Lec 18: Instrumentation

CS492E: Introduction to Software Security

Sang Kil Cha

How to Monitor Program Execution?

- Attaching debugger to a running process (e.g., `ptrace`)
  - GDB, LLDB, WinDbg, etc.
  - Single stepping: context switching for every single execution
- Instrumentation
  - Pin, DynamoRio, Valgrind, etc.

Instrumentation?

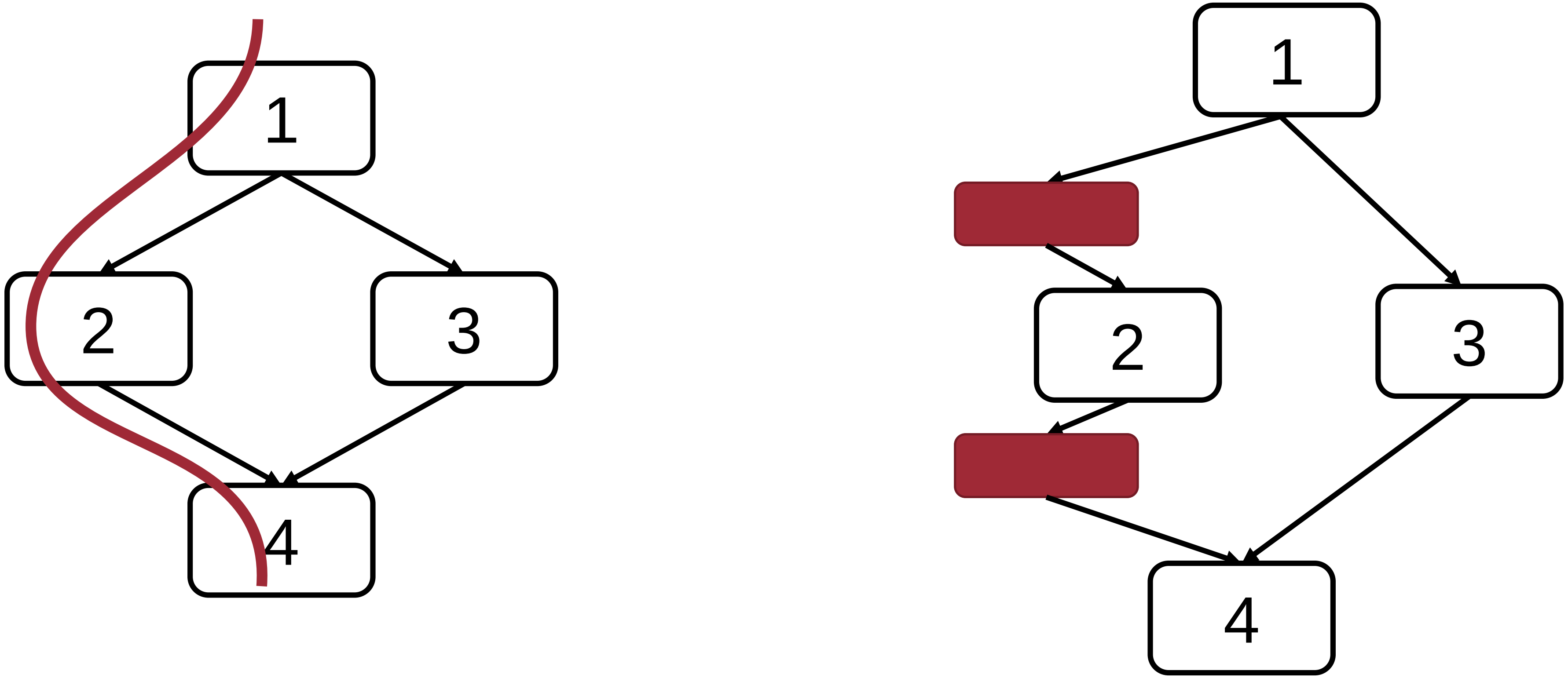
```
void somefn()
{
  char array[42];

  for (int i = 0; i < 42; i++ ) {

    array[i] = i;
  }
}
```

Instrumentation?

```
void somefn()
{
  char array[42];
  printf("before loop\n");
  for (int i = 0; i < 42; i++ ) {
    printf("inner loop\n");
    array[i] = i;
  }
}
```

	Source-based	Binary-based
Dynamic		Pin (***PLDI 2005***) DynamoRio (***CGO 2003***) Valgrind (***PLDI 2007***)
Static	CIL (***CC 2002***) LLVM (***CGO 2004***)	PEBIL (***ISPASS 2010***) DynInst (***HPCA 2000***) Diablo (***ISSPIT 2005***)

Dynamic Instrumentation



Code → JIT-compiled Code

Dynamic vs. Static Instrumentation

- Dynamic
  - High overhead
  - Easy to instrument external libraries
  - Handles dynamically generated code

- Static
  - Fast
  - Difficult to instrument external libraries (need to be separately instrumented)
  - Cannot handle dynamically generated code

Valgrind

- Developed in 2003 by Nicholas Nethercote
  - Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,
    PLDI 2007
  - How to Shadow Every Byte of Memory Used by a Program,
    VEE 2007

- Memcheck tool detects memory errors (only for dynamically allocated memory objects)

Shadow Memory

- Shadow memory stores metadata for each memory cell

- Memcheck uses shadow memory
  - ***A* bits**: every memory byte is shadowed with a single *A* bit, which indicates if the memory byte is accessible or not (e.g., freed memory)

  - ***V* bits**: every register and memory byte is shadowed with eight *V* bits, which indicate if the value bits are initialized.

Shadow Memory

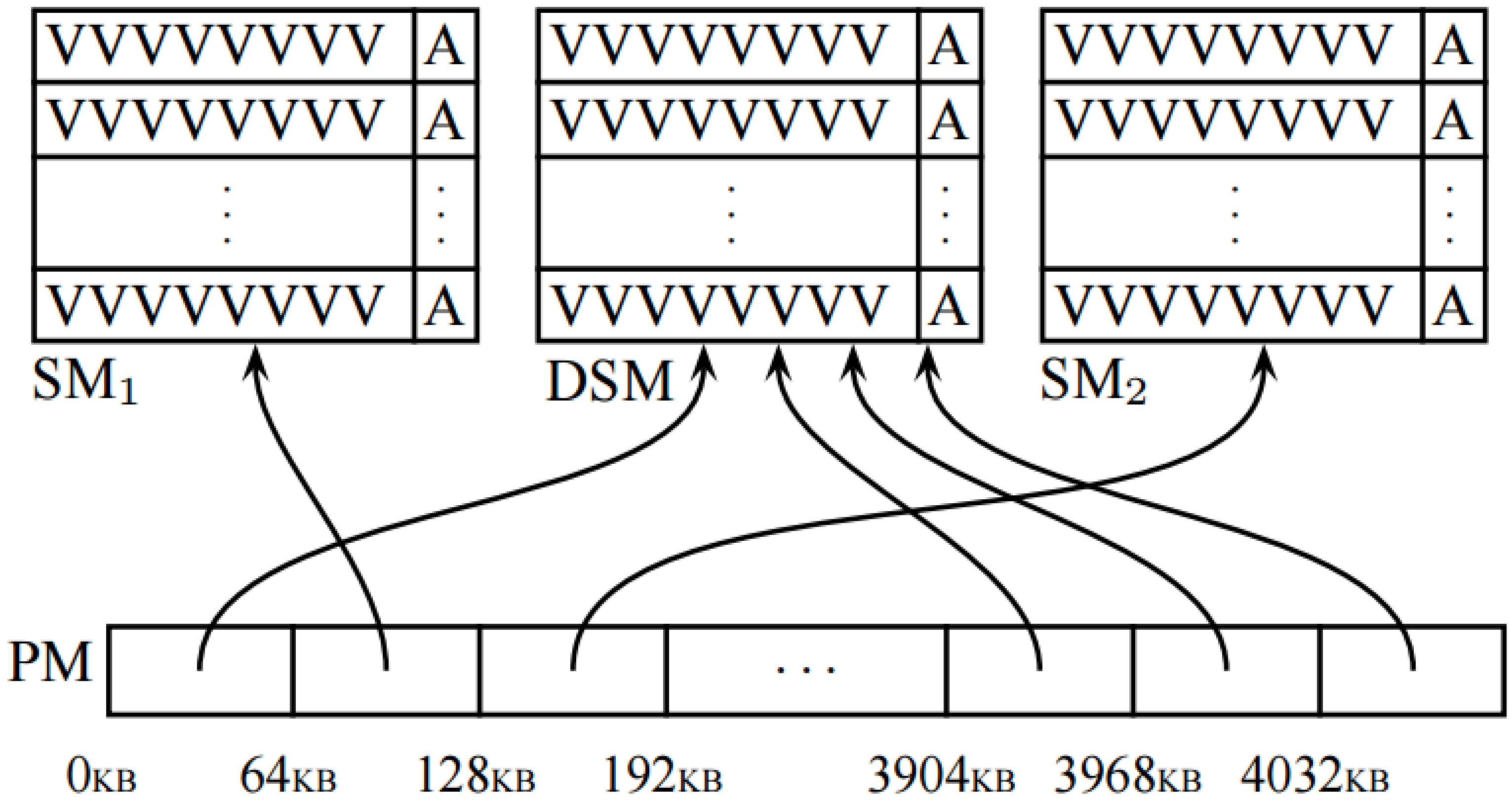


Image taken from How to Shadow Every Byte of Memory Used by a Program, VEE 2007

Address Sanitizer (Asan)

- Static instrumentation version of Memcheck

- AddressSanitizer: A Fast Address Sanity Checker, ***USENIX ATC 2012***

Compact Shadow Memory

- Memcheck: byte-to-byte mapping
- Asan: 8-byte-to-byte mapping
- Key idea: heap memory is always 8-byte aligned

9 States for 8-Byte Aligned Memory

Mapping from Real to Shadow Memory

- Memcheck: address translation table

- Asan: no table lookup
  - Reserve $1/2^3$ memory space
  - Shadow = (Addr >> 3) + Offset

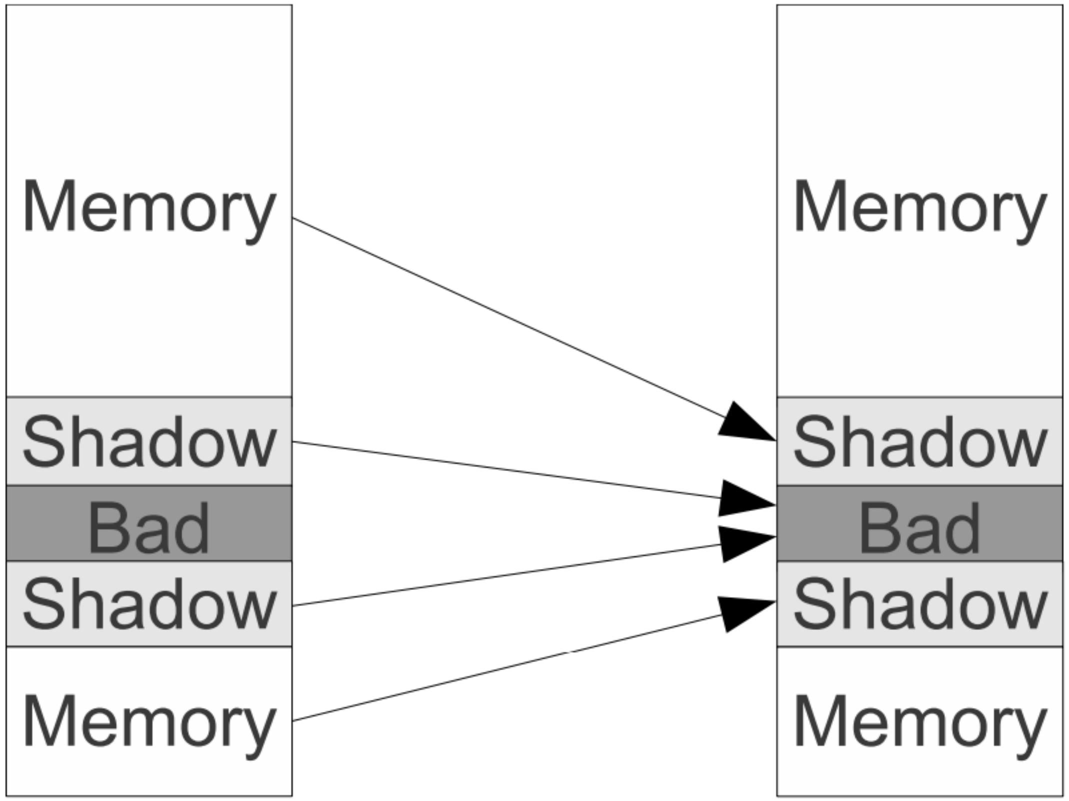


Image taken from AddressSanitizer: A Fast Address Sanity Checker, ATC 2012

Instrumentation: 8-byte Access

```
// Instrumentation begins
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0) ReportAndCrash(Addr);
// Instrumentation ends
```

```
*Addr = 42; // Original instruction
```

Instrumentation: 1-, 2-, or 4-byte Access

```
// Instrumentation begins
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k))
    ReportAndCrash(Addr);
// Instrumentation ends
```

```
*Addr = 42; // Original instruction
            // Accessing (AccessSize) bytes
```

SOFTWARE SECURITY LAB KAIST

Instrumenting Stack

```
void foo() {

  char arr[10];

  <function body>
```

Instrumenting Stack

```
void foo() {
  char rz1[32];
  char arr[10];
  char rz2[32-10+32];
  unsigned *shadow = (unsigned*)(((long)rz1>>3)+Offset);
  // poison the redzones around arr.
  shadow[0] = 0xffffffff; // rz1
  shadow[1] = 0xffff0200; // arr and rz2
  shadow[2] = 0xffffffff; // rz2
  <function body>
  // un-poison all.
  shadow[0] = shadow[1] = shadow[2] = 0;
```

32-byte aligned redzones around the stack object

Memory Alloc/Dealloc

- Insert redzones around allocated memory
- Freed page is set to be "red"
- Similar to sparse page mapping
  (We will discuss this later again)

AddressSanitizer has False Negatives

```
int *a = new int[2]; // 8-byte aligned
int *u = (int*)((char*)a + 6);
*u = 1; // Access to range [6-9]
```

Anti Debugging

Anti- Debugging/Instrumentation

- Benign use: software copy protection

- Malicious use: malware

Software Copy Protection

How would you protect your software?

Example of Copy Protection

Ask a question that only a valid user can answer:

- What is the xth word in page y of the manual?
- What is your serial number that is given at the time you purchased?

Example of Copy Protection (cont'd)

Check if a program is running on a registered device

- IMEI of a smartphone
- IP address, Mac address, user ID, etc.

Example of Copy Protection (cont'd)

A phone-based activation

- Only a registered phone number can be used
- You will not share your license (or serial) with many people

Altering Software?

You can easily bypass all such protections by simply modifying the program executables.

Typically one-byte change in binary

```
// ...

if ( phone_activation() == SUCCESS )
  return VALID_USER;

// ...
```

Software Cracking

- Remove or disable features
  - Copy protection routines
  - Advertisement

- Reversing is crucial: no source code for COTS software

This is Illegal!

PTRACE Recap

Debugee process

```
ptrace(PTRACE_TRACEME, 0, 0, 0);
execve("/bin/ls", args /* arguments */, 0);
```

PTRACE Recap

Debuger process

```
int status;
waitpid(pid, &status, 0);
while (WIFSTOPPED(status)) {
  ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
  // Do something
  waitpid(pid, &status, 0);
}
```

Breakpoints?

- Software breakpoints
  - int3 instruction (0xcc) replacement
  - Unlimited

- Hardware breakpoints
  - DR registers on x86
  - Limited to 4 (on x86)

Software Breakpoint

```
4004d6: 55                push   rbp
4004d7: 48 89 e5          mov    rbp,rsp
4004da: b8 00 00 00 00    mov    eax,0x0
4004df: 5d                pop    rbp
4004e0: c3                ret
```

BP

Software Breakpoint

```
4004d6: 55                push   rbp
4004d7: 48 89 e5          mov    rbp,rsp
4004da: cc                int3
4004db: 00 00             add    BYTE PTR [rax],al
4004dd: 00 00             add    BYTE PTR [rax],al
4004df: 5d                pop    rbp
4004e0: c3                ret
```

1. SIGTRAP at 4004da
2. Replace the byte at 4004da with the original byte (b8)
3. Modify the program counter (EIP/RIP)
4. Resume

Anti-Debugging (1)

```
if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) {
    return 1;
}
```

Anti-Debugging (2)

`/proc/$PPID/status`

Check the parent’s name!

Anti-Debugging (3)

```
signal(SIGTRAP, handler); // Implicit control flow
```

Anti-Debugging (4)

```
memchr(code, 0xcc, size);
```

Debugger without PTRACE?

- Emulator-based debugging

- Instrumentation-based

Red Pill and Blue Pill

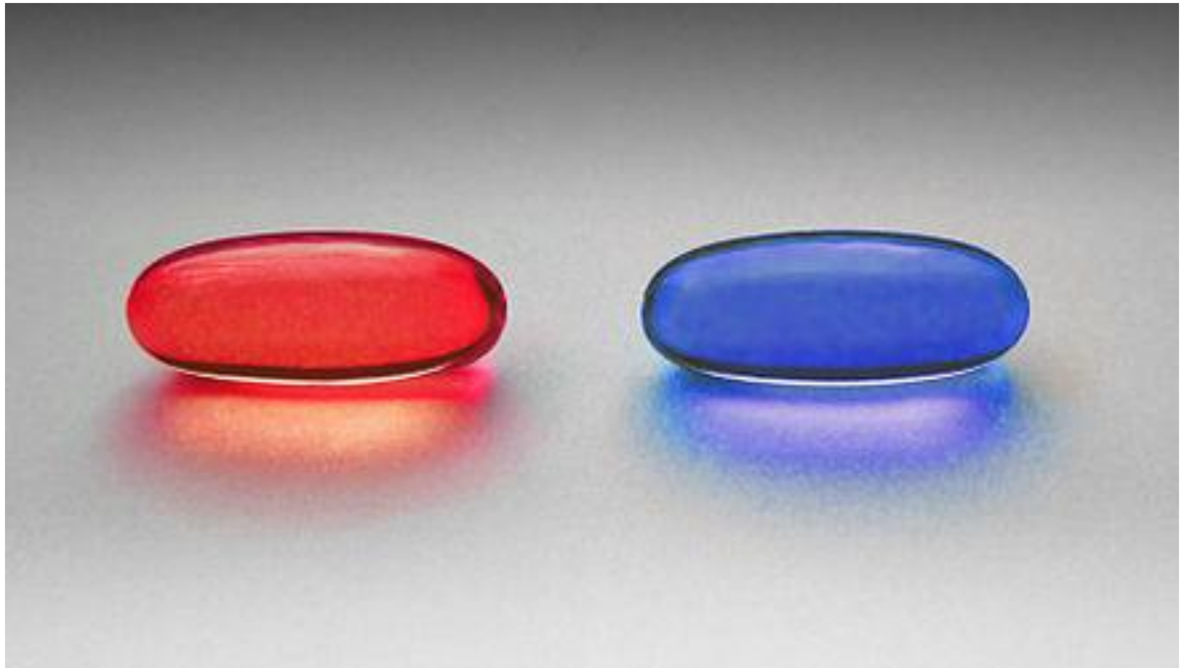


Image from https://en.wikipedia.org/wiki/Red_pill_and_blue_pill

Red Pill = Detect Virtualization

- /proc/ide/hd*/model
- dmidecode
- Timing channel
- Etc.

Static Instrumentation

Binary Rewriting
= Static Binary Instrumentation

Given a binary, statically instrument it in such a way that the rewritten binary will run as it is.

Why Binary Rewriting is Difficult?

```
// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: sub rsp, 0x50
…

// func2:
0x1200: push rbp
0x1203: mov rbp, rsp
…
```

What happens when we add instrumentation code here?

Fixing Cross-References is Difficult

- Identifying **dynamically computed references** is difficult
- Correctly identifying **jump tables** in is difficult
- Correctly **recovering CFG** is difficult

Compiler-Assisted Rewriters

- Assuming the existence of source code

- Or **debugging symbols**
  (like a cheat key for binary analysis)

- Tools: ATOM, Vulcan, Diablo, Pebil, etc.

Debugging Symbols?

- You can use the "-g" option to produce a binary with full symbolic information.
  - It is nearly equivalent to having the source code

- Even if you do not use the "-g" option, there still remain partial information.

- When you run the "`strip`" command, then you can completely remove debugging symbols.

Patch-based Rewriters

Fix the layout of the binary. So there's no need to fix the references in the binary.

But how do you add instrumentation without changing the layout of the binary?

Fixing the Layout

```
// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: sub rsp, 0x50 => jmp detour
0x110b:
…

// func2:
0x1200: push rbp
0x1203: mov rbp, rsp
…
detour:
// instrumentation routine starts here.
sub rsp, 0x50
jmp 0x110b
```

This part is simply appended without touching the original layout

Many tools: **Detour**, DynInst, E9Patch, etc.

* Detours: Binary interception of win32 functions. USENIX 1999

Any Problem?

```
// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: sub rsp, 0x50 => jmp detour
0x110b:
…
```

What if the target instruction is smaller than the jump instruction?

```
…
detour:
// instrumentation routine starts here.
sub rsp, 0x50
jmp 0x110b
```

Table-based Rewriters

- Address the applicability of patch-based rewriting methods.

- Create a duplicate copy of a binary, and use an address-translation table at runtime.
  - The table maps an original address to a new address (of the copy)

Table-based Rewriters (cont'd)

```
// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: call rax; func2
…

// func2:
0x1200: push rbp
0x1203: mov rbp, rsp
…
```

```
// func1:
0x11100: push rbp
0x11103: mov rbp, rsp
; instrumentation code
...
0x11117: call table_lookup_rax
0x11119: call rax ; 0x11300
…

// func2:
0x11300: push rbp
0x11303: mov rbp, rsp
…
```

1200 -> 11300

What's the Problem?

- Time overhead
- Space overhead

Conclusion

- Instrumentation is crucial for monitoring program executions
- Dynamic instrumentation is slow, but can be used in several practical scenarios
- Anti-debugging technique tries to hinder dynamic analyses
- Static binary instrumentation is still an on-going research area

Questions?