

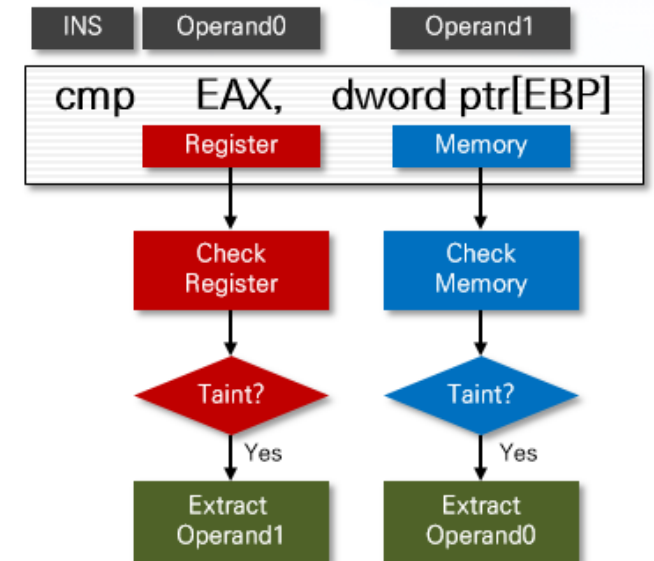


Extract comparison value

- Instrument compare instructions and functions
 - CMP and TEST assembly instruction
 - cmp, cmps, cmpsb, cmpsw, cmpsd, cmpsq, test
 - Repeat prefix instruction set(repe, repz, repne, repnz)
 - CMPSB, CMPSW, CMPSD, SCASB, SCASW, SCASD can be preceded by the rep prefix
 - Repeat execution of string instruction the number of times specified in counter register
- *cmp library functions
 - memcmp
 - strcmp family

Extract comparison value

- Identify location which actually has comparison value.
- Which operands are tainted at comparison time
 - We need to identify non-tainted operand
- Check operand type of 'non-tainted' operand
 - Register, memory and immediate value
- Extract value of non-tainted operand according to type
 - CMP → Get register, memory or immediate value
 - Rep prefix → Get memory(RAX, RDI, RSI) with ECX
 - *cmp function → Get argument



Support

- In Scope
 - Raw value of target operand
- Out of Scope
 - Compare it with transformed input
 - Dynamically encoded or encrypted
 - And there is no original of comparison value
 - No comparison target value
 - Get function pointer only by user input

```
static int do_cmd(LHASH_OF(FUNCTION) *prog, int argc, char *argv[])
{
    FUNCTION f, *fp;

    if (argc <= 0 || argv[0] == NULL)
        return 0;
    f.name = argv[0];
    fp = lh_FUNCTION_retrieve(prog, &f);
    if (fp == NULL) {
        if (EVP_get_digestbyname(argv[0])) {
            f.type = FT_md;
            f.func = dgst_main;
            fp = &f;
        } else if (EVP_get_cipherbyname(argv[0])) {
            f.type = FT_cipher;
            f.func = enc_main;
            fp = &f;
        }
    }
    if (fp != NULL) {
        return fp->func(argc, argv);
    }
}
```

input processing in OPENSSL

1. Get digest module object only by user input (It doesn't compare)

2. Call func pointer of digest (jump to the new path)

Comparison of values from offset

- Extract the offset of 'tainted' operand
 - For that, check whether tainted operand uses index addressing before comparison
- Type of offset
 - Indirect offset → Index register
 - Direct offset → Constant, Immediate value
- Offset type is determined at compile time

```
if(!strcmp(input[10], "conf", 4)) { something; }  
  
/*  
push 0x4  
push 0x80485d0 ; 'conf'  
lea  eax,[ebp-0x48]  
add  eax, 0xa           ; add addressing  
push eax  
call 0x8048380 <strcmp@plt>  
add  esp,0x10  
test eax, eax  
jne  0x804853e <main+115>  
*/
```

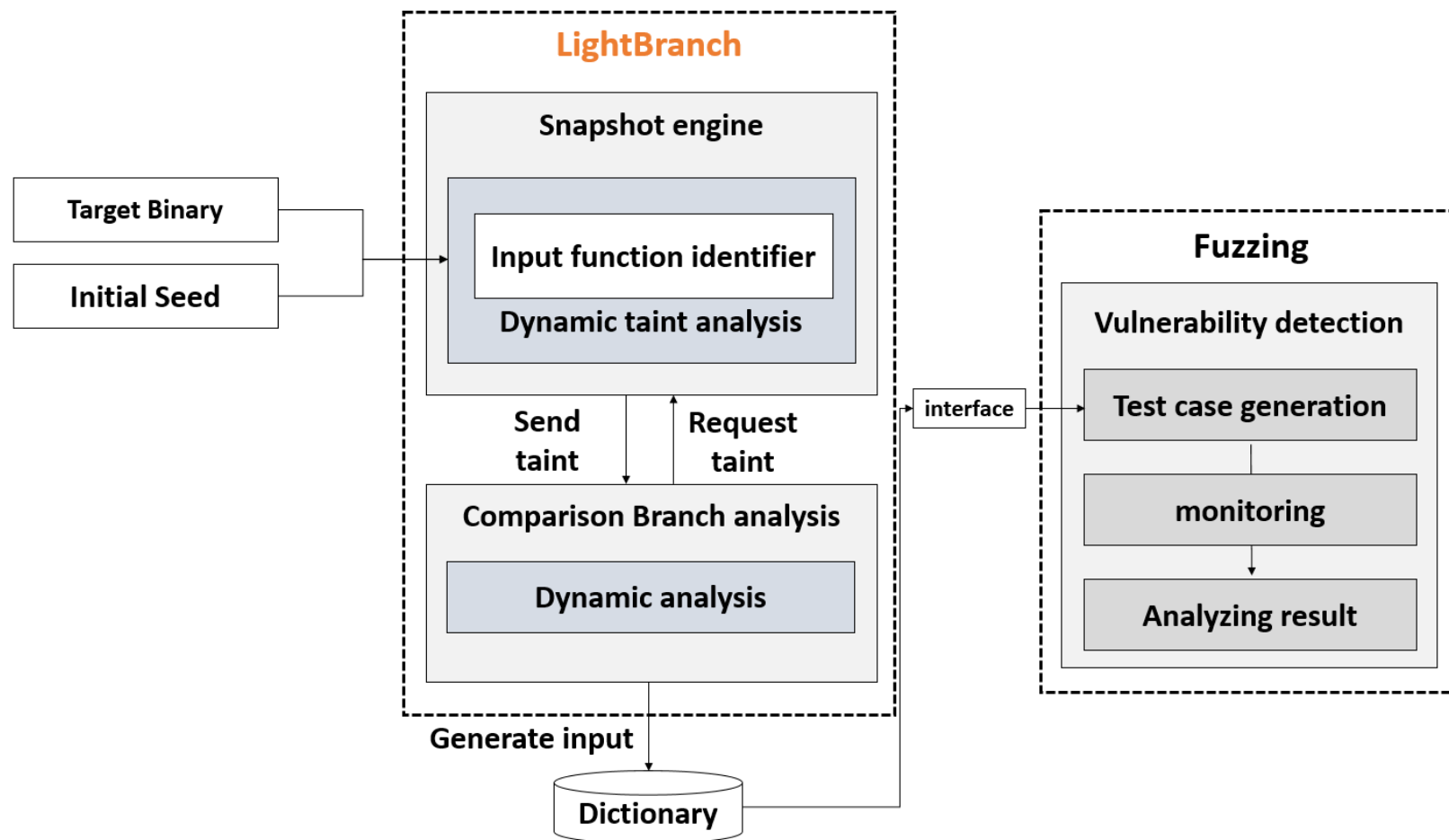
How we extract offset

- Use Backward taint analysis from tainted branch
- Which operands are tainted?
- Check the index addressing modes at a nearby basic blocks
 - Stack addressing, indirect/direct addressing, displacement addressing
- Extract offset value of “tainted” operand

Byte sequencing

- Identifying one byte character in the output
- Sort in ascending order of instruction addresses that was extracted
- Check offset value to concatenate byte strings
- Represent a sequence of bytes
- Save string to dictionary file

Fuzzing with LightBranch



Thank You

For your attention

