CS 250 Software Security

Automatic Exploit Generation

An Example of Stack Overflow

```
int main(int argc, char **argv) {
  int skfd;               /* generic raw socket desc.    */
  if(argc == 2)
    print_info(skfd, argv[1], NULL, 0);
...
static int print_info(int skfd, char *ifname, char *args[], int count)
    {
  struct wireless_info  info;
  int                   rc;
  rc = get_info(skfd, ifname, &info);
...
static int get_info(int skfd, char *ifname, struct wireless_info * info
    ) {
  struct iwreq          wrq;
  if(iw_get_ext(skfd, ifname, SIOCGIWNAME, &wrq) < 0) {
      struct ifreq ifr;
      strcpy(ifr.ifr_name, ifname); /* buffer overflow */
...
```

Figure 1: Code snippet from Wireless Tools' `iwconfig`.

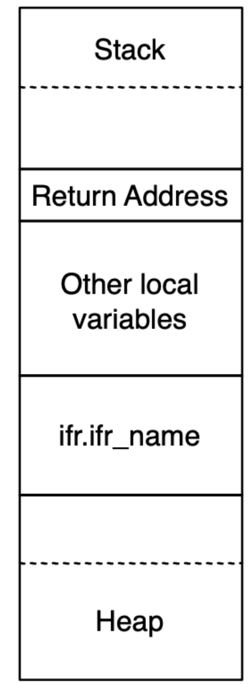


Figure 2: Memory Diagram

Constructing a Simple Exploit for Stack-based Overflow

- Try to find an execution trace that would allow you to control the program counter
  - Perform symbolic execution, and check if PC is symbolic
- In the exploitable state (right before jump to the symbolic PC), find a location to inject your shellcode
  - Search the virtual memory for a sequence of continuous symbolic bytes that is large enough to fit the shellcode
- Set the symbolic PC to the location of shellcode
- Query the solver for the following constraints:
  - For i from 0 to shellcode_size: shellcode_location[i] = shellcode[i]
  - Symbolic_PC = shellcode_location
  - Path Predicate

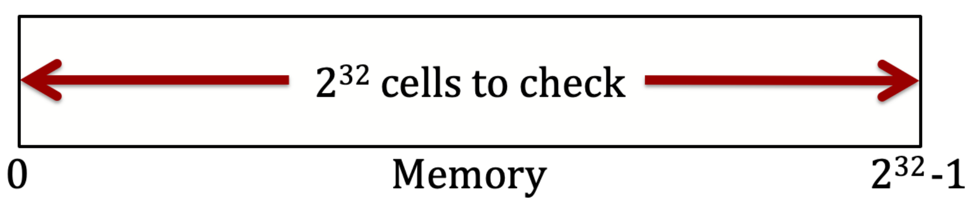
Challenges

- Symbolic execution is slow
  - We have talked about fast concolic execution
- Transformation over input
  - Especially table lookup (e.g., isspace, isalpha, toupper, tolower, mbtowc)

```
x = user_input();
y = mem[x];
assert (y == 42);
```

x can be anything

Which memory cell contains 42?



Symbolic Memory Index is Hard to Handle

- Method 1: Concretization

$\Pi \wedge mem[x] = 42 \wedge \Pi'$ → $\Pi \wedge x = 17 \wedge mem[x] = 42 \wedge \Pi'$

✓ Solvable

✗ Exploits

Symbolic Memory Index is Hard to Handle

- Method 2: Fully Symbolic

$$\Pi \land mem[x] = 42 \land \Pi'$$

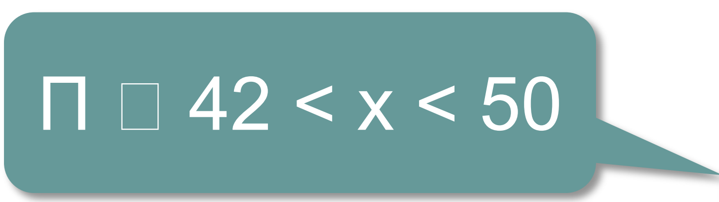
$$\Pi \land mem[x] = 42$$
$$\land\ mem[0] = v_0 \land \cdots \land mem[2^{32}\text{-}1] = v_{2^{32}\text{-}1}$$
$$\land\ \Pi'$$

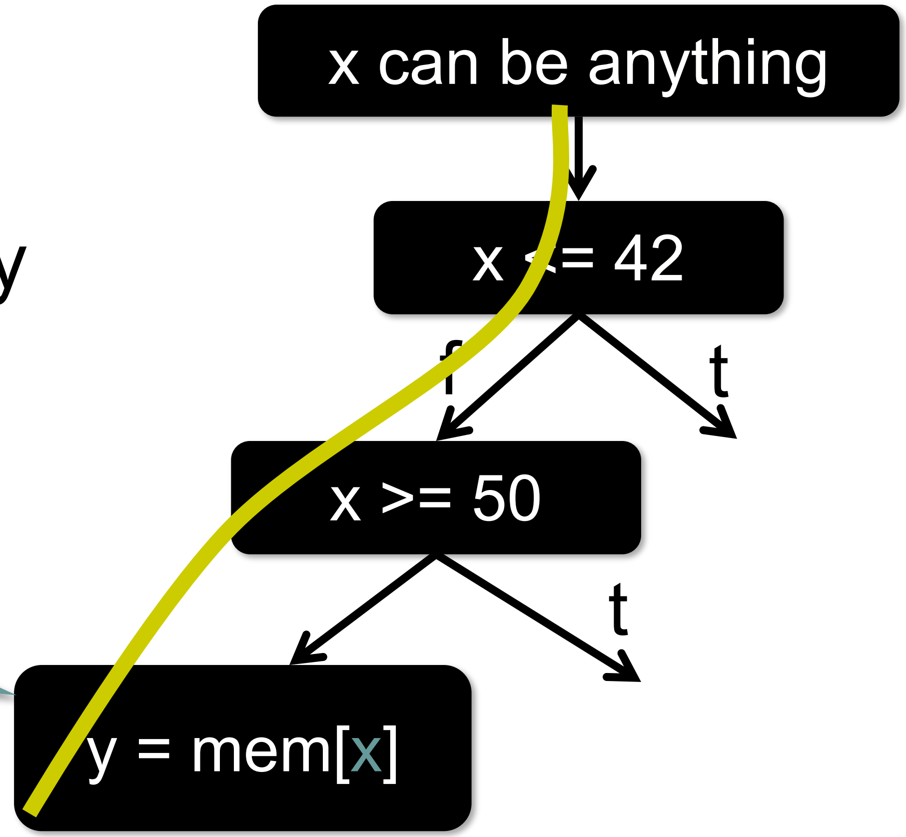
✗ Solvable

✓ Exploits

Mayhem’s Solution

Path predicate (Π) constrains ***range*** of symbolic memory accesses





Use symbolic execution state to:

Step 1: Bound memory addresses referenced

Step 2: Make search tree for memory address values

Step 1 — Find Bounds

mem[x & 0xff]

Lowerbound = 0, Upperbound = 0xff

1. Value Set Analysis[1] provides initial bounds
    - Over-approximation
2. Query solver to refine bounds

[1] Balakrishnan *et al.*, Analyzing memory accesses in x86 executables, ICCC 2004

Step 2 — Index Search Tree Construction

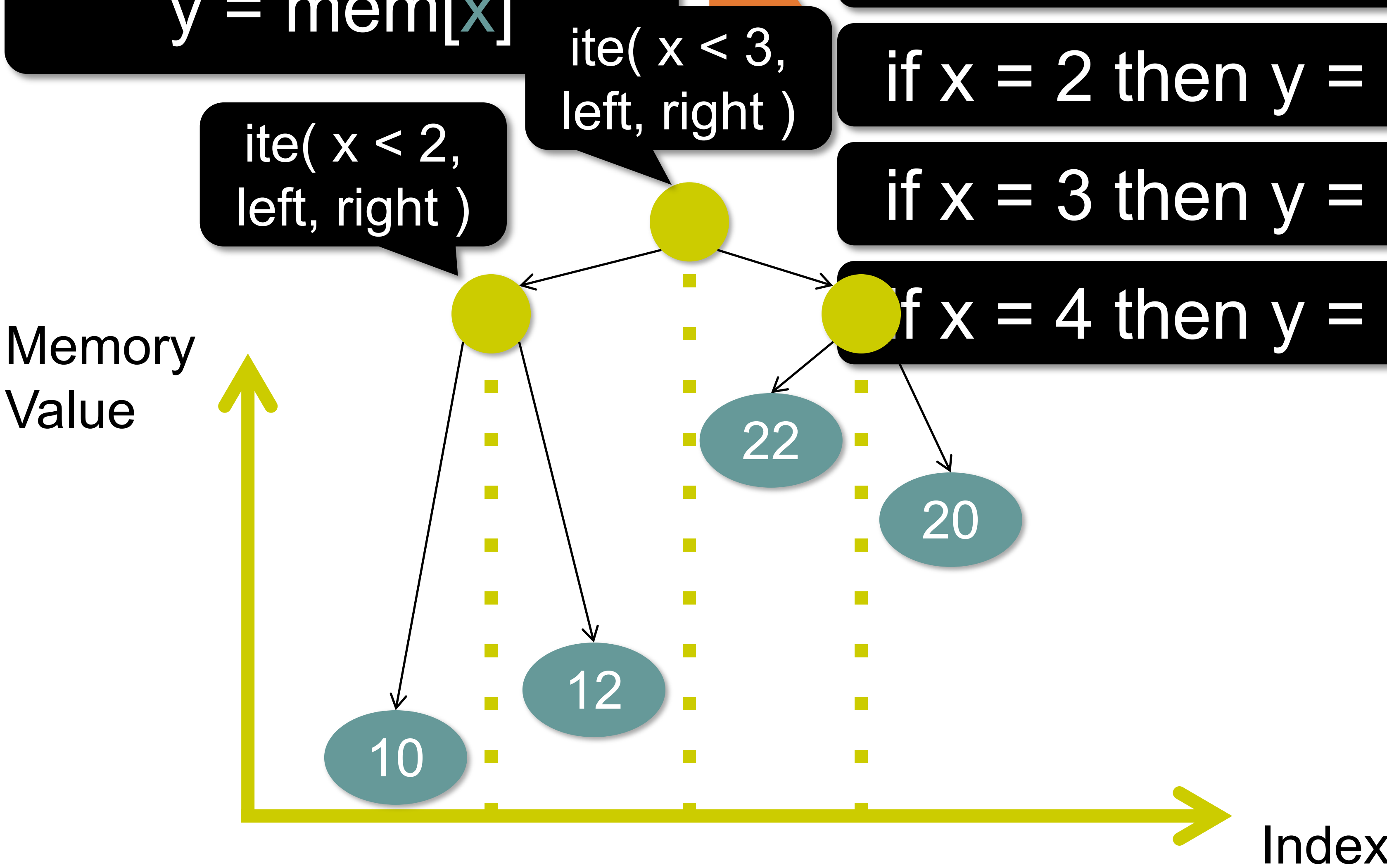
y = mem[x]

if x = 1 then y = 10

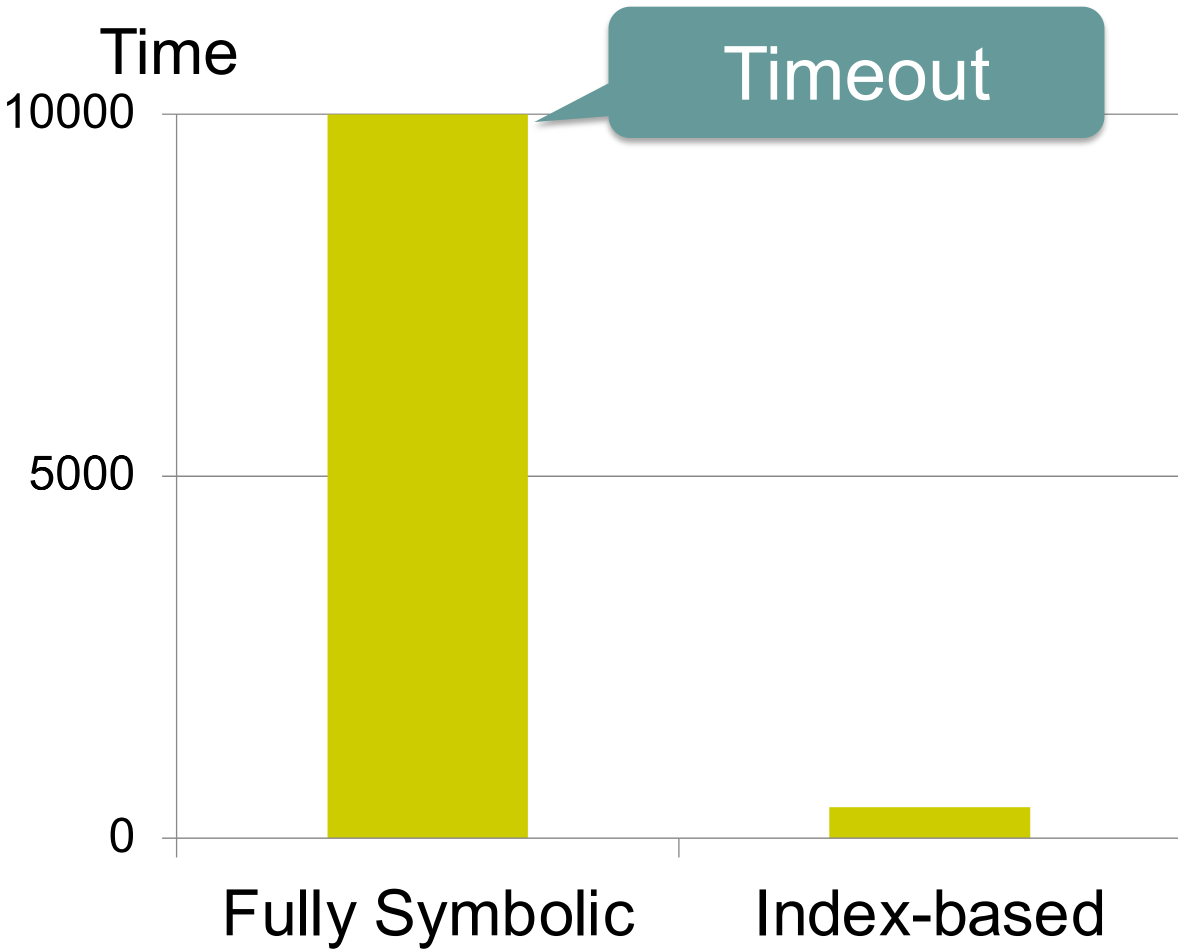
if x = 2 then y = 12

if x = 3 then y = 22

if x = 4 then y = 20

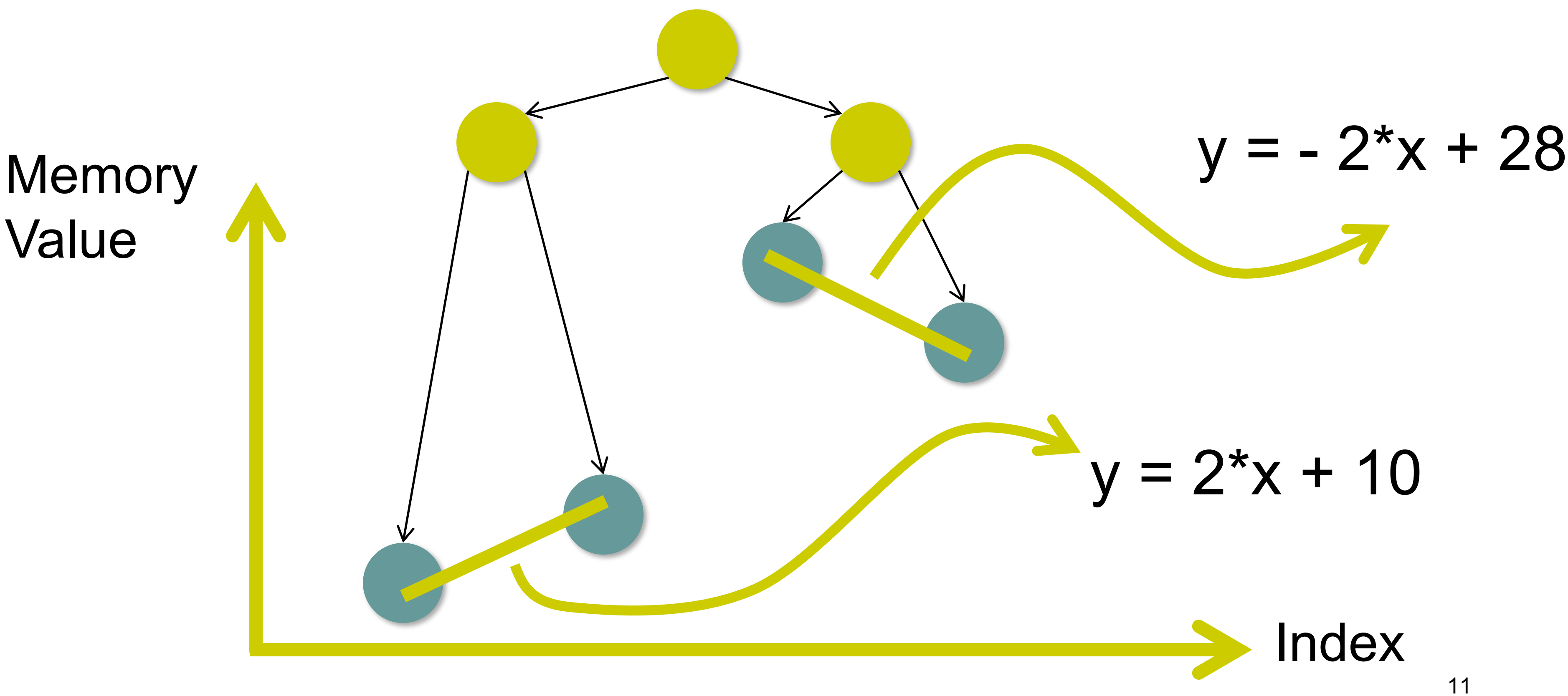


Fully Symbolic *vs.* Index-based Memory Modeling

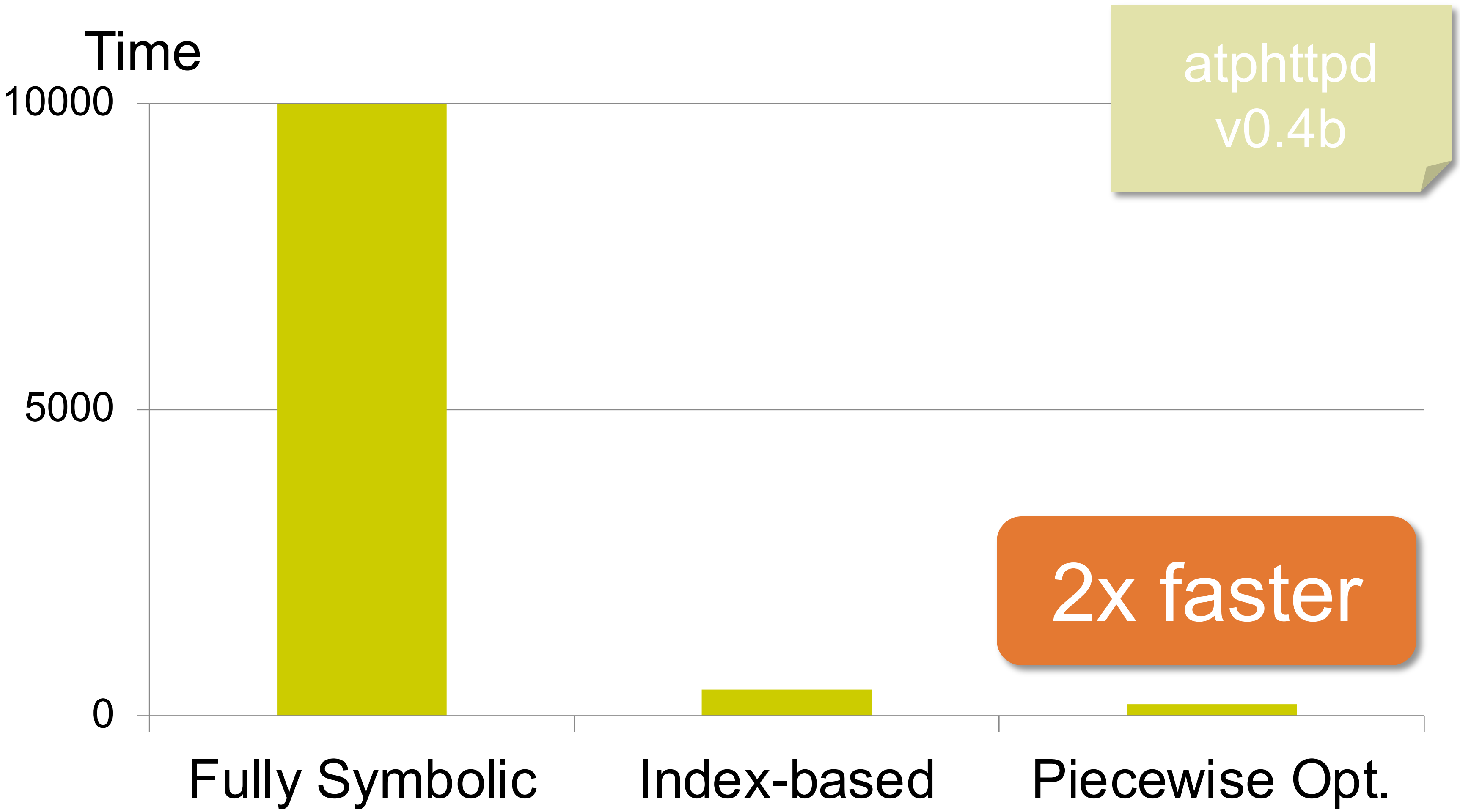


atphttpd v0.4b

Index Search Tree Optimization: *Piecewise Linear Approximation*



Piecewise Linear Approximation



More Challenges

- Path predicate might be overly constrained
  - This path is not feasible, but a slightly different path is
  - Memory index concretization
  - Unsound concolic execution (concretize on complex cases)
- What about fuzzing?

Such exploits are too simple/unrealistic!

- Easily defeated by existing defense
  - DEP: Data Execution Prevention
  - ASLR: Address Space Layout Randomization
- Bypass DEP: ROP (Return-Oriented Programming)
  - https://github.com/mantvydasb/RedTeaming-Tactics-and-Techniques/blob/master/offensive-security/code-injection-process-injection/binary-exploitation/rop-chaining-return-oriented-programming.md
- Bypass ASLR: Leverage information leakage
  - Some register or memory at a relative position might already contain a useful address
  - mov [reg], esp, add [reg], esp
- How to fully automate it?
  - Angrop: https://github.com/angr/angrop

How to defend against control-flow hijacking exploits?

- Program Hardening (will be discussed later)
  - Control-Flow Integrity
  - Shadow Stack
  - Control Pointer Integrity

- How to bypass these protections?
  - Data-oriented exploits.

Automatic Generation of Data-Oriented Exploits

USENIX Security 2015

```c
1 int server() {
2   char *userInput, *reqFile;
3   char *privKey, *result, output[BUFSIZE];
4   char fullPath[BUFSIZE] = "/path/to/root/";
5
6   privKey = loadPrivKey("/path/to/privKey");
7   /* HTTPS connection using privKey */
8   GetConnection(privKey, ...);
9   userInput = read_socket();
10  if (checkInput(userInput)) {
11    /* user input OK, parse request */
12    reqFile = getFileName(userInput);
13    /* stack buffer overflow */
14    strcat(fullPath, reqFile);
15    result = retrieve(fullPath);
16    sprintf(output,"%s:%s",reqFile,result);
17    sendOut(output);
18  }
19 }
```

Code 1: Vulnerable code snippet. String concatenation on line 14 introduces a stack buffer overflow vulnerability.

```
struct user_details { uid_t uid; ... } ud;
...                    //run with root uid
ud.uid = getuid();     //in get_user_info()
...
vfprintf(...);         //in sudo_debug()
...
setuid(ud.uid);        //in sudo_askpass()
...
```

Code 3: Code snippet of sudo, setting uid to normal user id.

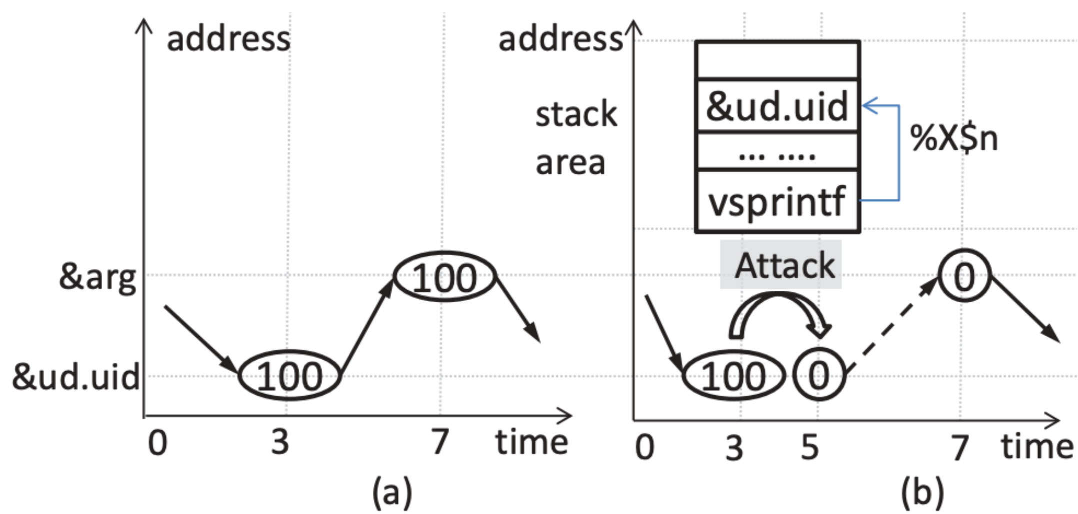


Figure 7: Stitch by complete memory address reuse of sudo. The dashed line is the new edge (single-edge stitch). An address of *ud.uid* exists on ancestor's stack frame, which is reused to overwrite *ud.uid*.

A successful data-oriented exploit requires

- The exploit input satisfies the program path constraints to reach the memory error, create new edges and continue the execution to reach the target

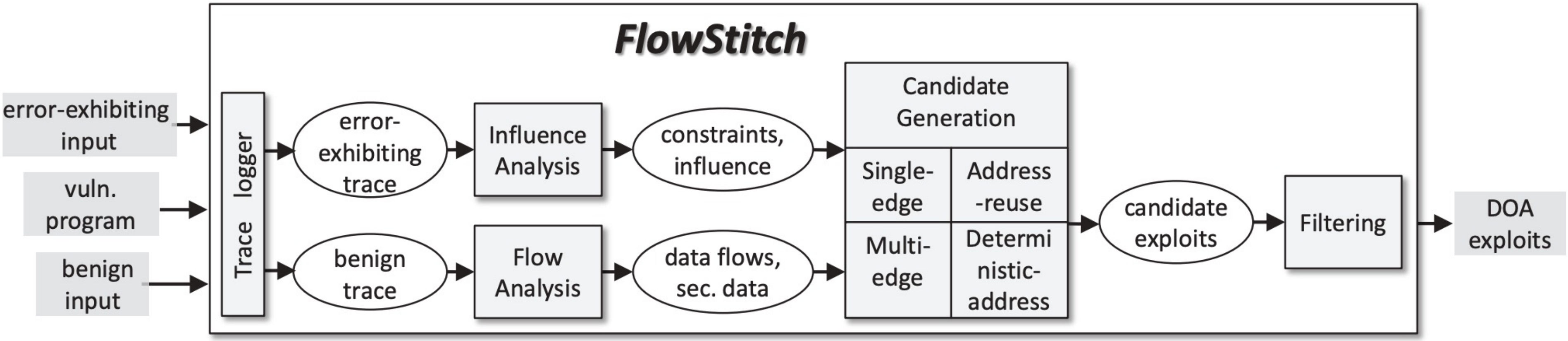
- The instructions executed in the exploit must conform to the program's static control flow graph

- A data flow stitching problem

Challenges

- Large search space for stitching
  - Many possible target variables
- Limited knowledge of memory layout.
  - How to bypass ASLR?
- Complex program path constraints
  - The exploit must satisfy all path constraints
  - Avoid invalid memory accesses

How it works


FlowStitch
error-exhibiting input
vuln. program
benign input
Trace logger
error-exhibiting trace
benign trace
Influence Analysis
Flow Analysis
constraints, influence
data flows, sec. data
Candidate Generation
Single-edge
Address -reuse
Multi-edge
Determi nistic-address
candidate exploits
Filtering
DOA exploits

More Details

- Memory Error Influence Analysis
  - Use Symbolic Execution
- Security-Sensitive Data Identification
  - Specific syscalls/libc calls: printf, send, setuid, etc.
  - Program secret, permission flags
- Stitching Candidates
  - Path conditions reach memory error instructions
  - Path conditions continue to the target flow
  - Integrity of the control data

More Thinking

- Exploit Generation vs. Vulnerability Discovery
  - Both are search problems
  - Exploit generation relies more on symbolic execution, but fuzzing is useful too
  - Vulnerability discovery uses both fuzzing and symbolic execution
  - Exploit generation is more directed
  - Vulnerability discovery can be directed or not.