HARDWARE-ASSISTED ROOTKITS:

ABUSING PERFORMANCE COUNTERS ON THE ARM AND X86 ARCHITECTURES

Matt Spisak
@matspisak
ENDGAME.

USENIX WOOT '16, Austin TX

OUTLINE

- Motivation
- Performance Monitoring Unit
- ARM PMU-Assisted Rootkit
- Intel x64 PMU-Assisted Rootkit
- Analysis

KERNEL PATCH PROTECTION

- Mitigations such as Kernel Patch Protection complicate rootkit development
- Examples of KPP:
  - Microsoft PatchGuard - x64 
  - Samsung TIMA-RKP - TrustZone based kernel Monitor 
  - Apple "WatchTower" - iOS 9+ 

PMU-ASSISTED SECURITY RESEARCH

- Prior art in (x86) PMU for debugging or defensive applications:
  - ROP detection with mispredicted branches
  - Control flow integrity (CFI) using Intel BTS and PMU
  - Rootkit detection (NumChecker) using perf counters
- **What about offensive applications for the PMU?**
- **And what about ARM?**

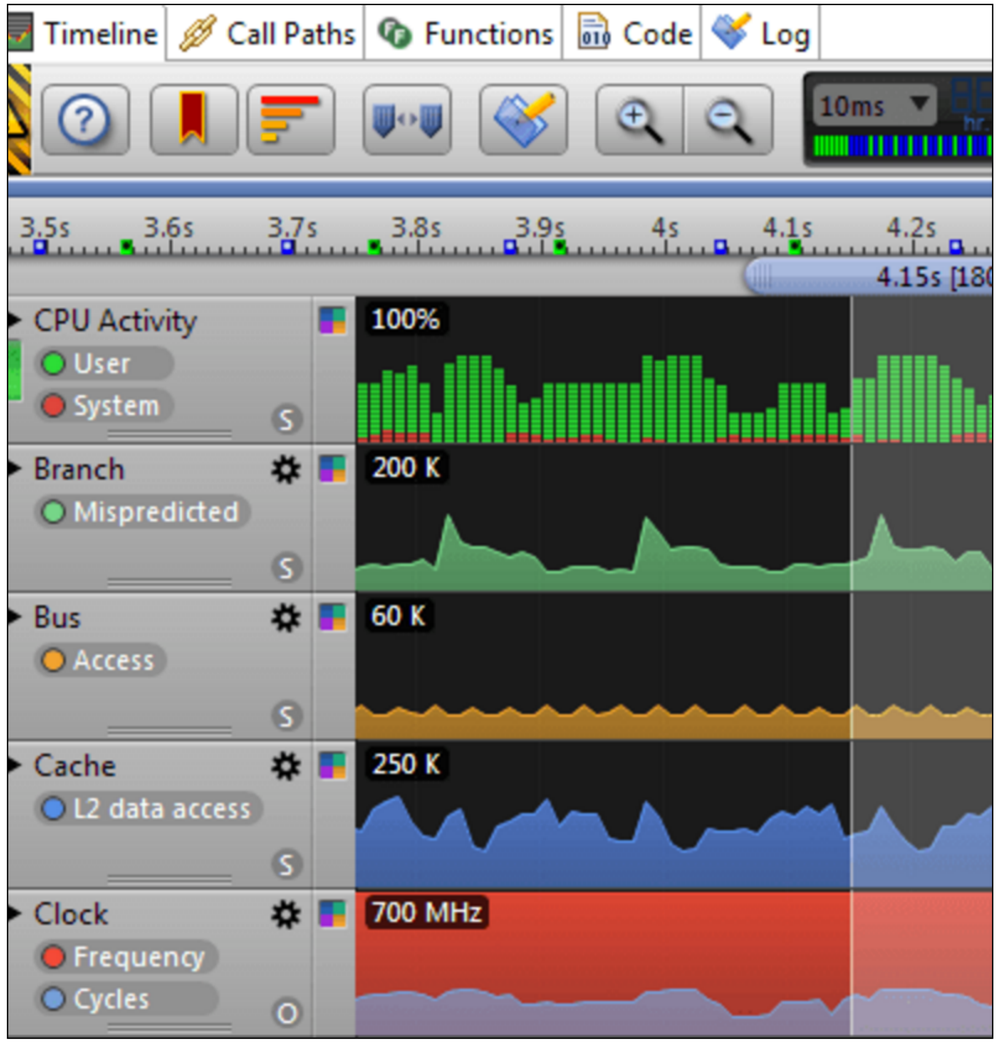
PERFORMANCE MONITORING UNIT

PMU BACKGROUND

- Found in most modern CPUs
- Typically consists of the following components:
  - 1 or more counters (PMCs) for counting events
  - Set of events that can be counted
  - Interrupt (PMI) to signal a counter overflow == sampling period

NORMAL USAGE OF PMU

- Provides real-time feedback on system
- Useful for software/hardware engineers
- Tools:
  - Intel VTune
  - ARM DS-5 Streamline
  - Linux perf / oprofile
  - Apple Xcode Instruments



ARM DS-5 Streamline

PMU COMPARISON

	ARM	intel
Introduced	**ARMv6**	**Original Pentium**
Interfaces	**CP15 system control coprocessor** memory-mapped (optional) external interface (optional)	**Model Specific Registers (MSRs)**
Interrupt Delivery	**IRQ**	**Non-maskable Interrupt (NMI)** **Interrupt Descriptor Table (IDT)**
Number of Counters	**1 cycle, up to 31 general purpose**	**3 fixed, 4+ general purpose**
Events	**Extensible**	**Fixed (Intel Manual)**
User mode Access	**Yes***	**No**

*** PMUSERENR.EN bit must first be set from PL1/EL1 or higher**

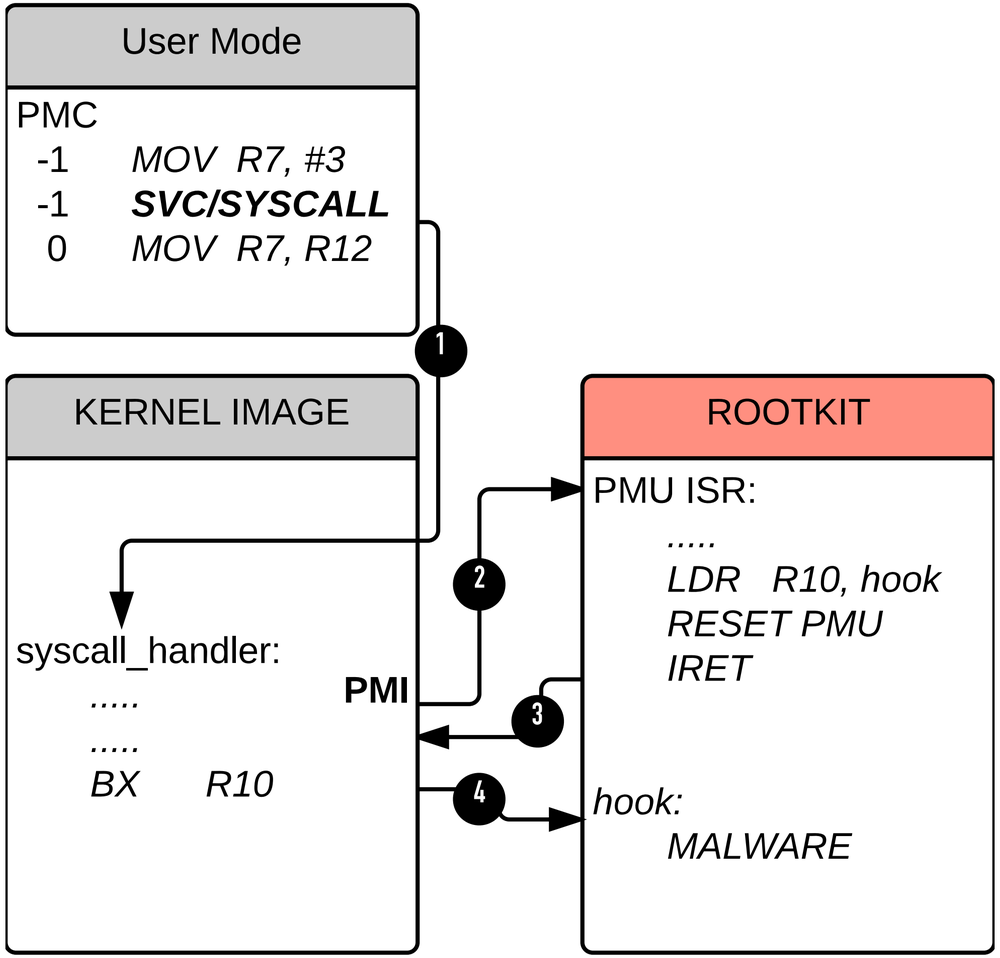
PMU WORKFLOW

PMC1: 0xFFFFFFFD (-3) Event: 0x08 (Instruction Retired)

	PMC	*INSTRUCTION*		
	-3	PUSH.W	{R4-R11, LR}	
	-2	SUB	SP, SP, #0x1C	
	-1	LDR.W	R8, [SP, #0x40]	
overflow	**0**	**LDR**	**R4, [R0, #0x18]**	**<< PMI**
	-2	MOV	R7, R0	
	-1	MOV	R6, R1	
overflow	**0**	**MOV**	**R0, R4**	**<< PMI**
	-2	MOV	R1, R8	
	-1	MOV	R5, R2	
overflow	**0**	**MOV**	**R9, R3**	**<< PMI**
	-2	BL	sub_xyz	

PMU-ASSISTED ROOTKIT APPROACH

- Identify candidate rootkit PMU events:
  - event is superset of all system calls
  - overhead of non-syscalls is low
- Trap all occurrences of "rootkit" event
- Attacker controlled ISR can optionally redirect execution



ARM ROOTKIT

INSPIRATION

Table C-1 PMU IMPLEMENTATION DEFINED event numbers (continued)

Event number	Event mnemonic	Description
0x7F-0x80	-	Reserved
0x81	EXC_UNDEF	Exception taken, Undefined Instruction
0x82	EXC_SVC	Exception taken, Supervisor Call
0x83	EXC_PABORT	Exception taken, Prefetch Abort
0x84	EXC_DABORT	Exception taken, Data Abort
0x85	-	Reserved
0x86	EXC_IRQ	Exception taken, IRQ
0x87	EXC_FIQ	Exception taken, FIQ
0x88	EXC_SMC	Exception taken, Secure Monitor Call
0x89	-	Reserved
0x8A	EXC_HVC	Exception taken, Hypervisor Call

ARM Architecture Manual ARMv7-A&R - Appendix C

COUNTING THE EXCEPTION VECTOR TABLE

	ARM Design				Custom ARM-based Design		
EVENT	*Cortex-A7*	*Cortex-A53*	*Cortex-A57*	*Cortex-A72*	*Scorpion*	*Krait*	*Kryo*
Undefined Instruction			√	√	√	√	?
SVC			√	√	√	√	?
Prefetch Abort			√	√	√	√	?
Data Abort			√	√	√	√	?
IRQ	√	√	√	√	√	√	?
FIQ	√	√	√	√	√	√	?
SMC	*	*	√	√	√	√	?
HVC			√	√	?	?	?

APPROACH

- Count and trap SVC instructions

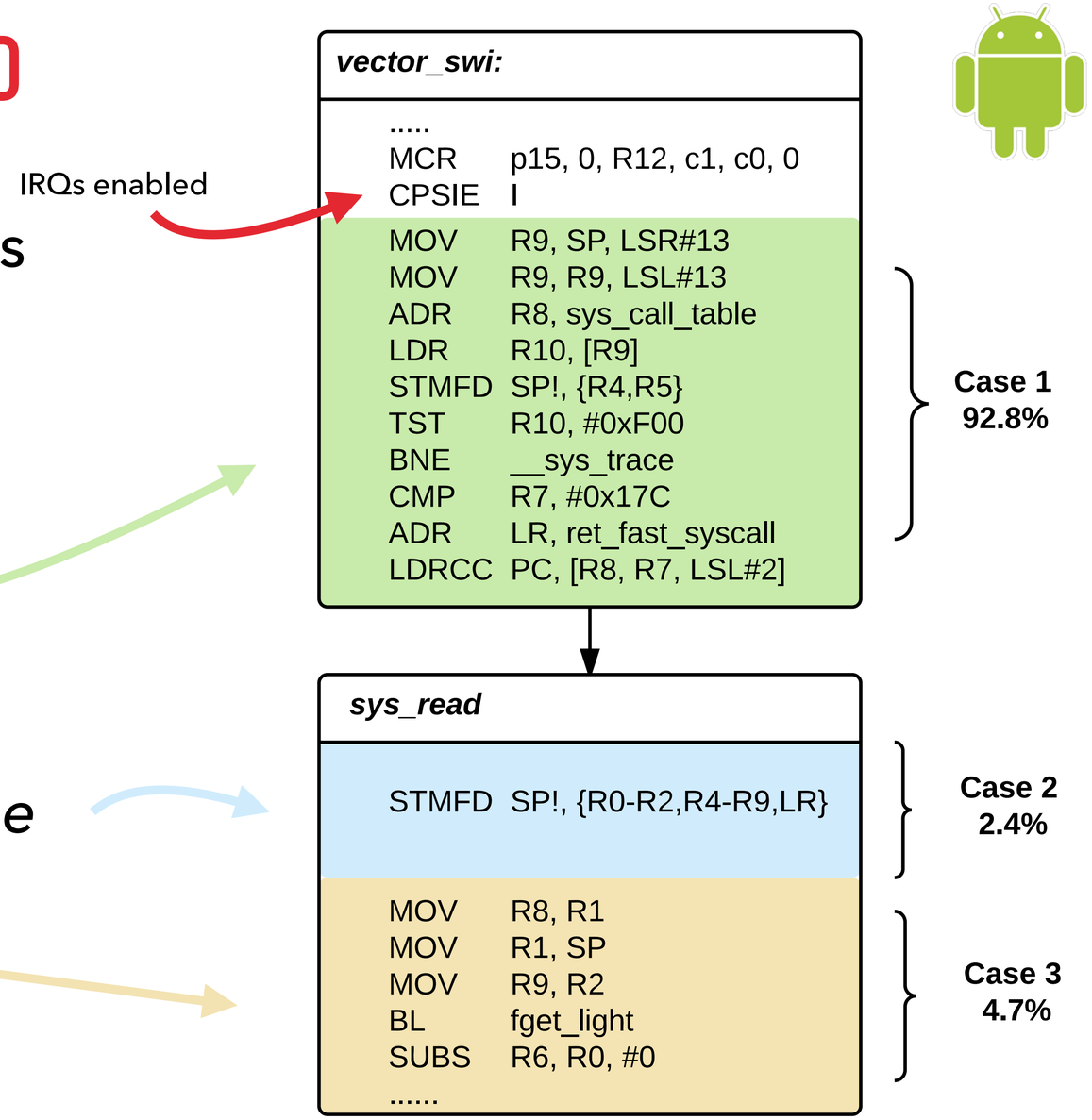
Motorola Nexus 6
Qualcomm APQ8084 (Krait) CPU
Android 5.0

CHALLENGES

- Finding the PMU Interrupt
  - Device tree source
  - Brute force: Register unused PPI/SPI's; trigger PMIs; diff `/proc/interrupts`
  - Registration: `request_percpu_irq()`, `request_threaded_irq()` on Android
- CPU Hot-Plugging
  - Linux/Android provides a callback: `register_hotcpu_notifier()`
- **Interrupt instruction skid**

CHALLENGE: DELAYED INSTRUCTION SKID

- PMI triggered at some point after IRQs enabled in *vector_swi*
- *3 cases we must deal with:*
  1. *PMI before branch to syscall routine within vector_swi*
  2. *PMI at entry point of syscall routine*
  3. *PMI in middle of syscall routine*



CASE 1: INTERRUPT BEFORE BRANCH TO SYSCALL ROUTINE

```
 define CPSIE_ADDR  0xC01064D0
…
    irq_regs = get_irq_regs();
    pregs = task_pt_regs(current);
    …
    if (pregs->ARM_r7 == 0x3  //sys_read
    {
       switch (irq_regs->ARM_pc - CPSIE_ADDR) //offset after CPSIE
       {
           //emulate remaining instructions up to LDRCC
           //can skip those involved in resolving syscall routine
           case 0x0
           case 0x4
              irq_regs->ARM_r9 = irq_regs->ARM_sp & 0xFFFFE000
            …
           case 0x14
           case 0x18
           case 0x1C
           case 0x20
               irq_regs->ARM_lr = ret_fast_syscall;
           case 0x24
               irq_regs->ARM_pc = (uint32_t)hook_sysread;
```

vector_swi:

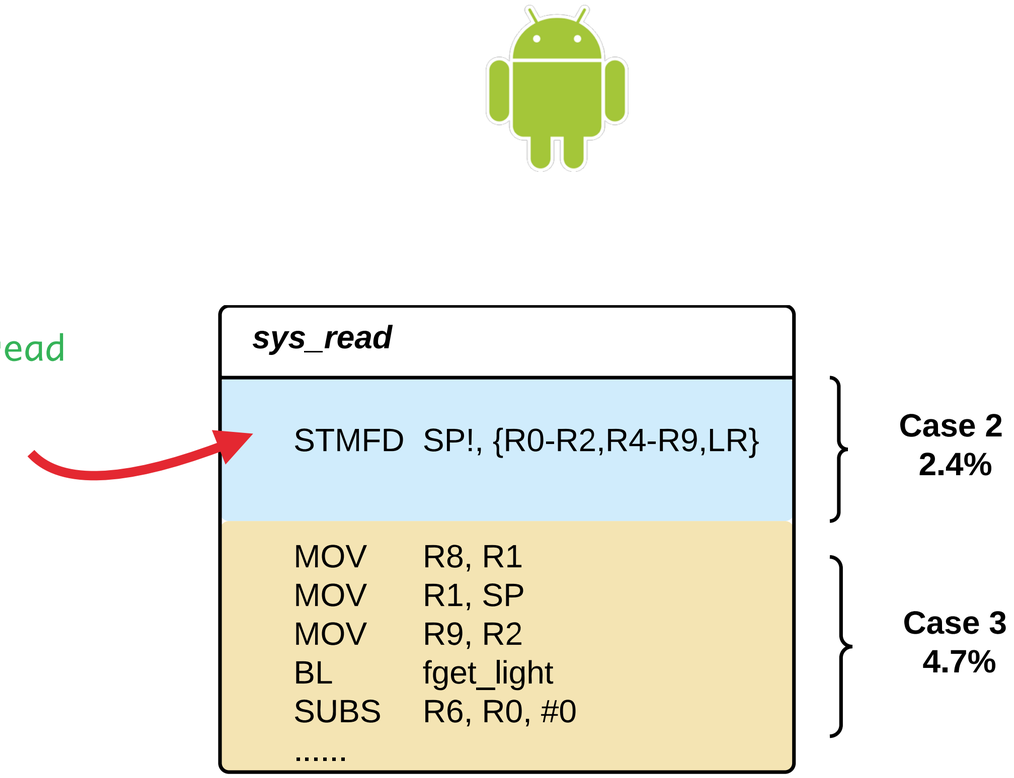
```
.....
MCR     p15, 0, R12, c1, c0, 0
CPSIE   I
MOV     R9, SP, LSR#13
MOV     R9, R9, LSL#13
ADR     R8, sys_call_table
LDR     R10, [R9]
STMFD   SP!, {R4,R5}
TST     R10, #0xF00
BNE     __sys_trace
CMP     R7, #0x17C
ADR     LR, ret_fast_syscall
LDRCC   PC, [R8, R7, LSL#2]
```

Case 1
92.8%

CASE 2: SYSCALL ROUTINE ENTRY POINT

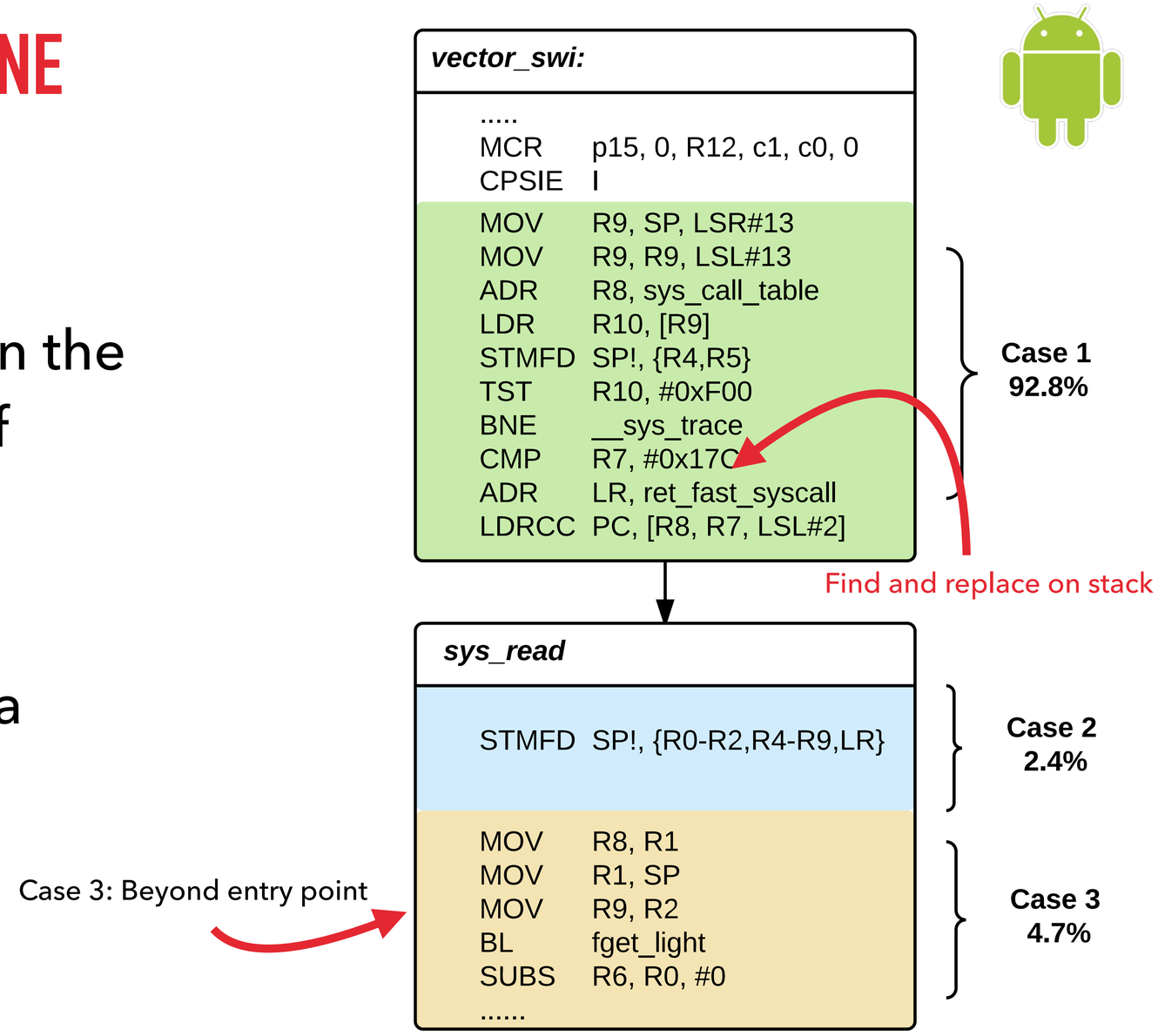
- Replace saved PC with address of hook

```
irq_regs = get_irq_regs();
pregs = task_pt_regs(current);
…
if (pregs->ARM_r7 == 0x3) //sys_read
{
  //Check if PMU interrupted at entry point addr of sys_read
  if (pregs->ARM_pc == orig_sys_read)
  {
      pregs->ARM_pc = (uint32_t)hook_sys_read;
```



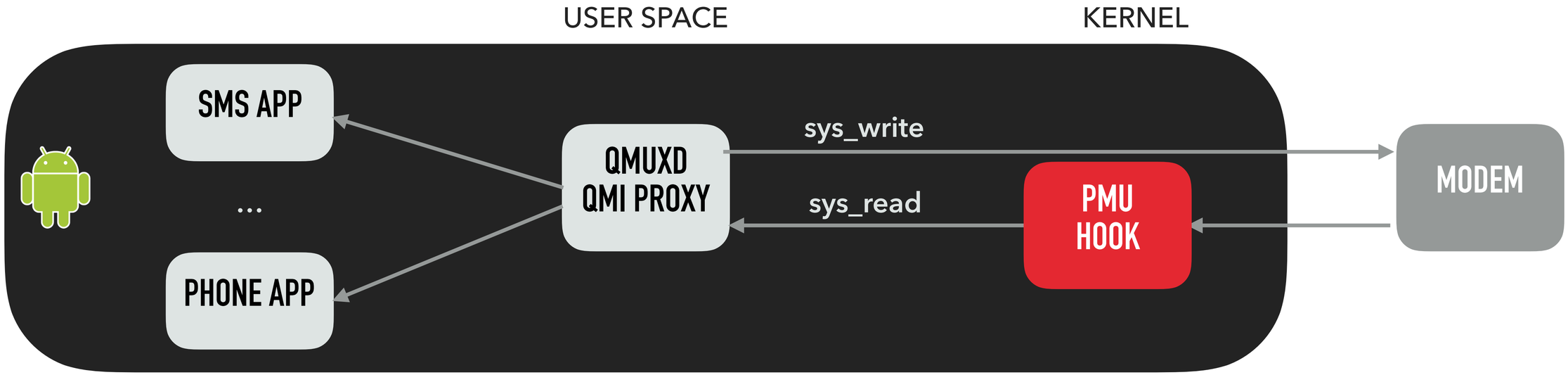
CASE 3: MIDDLE OF SYSCALL ROUTINE

- Let syscall routine complete
- Find address of ret_fast_syscall on the stack and replace with address of trampoline
- Trampoline loads LR with ret_fast_syscall, and branches to a post_hook function
- post_hook can retrieve original params, and modify as necessary



DEMO

- Hook sys_read in context of qmuxd in order to intercept all QMI comms from modem to Android (using only the PMU)



INTEL X64 ROOTKIT

FINDING A SYSCALL COUNTING EVENT

- No obvious PMU event like SVC for ARM
- SYSCALL instruction effectively takes a far branch to address in IA32_LSTAR (e.g. KiSystemCall64 on Windows)
- We can capture Far branches two ways with Intel PMU
  - BR_INST_RETIRED.FAR_BRANCH
  - ROB_MISC_EVENTS.LBR_INSERTED (requires LBR filtered to FAR only)
- Using this approach, we can now trap SYSCALLs on x64

PMU CONFIGURATION

- We can count Far Branches occurring in ring-0 only to reduce additional interrupts branches to user mode

PERFEVTSELx	LBR_FILTER	PMCx
Event: BR_INST_RETIRED Umask: FAR_BRANCH **MSR encoding: 0x5240C4**	N/A	-2
Event: ROB_MISC_EVENTS Umask: LBR_INSERTS **MSR encoding: 0x5320CC**	0xFE	-2

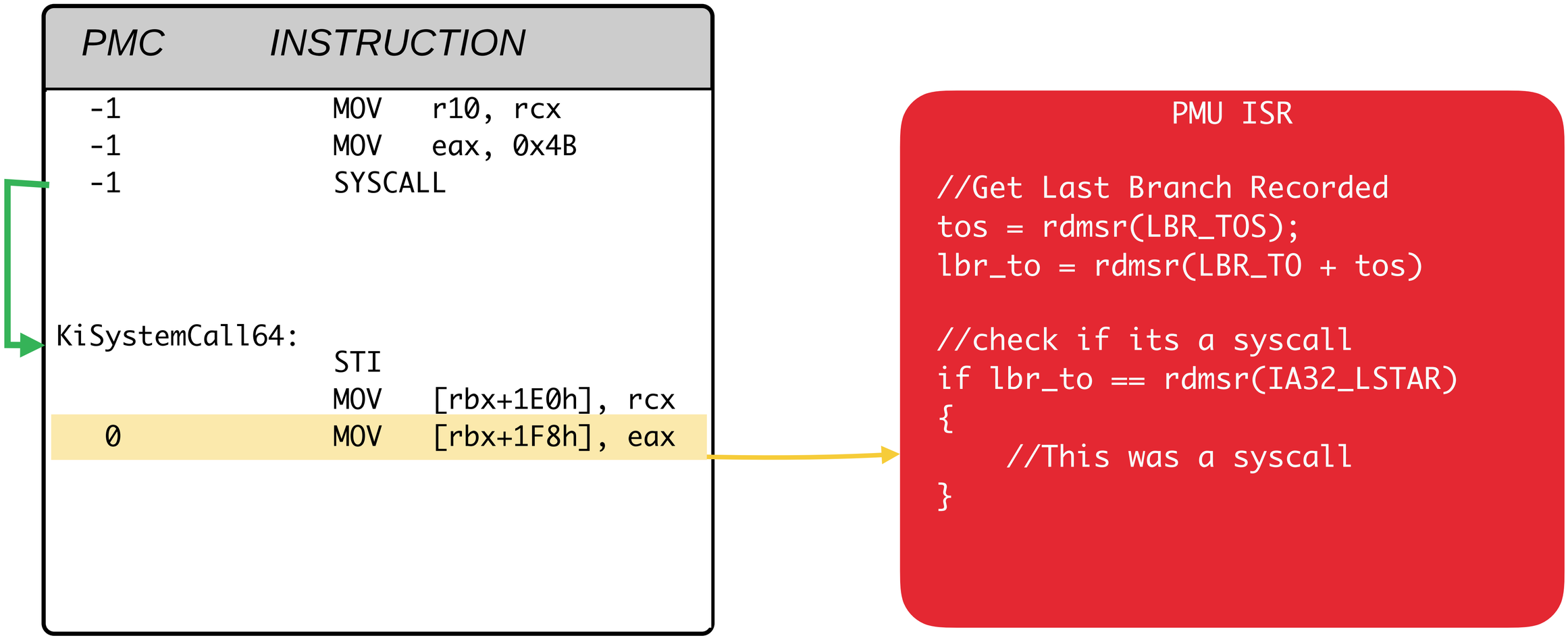
SAMPLING PERIOD

- Resetting counter value to -1 will result in an interrupt loop due to the iretq in ISR returning to interrupted kernel code
- Choose counter value of -2

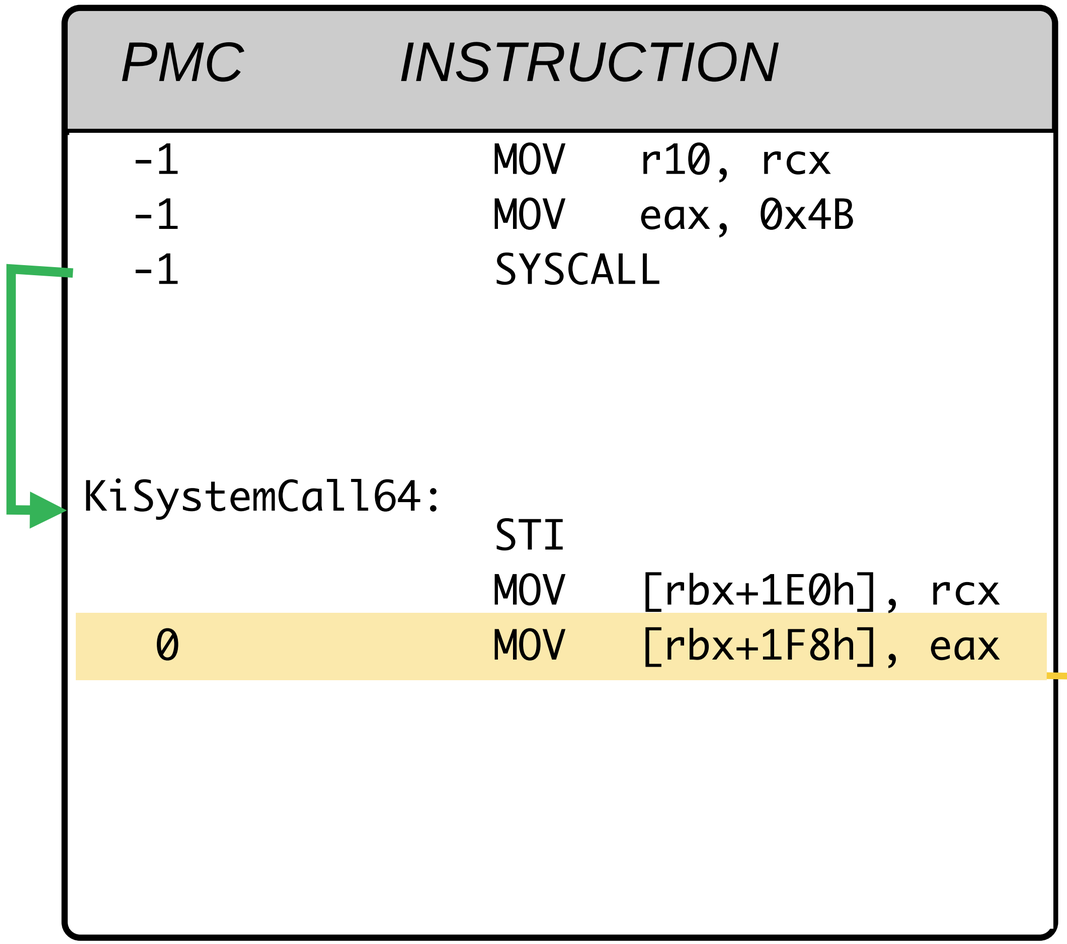
LBR_SELECT: 0xFE (FAR_BRANCH in RING0)

LBR FROM IP	LBR TO IP
userspace addr	KiSystemCall64
IRET from Perf ISR	KiSystemCall64 + X
userspace addr	KiPageFault
IRET from Perf ISR	KiPageFault + Y
userspace addr	KiSystemCall64
IRET from Perf ISR	KiSystemCall64 + Z

IMPLEMENTATION WITH LBR



SUPPORTING VIRTUALIZED ENVIRONMENTS (NO LBR)



PMU ISR

```
//Get saved IP
ip = KTRAP_FRAME.RIP;

//check if IP is in KiSystemCall64
if rdmsr(IA32_LSTAR) < ip &&
   ip < (IA32_LSTAR + 0x290)
{
    //This was a syscall
}
```

CHALLENGES

- ~~Finding the PMU Interrupt~~
- ~~CPU Hot-Plugging~~
- Interrupt instruction skid
  - 99.9999% of trapped Win7 syscalls caught before jump to resolved routine from SSDT
- In other words, implementation on Windows/x64 is much easier than ARM

DEMO

LIMITATIONS

- PMU Registers are not persistent to a core reset
- PMU registers could be modified by other kernel code
  - PMU Watchdog may be necessary, a cloaked thread that monitors for someone changing PMU and changing back
- Detection could be based on increase in PMU interrupts serviced, or just presence of particular values in PMU registers
- That said, this is still a practical approach towards rootkits

RESULTS

- Evades Kernel Patch Protection, including PatchGuard
- Could be extended to hook other IDT/EVT entries beyond SYSCALL/SVC
- Overhead is quite low:
  - Benchmarking "real-world" usage is tough
  - PassMark and JavaScript benchmarks used (2-6% Android, <10% Windows)
  - Not noticeable at all with subjective testing and analysis

ACKNOWLEDGEMENTS

- Cody Pierce, Endgame
- Kenny Fitch, Endgame
- Eric Miller, Endgame
- Jamie Butler, Endgame
- Several others at Endgame
- Researchers that paved the way for PMU assisted security research

QUESTIONS?