HARDWARE-ASSISTED ROOTKITS & INSTRUMENTATION:

ARM EDITION

REcon 2016, Montreal

Matt Spisak
ENDGAME.

ABOUT

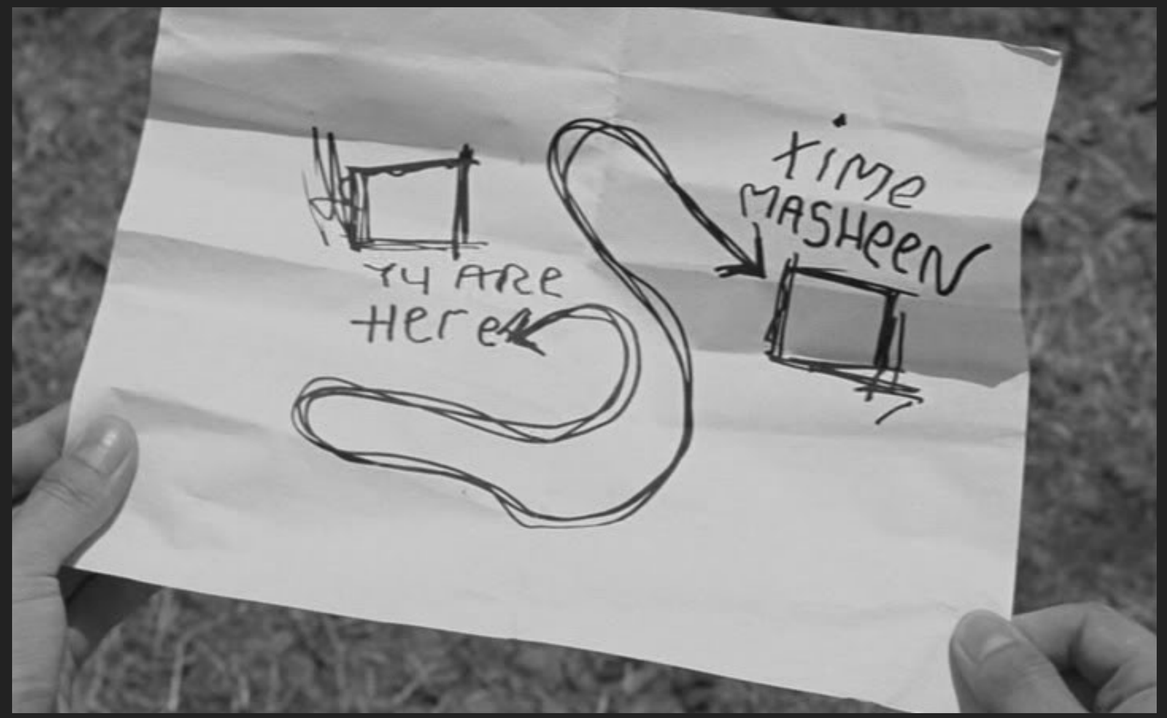
ENDGAME.

- Offense-based approach to security and hunting adversaries
- Research thrusts in malware, threat intel, data science, and exploit prevention

- Matt Spisak (@matspisak)
  - Vulnerability and exploit mitigation research at Endgame
  - Mobile security since Nokia N series (before iPhone)

OUTLINE

- Motivation
- ARM Debug Architecture
- Tracing and Instrumentation
- Rootkits
- ~~TrustZone~~
- Exploit Mitigations



DEBUGGING EMBEDDED SYSTEMS IS COMPLICATED

	✓	✗
Hardware	▸ JTAG is a gold standard ▸ Custom dev boards + Virtualization extensions	▸ JTAG access can be hit/miss ▸ Destructive ▸ Expensive
Software	▸ Portable, scalable ▸ existing tools for HLOS like iOS, Android	▸ Can be tightly coupled to OS ▸ Often limited to PL0/EL0 ▸ Lots of reinventing wheel
Emulation	▸ Scalable and powerful ▸ Cost-effective ▸ Sometimes a good option (e.g. CTF)	▸ Lack support for HW interfaces ▸ Requires big time investment

SEARCHING FOR ALTERNATIVES

- Whats a good general approach?
- Personal philosophy:
  - Always make use of real hardware
  - Lean towards software-based tools
- *GOAL: find common ARM architectural debug features accessible from software (on COTS devices)*

ARM DEBUG ARCHITECTURE

INVASIVE DEBUG

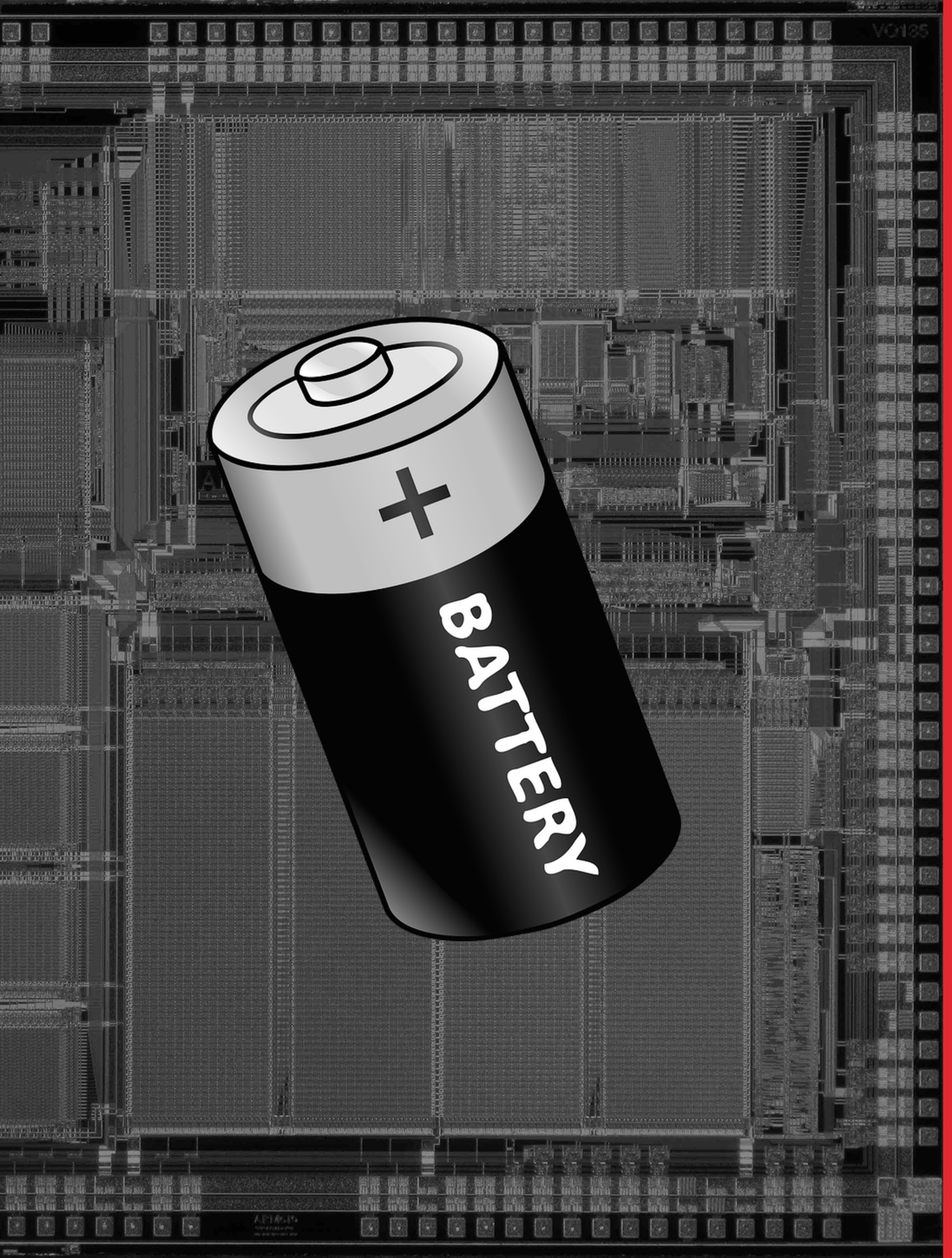
- Debug-modes: Monitor, Halting, or None
  - Software debug events: BKPT, breakpoint, watchpoint, vector trap
  - Halting debug events result in processor entering debug state
- Support driven by DBGEN and SPIDEN authentication signals
  - if DBGEN is low –> BKPT instruction only event supported
  - Authentication signals typically controlled externally
- Without DBGEN, options are limited

NON-INVASIVE DEBUG

- **Trace**: Embedded Trace Buffer (ETB) / CoreSight Program Flow Trace (PFT)
  - PFT/PTM generates traces for waypoints: branch & exception instructions
  - Accessible from external and software (coprocessor or memory-mapped)
  - PFT/PTM can be locked (ETMLAR) - only writeable in memory-mapped
  - memory-mapped access is *IMPLEMENTATION DEFINED*
  - Trace drivers in Android kernel check CoreSight fuse status
  - A potential software-based debug feature for COTS devices

NON-INVASIVE DEBUG

- Sample-based Profiling
  - Registers for sampling Program Counter and Context ID
  - No CP14 visibility, optional memory-mapped and external interfaces
- PMU
  - Focus of remainder of talk



NOT THIS PMU.

THIS PMU.

performance counters

PERFORMANCE MONITORING UNIT (PMU)

- Optional extension, but recommended
- Interfaces: CP15 (mandatory), memory-mapped (optional), external (optional)
- Dates back to ARMv6, common in ARM11, Cortex-R, Cortex-A
- 1 cycle counter, up to 31 general counters
- Set of event filters for counting
- Support for interrupts on counter overflow

sampling period

PERFORMANCE MONITORING UNIT (PMU)

- Provides real-time feedback on system
- Useful for software/hardware engineers
- Diagnose bugs
- Tools:
  - ARM DS-5 Streamline
  - Linux perf / oprofile

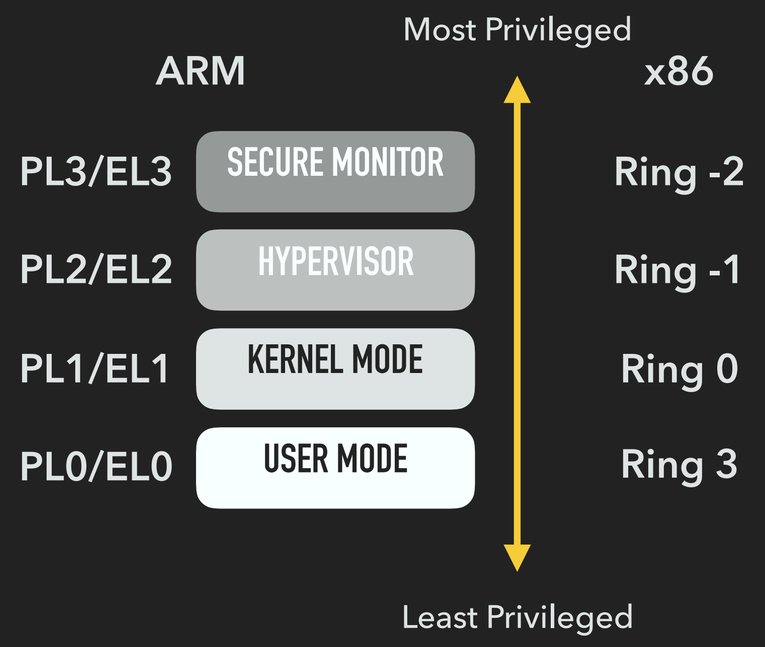
ARM DS-5 Streamline

TERMINOLOGY & ABBREVIATIONS

- PMU - Performance Monitoring Unit
- PMI - Performance Monitoring Interrupt
- PMC - Performance Monitoring Counter

ARM Exception Vector Table (EVT)

EXCEPTION	
Reset	
Undefined Instruction	
SVC	Supervisor Call (e.g. SYSCALL)
Prefetch Abort	BKPT, or code Page Fault
Data Abort	Data Page Fault
IRQ	Interrupts (Normal World)
FIQ	Fast Interrupts (Secure World)



PMU RELATED WORK

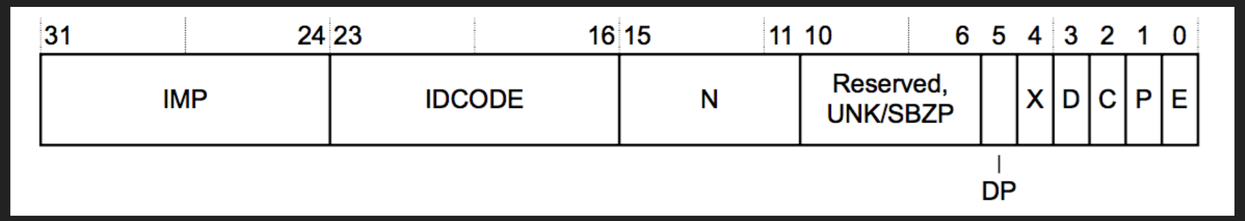
- "Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture", [Vogl, Eckert]
- ROP detection with PMU using mispredicted RET [Wicherski], [Li, Crouse]
- Rootkit detection with performance counters [Wang, Karri]
- Control-flow integrity using BTS [Xia et al]
- Control-flow integrity using PMU [Endgame] - BlackHat USA 2016
- All prior art is focused on Intel / x86 architecture

SAMPLE ARM PMU EVENTS

EVENT TYPE	EVENT CODE
LD_RETIRED: Load instruction executed	0x06
ST_RETIRED: Store instruction executed	0x07
INST_RETIRED: Instruction executed	0x08
PC_WRITE_RETIRED: Software change of PC	0x0C
BR_RETURN_RETIRED: Branch Return retired	0x0E
BR_MISP_PRED: Branch mispredicted	0x10
L1I_CACHE: Level 1 instruction cache access	0x14

PMU REGISTERS

- PMCR - Control Register



  - N: Number of counters
  - E: Enable / Disable all counters
  - ARMv6: *MRC/MCR p15, 0, <Rd>, c15, c12, 0*
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c12, 0*

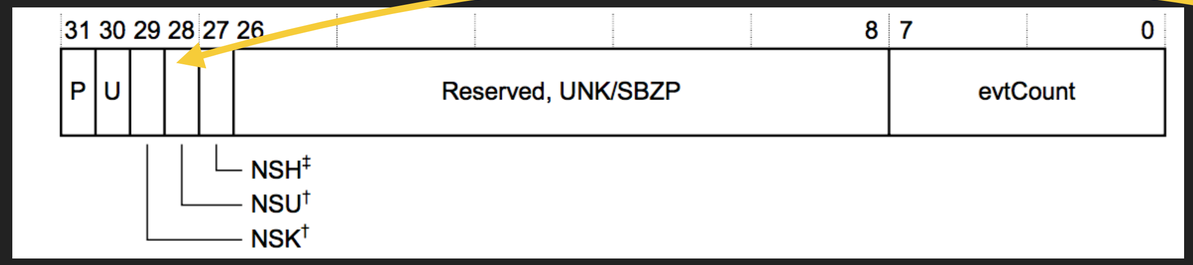
PMU REGISTERS – CONFIGURE COUNTERS

- PMCNTENSET - Enable Counter
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c12, 1*
- PMCNTENCLR - Disable Counter
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c12, 2*
- PMSELR - Counter Selection Register
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c12, 5*

Use this register prior to read/write of event type or counter registers

PMU REGISTERS – CONFIGURE COUNTERS

- PMXEVTYPER - Counter Event Filter Register



  - Selects event and modes to count
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c13, 1*

- *PMXEVTCNTR - Event Counter Register*
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c13, 2*

EVENT CODE	MODES INCLUDED
0x6800000C	Branches in Secure PL1 and HYP
0x6000000C	Branches in Secure PL1
0x9800000C	Branches in Secure PL0 and HYP
0x9000000C	Branches in Secure PL0
0x3800000C	Branches in Secure PL0,PL1,HYP
0x4000000C	Branches in non-secure PL1
0x8000000C	Branches in non-secure PL0

PMU REGISTERS – CONFIGURE COUNTERS

```
//Enable armv7 PMU Counters
MRC p15, 0, R1, c9, c12, 0
ORR R1, R1, #1
MCR p15, 0, R1, c9, c12, 0

//Set PMC1 to count Instructions Executed
MOV R1, #1
MCR p15, 0, R1, c9, c12, 5 //PMSELR
MOV R1, #0x8
MCR p15, 0, R1, c9, c13, 1 //PMXEVTYPER

//Initialize PMC1 to -3
MOV R1, #0xFFFFFFFD
MCR p15, 0, R1, c9, c13, 2 //PMXEVTCNTR

//Enable PMC1
MOV R1, #1
MCR p15, 0, R1, c9, c12, 1 //PMCNTENSET
```

PMU REGISTERS – CONFIGURE INTERRUPTS

- PMINTENSET - Interrupt Enable Register
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c14, 1*
- PMINTENCLR - Interrupt Disable Register
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c14, 2*
- PMOVSR - Overflow Status Register
  PMOVSET - Overflow Status Set Register
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c12, 3*
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c9, c14, 3*

PMU REGISTERS – CONFIGURE INTERRUPTS

```
//Enable Interrupts for PMC1 and PMC2
MOV R1, #3
MCR p15, 0, R1, c9, c14, 1 //PMINTENSET
```

```
//Read and Clear Overflow on Interrupt
MRC p15, 0, R0, c9, c12, 3 //PMOVSR
MCR p15, 0, R0, c9, c12, 3 //PMOVSR
```

DO YOU EVEN COUNT?

- DBGAUTHSTATUS
  - Lists whether invasive/non-invasive debug are supported in secure and non-secure worlds
  - ARMv7: *MRC/MCR p14, 0, <Rd>, c7, c14, 6*
- ID_DFR0
  - Lists PMU version supported (if any)
  - ARMv7: *MRC/MCR p15, 0, <Rd>, c0, c1, 2*

THE CENTER FOR CHIPS WHO CAN COUNT GOOD

DEVICE	CHIPSET	DBGAUTHSTATUS	VERSION
Motorola Nexus 6	Qualcomm Snapdragon 805 (4x Krait Core)	Non-Invasive Debug (NIDEN) Enabled	PMUv2
Amazon Fire HD 7"	MediaTek MT8135 (2x Cortex-A15 + 2x Cortex A7)	Non-Invasive Debug (NIDEN) Enabled Secure Non-Invasive Debug (SPNIDEN) Enabled	PMUv2
Samsung Galaxy Note 2	Samsung Exynos 4412 (4x Cortex-A9)	Non-Invasive Debug (NIDEN) Enabled Invasive Debug (DBGEN) Enabled	PMUv2
Huawei Ascend P7	HiSilicon Kirin 910T (4x Cortex-A9)	Non-Invasive Debug (NIDEN) Enabled Invasive Debug (DBGEN) Enabled Secure Non-Invasive Debug (SPNIDEN) Enabled Secure Invasive Debug (SPIDEN) Enabled	PMUv2
Multiple	Broadcom BCM4356 WiFi Chip (Cortex R4)	Non-Invasive Debug (NIDEN) Enabled	PMUv1

CASE STUDY: PMU TRACING

APPROACH

- Make the PMU more invasive with frequent PMC-based traps
- CoreSight Program Flow Trace (PFT) captures waypoints (i.e. branches)
- We can come pretty close to PFT Trace using the PMU:
  - Count all branches: predicted and mispredicted
  - Interrupt all the things: set our counter(s) to -1
  - Use our ISR as the instrumentation logic

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	*INSTRUCTION*	
-1	BL	func
func:		
	STMFD	SP!, {R0-R2,R4-R9,LR}
	MOV	R8, R1
	MOV	R1, SP
	MOV	R2, R2
	LDR	R7, [SP]
	CMP	R7, #0
	BEQ	error
error:		
	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	INSTRUCTION	
-1	BL	func
	func:	
overflow 0	STMFD	SP!, {R0-R2,R4-R9,LR}
	MOV	R8, R1
	MOV	R1, SP
	MOV	R2, R2
	LDR	R7, [SP]
	CMP	R7, #0
	BEQ	error
	error:	
	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	*INSTRUCTION*	
-1	BL	func
func:		
0	STMFD	SP!, {R0-R2,R4-R9,LR}
-1	MOV	R8, R1
	MOV	R1, SP
	MOV	R2, R2
	LDR	R7, [SP]
	CMP	R7, #0
	BEQ	error
error:		
	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	*INSTRUCTION*	
-1	BL	func
func:		
0	STMFD	SP!, {R0-R2,R4-R9,LR}
-1	MOV	R8, R1
-1	MOV	R1, SP
	MOV	R2, R2
	LDR	R7, [SP]
	CMP	R7, #0
	BEQ	error
error:		
	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	*INSTRUCTION*	
-1	BL	func
func:		
0	STMFD	SP!, {R0-R2,R4-R9,LR}
-1	MOV	R8, R1
-1	MOV	R1, SP
-1	MOV	R2, R2
	LDR	R7, [SP]
	CMP	R7, #0
	BEQ	error
error:		
	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	INSTRUCTION
-1	BL func
func:	
0	STMFD SP!, {R0-R2,R4-R9,LR}
-1	MOV R8, R1
-1	MOV R1, SP
-1	MOV R2, R2
-1	LDR R7, [SP]
	CMP R7, #0
	BEQ error
error:	
	MOV R4, #0xFFFFFFF7
	ADD SP, SP, #0xC

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	*INSTRUCTION*	
-1	BL	func
func:		
0	STMFD	SP!, {R0-R2,R4-R9,LR}
-1	MOV	R8, R1
-1	MOV	R1, SP
-1	MOV	R2, R2
-1	LDR	R7, [SP]
-1	CMP	R7, #0
	BEQ	error
error:		
	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	INSTRUCTION	
-1	BL	func
func:		
0	STMFD	SP!, {R0-R2,R4-R9,LR}
-1	MOV	R8, R1
-1	MOV	R1, SP
-1	MOV	R2, R2
-1	LDR	R7, [SP]
-1	CMP	R7, #0
-1	BEQ	error
error:		
	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	INSTRUCTION	
-1	BL	func
func:		
0	STMFD	SP!, {R0-R2,R4-R9,LR}
-1	MOV	R8, R1
-1	MOV	R1, SP
-1	MOV	R2, R2
-1	LDR	R7, [SP]
-1	CMP	R7, #0
-1	BEQ	error
error:		
0	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

overflow

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

APPROACH – BRANCH TRACING

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	INSTRUCTION	
-1	BL	func
func:		
0	STMFD	SP!, {R0-R2,R4-R9,LR}
-1	MOV	R8, R1
-1	MOV	R1, SP
-1	MOV	R2, R2
-1	LDR	R7, [SP]
-1	CMP	R7, #0
-1	BEQ	error
error:		
0	MOV	R4, #0xFFFFFFF7
-1	ADD	SP, SP, #0xC

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

BUT WHAT ABOUT LINUX PERF?

- We want a custom ISR for instrumentation
- Too tightly coupled to Linux
- Invoking API's != learning
- But perf source can be useful for understanding PMU interfaces

WHERE'S THE PMU INTERRUPT?

ARM GENERIC INTERRUPT CONTROLLER (GIC) SPECIFICATION

INTID	Interrupt type	Details
ID0 – ID15	SGI	These interrupts are local to a CPU interface.
ID16 – ID31	PPI	
ID32 – ID1019	SPI	Shared peripheral interrupts that the Distributor can route to either a specific PE, or to any one of the PEs in the system that is a participating node, see *Participating nodes on page 3-44*.

ARM GIC Architecture Specification

- SGI: Software Generated Interrupts
  PPI: Private Peripheral Interrupts
  SPI: Shared Peripheral Interrupts
- ARM GIC spec recommends PMU Overflows to use INTID 23

CHALLENGE: FINDING PMU INTERRUPTS

- Device Tree Source

```
cpu-pmu {
          compatible = "qcom,krait-pmu";
          qcom,irq-is-percpu;
          interrupts = <1 7 0xf00>;
};
```

PPI INT# = 16 + 7 = 23

- Brute Force
  - Register all unused PPI's & SPI's, trigger PMIs, diff /proc/interrupts

- Implementation:
  - Android: request_percpu_irq(), request_threaded_irq()
  - Embedded firmware: patch IRQ vector handler

CHALLENGE: INTERRUPT SHADOW

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	*INSTRUCTION*	
-1	BL	func
func:		
	LDR	R7, [SP]
	CMP	R7, #0
	BEQ	error
error:		
	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

CHALLENGE: INTERRUPT SHADOW

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

	PMC	INSTRUCTION	
	-1	BL	func
	func:		
overflow	0	LDR	R7, [SP]
		CMP	R7, #0
		BEQ	error
	error:		
		MOV	R4, #0xFFFFFFF7
		ADD	SP, SP, #0xC

CHALLENGE: INTERRUPT SHADOW

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

	PMC	INSTRUCTION	
	-1	BL	func
		func:	
overflow	0	LDR	R7, [SP]
	0	CMP	R7, #0
		BEQ	error
		error:	
		MOV	R4, #0xFFFFFFF7
		ADD	SP, SP, #0xC

CHALLENGE: INTERRUPT SHADOW

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	INSTRUCTION	
-1	BL	func
func:		
0	LDR	R7, [SP]
0	CMP	R7, #0
0	BEQ	error
error:		
	MOV	R4, #0xFFFFFFF7
	ADD	SP, SP, #0xC

overflow

CHALLENGE: INTERRUPT SHADOW

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

	PMC	INSTRUCTION	
	-1	BL	func
		func:	
overflow	0	LDR	R7, [SP]
	0	CMP	R7, #0
	0	BEQ	error
		error:	
	1	MOV	R4, #0xFFFFFFF7
		ADD	SP, SP, #0xC

CHALLENGE: INTERRUPT SHADOW

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	INSTRUCTION	
-1	BL	func
func:		
0	LDR	R7, [SP]
0	CMP	R7, #0
0	BEQ	error
error:		
1	MOV	R4, #0xFFFFFFF7
1	ADD	SP, SP, #0xC

overflow

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

CHALLENGE: INTERRUPT SHADOW

PMC1: 0xFFFFFFFF (-1) Event: 0x0C (All Branches)

PMC	INSTRUCTION	
-1	BL	func
func:		
0	LDR	R7, [SP]
0	CMP	R7, #0
0	BEQ	error
error:		
1	MOV	R4, #0xFFFFFFF7
1	ADD	SP, SP, #0xC

overflow

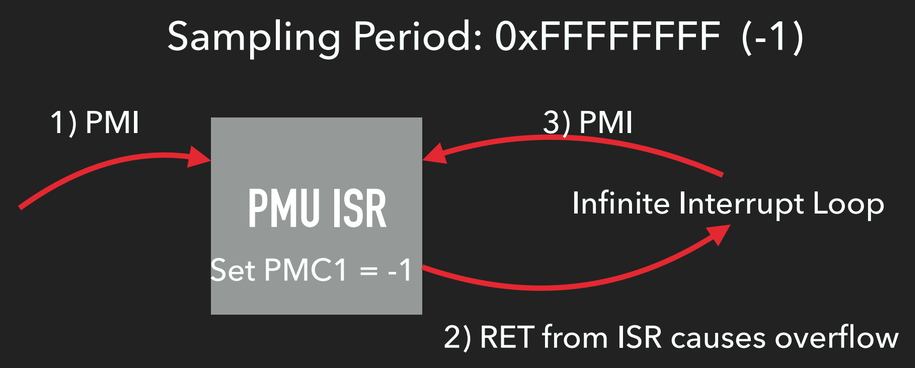
Interrupt Shadow
Skid = 4 Instructions

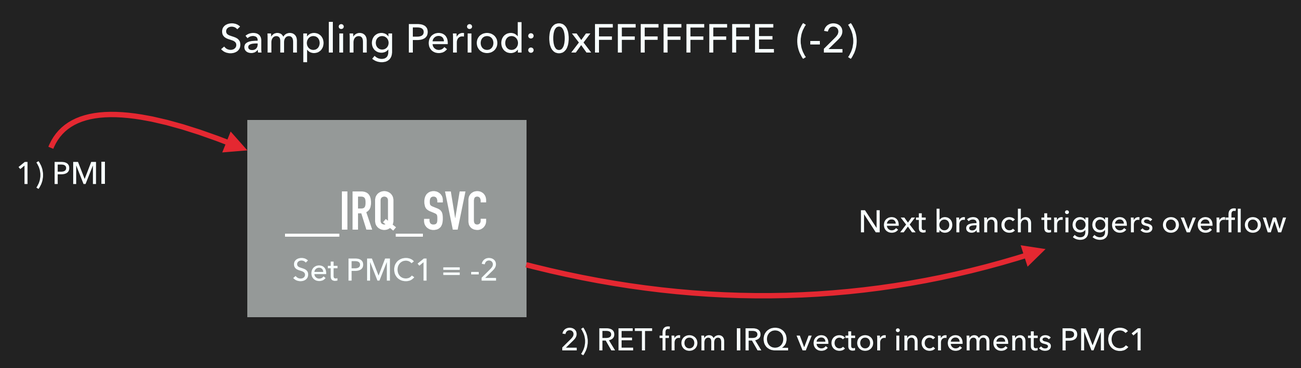
Causes miss of up to 15% covered basic blocks

PMU ISR
- CAPTURE PC
- CAPTURE REGS
- MEMORY SNAPSHOT
- RESET COUNTER

OTHER CHALLENGES

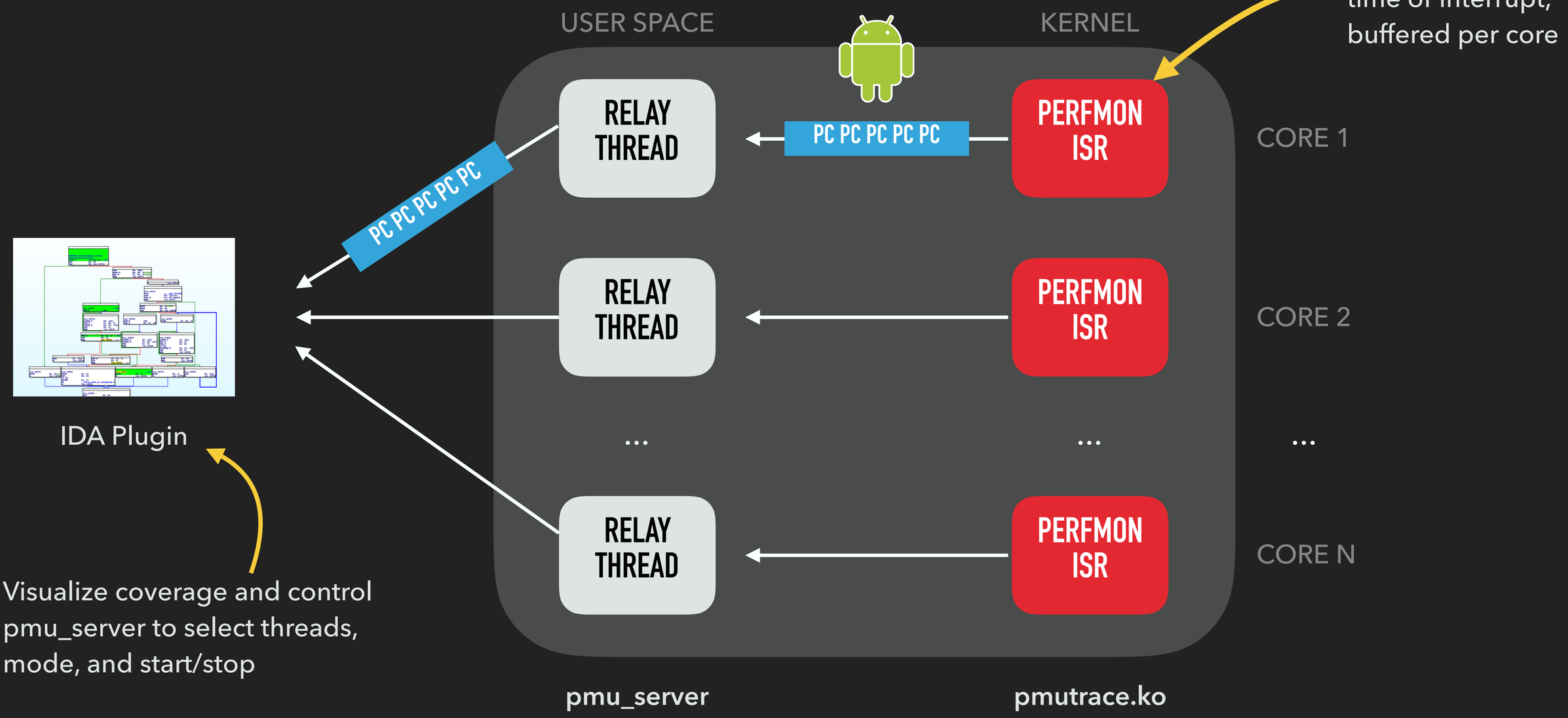
- CPU Hot-Plugging – easy solution for Android: register_hotcpu_notifier()
- Lack of Last Branch Recording feature on ARM
- Complicated kernel mode instrumentation: use sampling period of -2





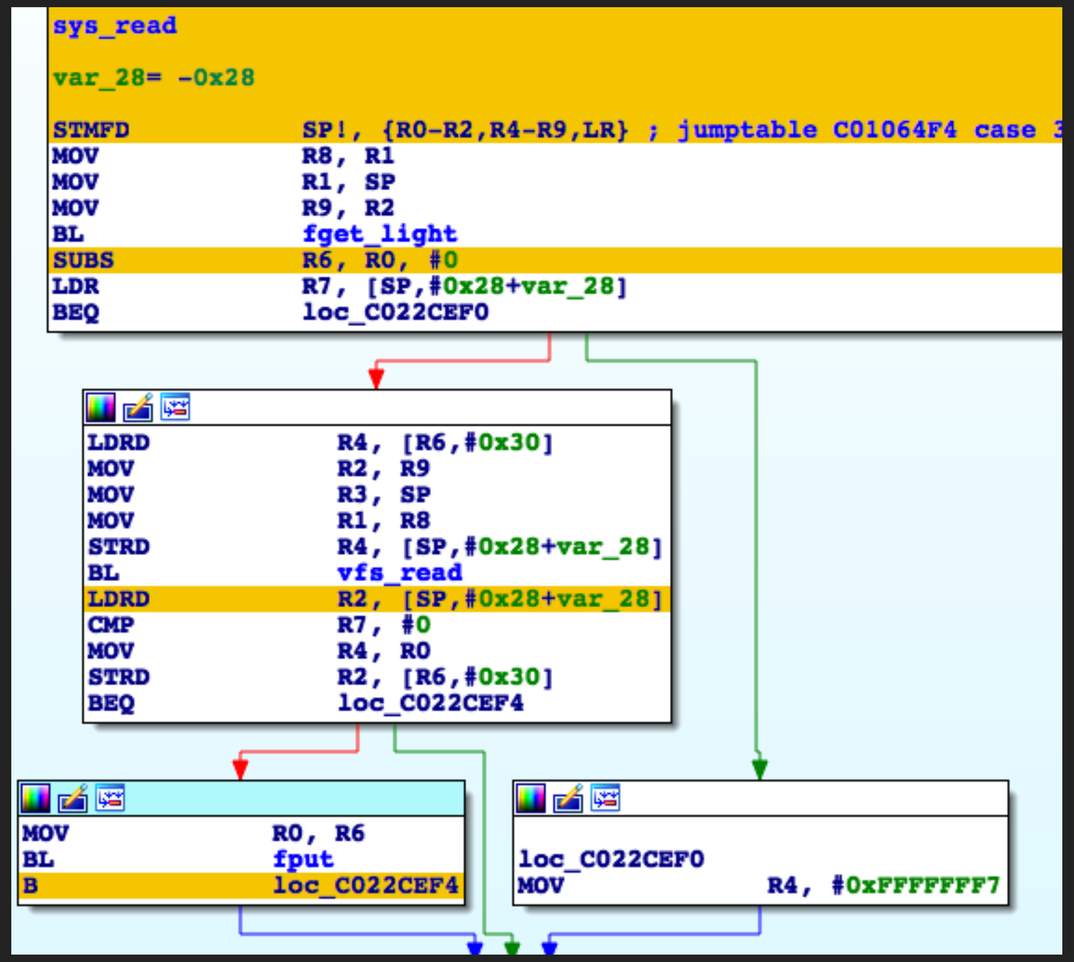
- Requires small patch to entry-armv.S (or hot patch)

ANDROID PROTOTYPE



CONNECTING THE DOTS

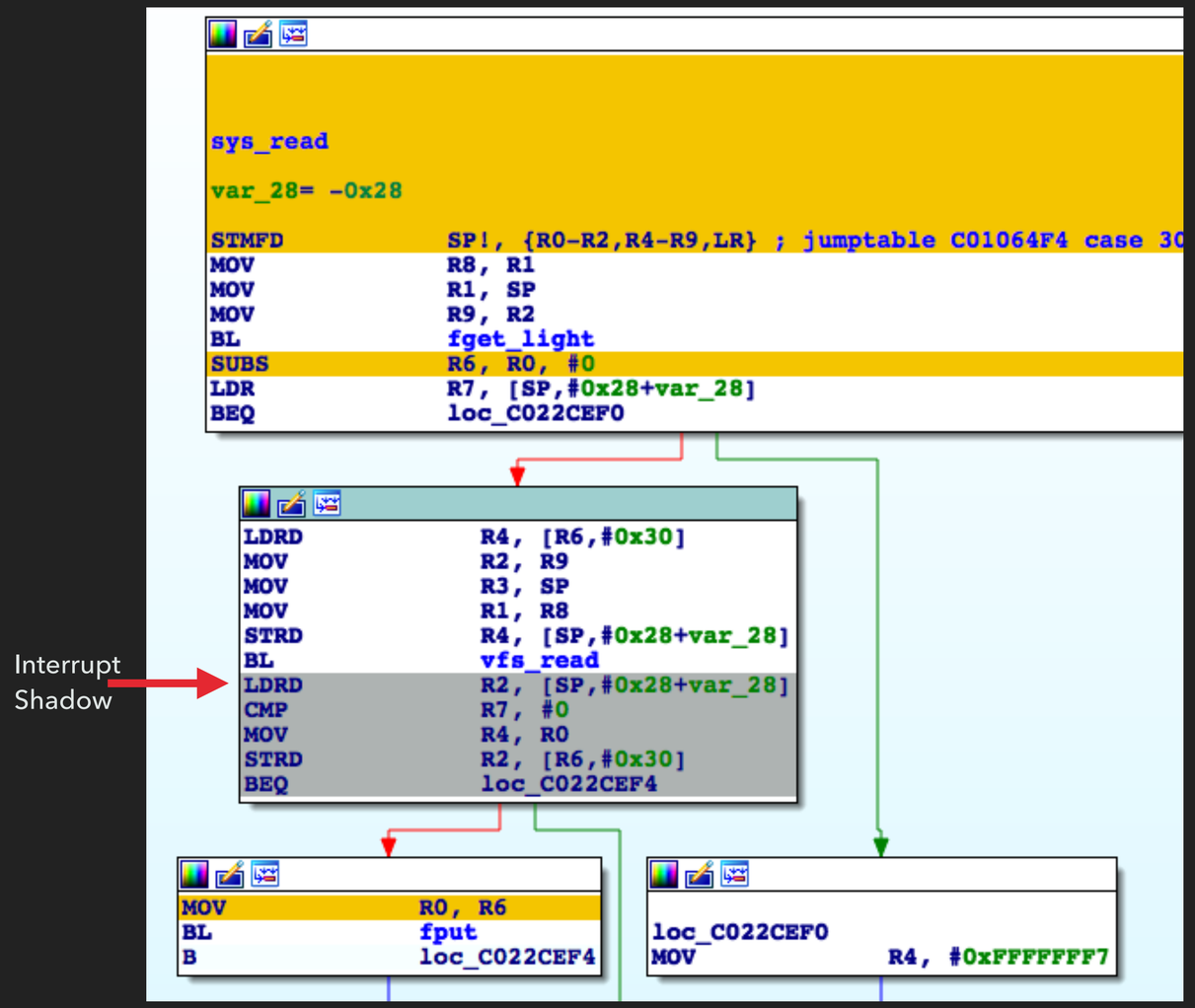
- Use IDA to our advantage
  - For each PMU waypoint:
    - Color/count all instructions in Basic Block
    - If only 1 xref from basic block: count/color it
    - If only 1 xref to basic block: count/color it



Example of a perfect PMU branch tracing run

CONNECTING THE DOTS

- Interrupt shadow
  - Basic block xref algorithm helps fill in missed blocks
  - Fuzzing / code coverage will eventually be interrupted in this block
  - Could improve by adding 2nd counter to count instructions between interrupts



Example of PMU trace missing basic block

DEMO: PMU TRACING

DEVICE REQUIREMENTS:

- ROOTED
- CONFIG_MODULES OPTION (NOT AS COMMON)
- CONFIG_PREEMPT OPTION (COMMON)
- IRQ HANDLER PATCH (PL1/EL1)

ANDROID INSTRUMENTATION. SO WHAT?

- Recall approach is hardware-assisted - not tied to a specific OS
- Less invasive than BKPT tracing
- Supports both user mode and kernel mode instrumentation
- Not limited to branch tracing, other potential instrumentation use-cases
- And these chips can count too:
  - Broadcom WiFi; Intel/Infineon, MediaTek + other ARM Cellular Basebands
  - Apple ARM SoCs
  - PowerPC, MIPS

CASE STUDY: PMU ROOTKITS

PRIOR ART IN ARM ROOTKITS

- Traditional rootkits: modify syscall table or EVT [Phrack Issue 68]
- Suterusu performs hot patching of kernel functions [Coppola]
- Cloaker toggles SCTLR to move EVT [David et al]
- Clock Locking Beats explores using CPU governor for hiding cycles [Thomas]
- TrustZone based rootkit [Roth]

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


PMU ASSISTED
NSTRUMENTATION
PMU ASSISTED
TRACING
PMU ASSISTED
ROOTKITS
PMU ASSISTED
TRUSTZONE
PMU ASSISTED
EXPLOIT PREVENTION
PMU ASSISTED
DEFENSE
PMU ASSISTED
HYPERVISOR

QUICK NOTE ON ARM LICENSES

- ARM Core License
  - Use core ARM designs
- ARM Architectural license
  - Enables custom cores provided it implements an ARM instruction set
  - Examples: Qualcomm Scorpion/Krait/Kryo, Apple A6/A7/etc.

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			√	√	?	?	?

DOWN THE RABBIT HOLE

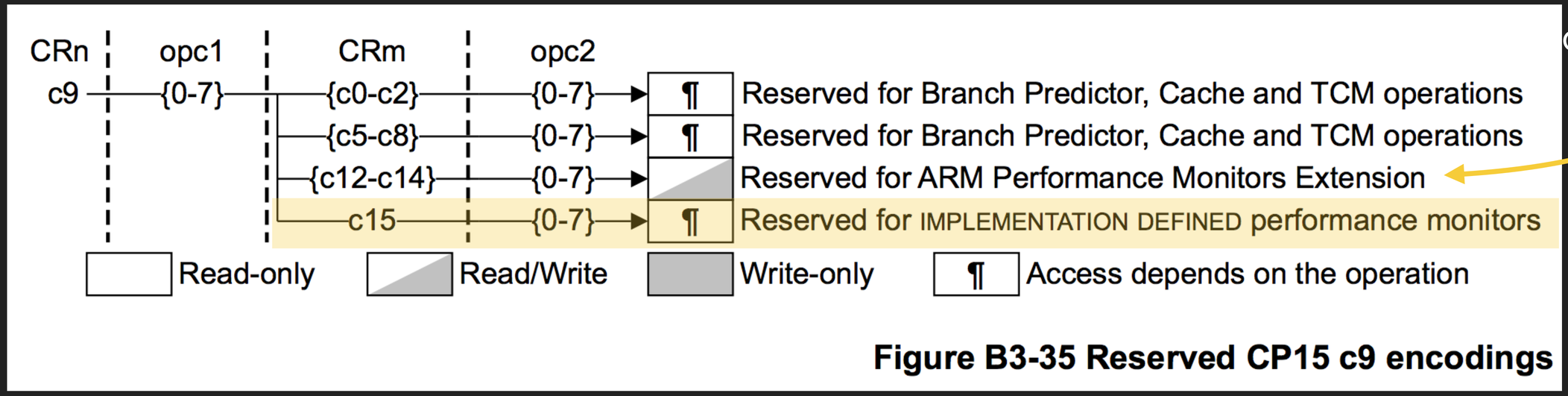


Figure B3-35 Reserved CP15 c9 encodings

ARM Architecture Manual ARMv7-A&R

Covered in earlier slides

- Chipset vendors with proprietary PMU implementations:
  - Qualcomm
  - Apple
  - Likely others

QUALCOMM®

SCORPION	KRAIT	KRYO
2008	2012	2015
ARMv7	ARMv7	ARMv8
1-2 Cores	2 or 4 cores	4 cores
Snapdragon S1/S2/S3	Snapdragon S4/400/600/800/805	Snapdragon 818/820/823
BlackBerry Bold 9900 Samsung Galaxy S2 (LTE) Nokia Lumia 900 HTC Droid Incredible	Nexus 4/5/6/7 Samsung Galaxy S4/S5 HTC One M8 LG G3	LG G5 Samsung Galaxy S7 HTC 10 Xiomi Mi 5

QUALCOMM KRAIT PMU

- Adds 4 event select registers: 1 for Venum VFP, 3 for other components of CPU
- Krait event encoded using code + group + region => (code << 8 * group)
- ARM event select register (PMXEVTYPER) set to link to Krait region and group

Krait Region 0	*Krait Region 1*	*Krait Region 2*
MRC/MCR p15, 1, <Rd>, c9, c15, 0	*MRC/MCR p15, 1, <Rd>, c9, c15, 1*	*MRC/MCR p15, 1, <Rd>, c9, c15, 2*
Interrupts/Exceptions + other	?	?
~100 event codes	~128 event codes	~156 event codes
PMXEVTYPER = 0xCC | group	PMXEVTYPER = 0xD0 | group	PMXEVTYPER = 0xD4 | group

Only a few documented in old Scorpion src. Black-box analysis used to determine # of events

QUALCOMM KRAIT PMU

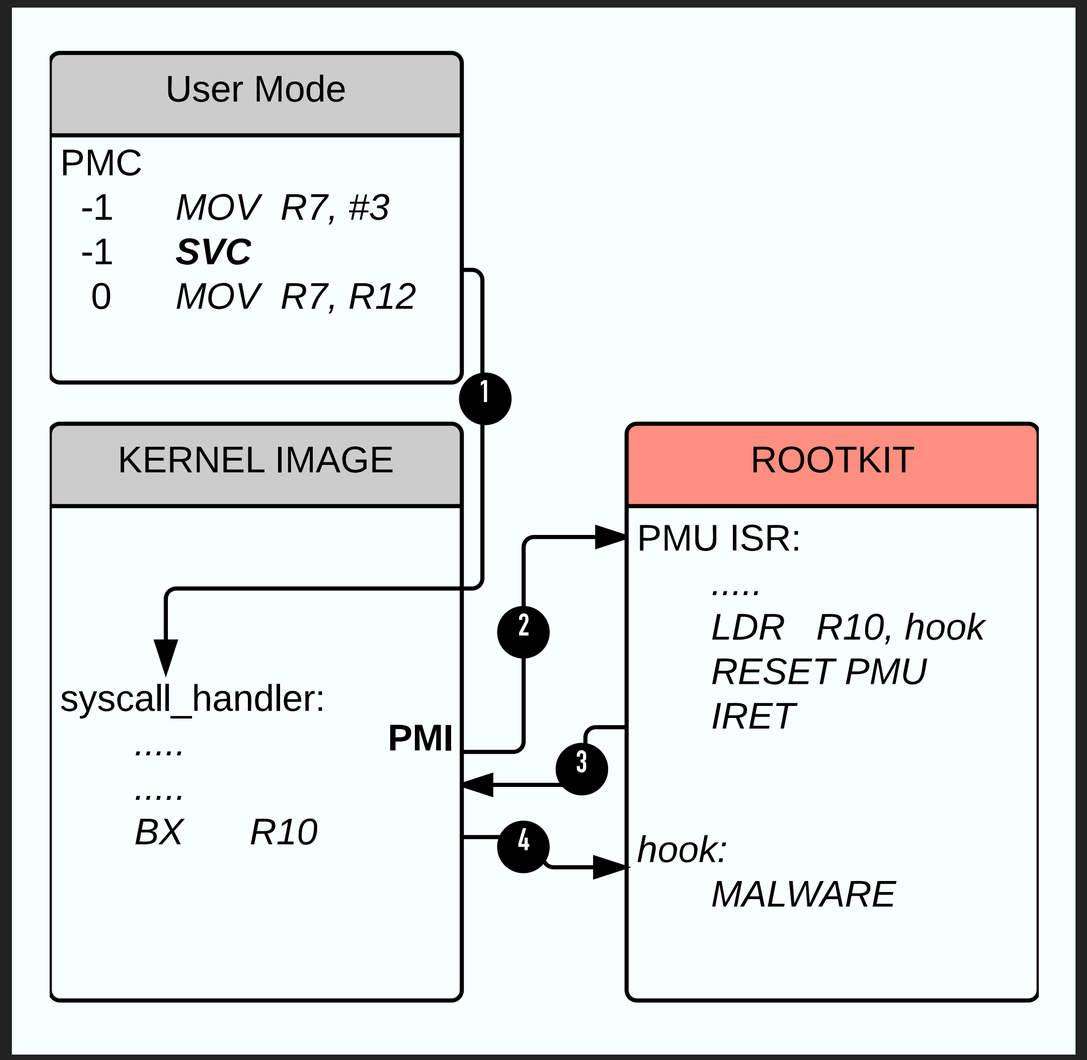
- Configure Krait + ARM PMU to count Prefetch Aborts:
  - Krait Event Code: 0x0B group: 3 Region: 0

```
/*Set Krait Region 0 event selection register
To count Prefetch Aborts*/
MRC p15, 0, R1, c9, c15, 0
ORR R1, R1, #0x8b000000
MCR p15, 0, R1, c9, c15, 0

//Set PMXEVTYPER to point to krait region 0
MOV R1, #0xCF
MCR p15, 0, R1, c9, c13, 1
```

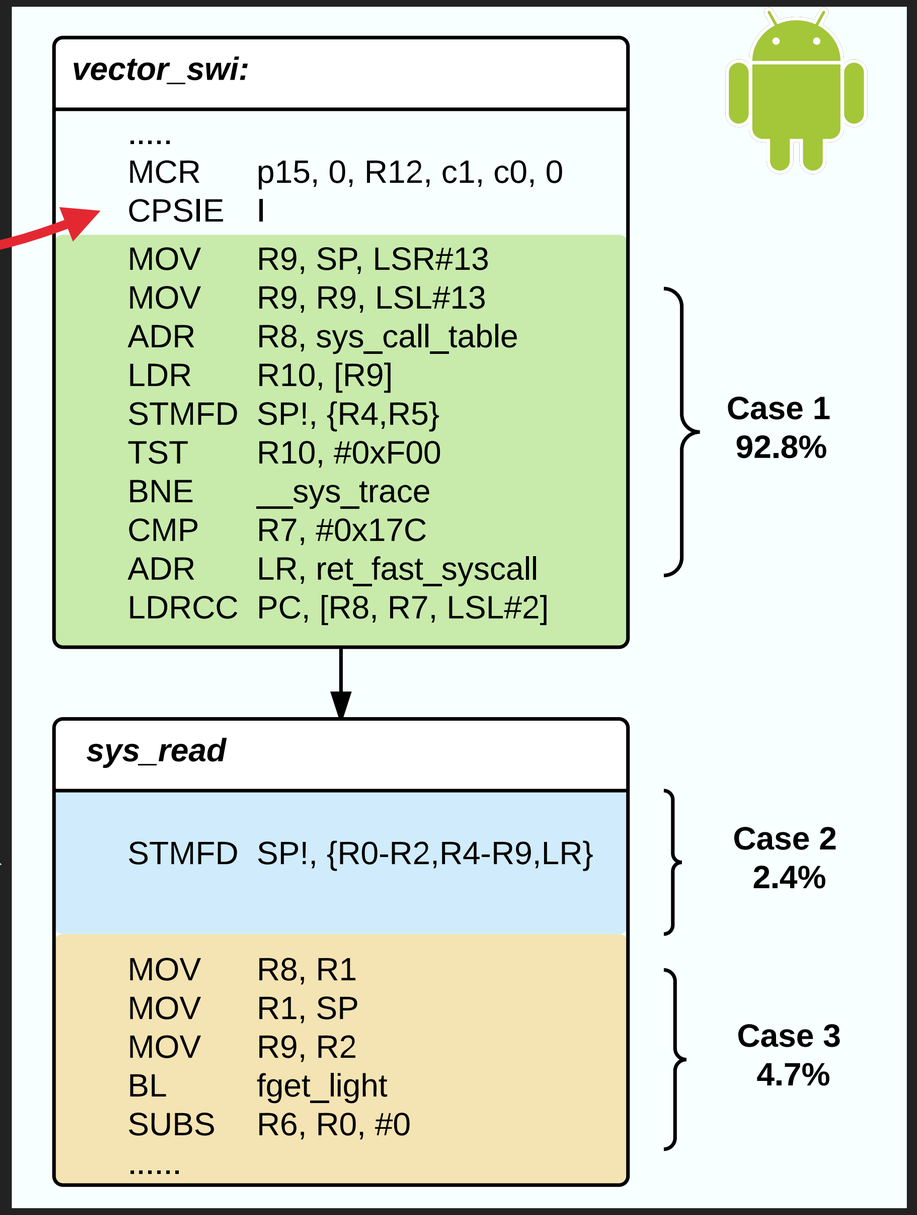
PMU-ASSISTED ROOTKITS

- Trap SVC instructions via PMU
- Use ISR to filter system calls, and redirect code execution after servicing PMI
- Avoids patch protection*
- Installation: a few instructions to initialize PMU registers, and then register ISR for PMU interrupts



CHALLENGE: DELAYED INSTRUCTION SKID

- PMI serviced 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();  //get SVC mode regs
    pregs = task_pt_regs(current); //get user mode regs
    …
    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:

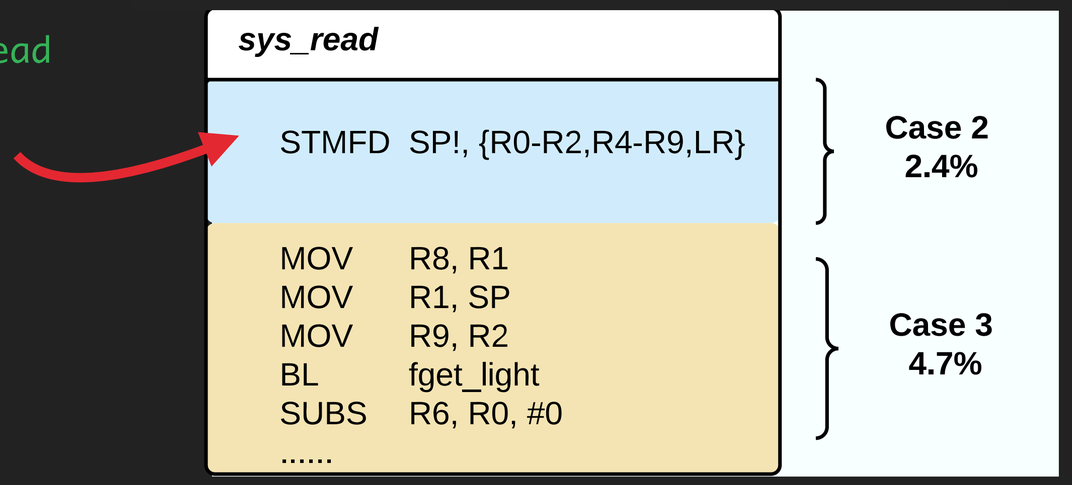
```
.....
MCR     p15, 0, R12, c1, c0, 0
CPSIE   I
MOV     R9, SP, LSR#13
MOV     R9, R9, LSL#13
ADR     R8, sys_call_table
LDR     R10, [R9]
STMFD   SP!, {R4,R5}
TST     R10, #0xF00
BNE     __sys_trace
CMP     R7, #0x17C
ADR     LR, ret_fast_syscall
LDRCC   PC, [R8, R7, LSL#2]
```

Case 1
92.8%

CASE 2: SYSCALL ROUTINE ENTRY POINT

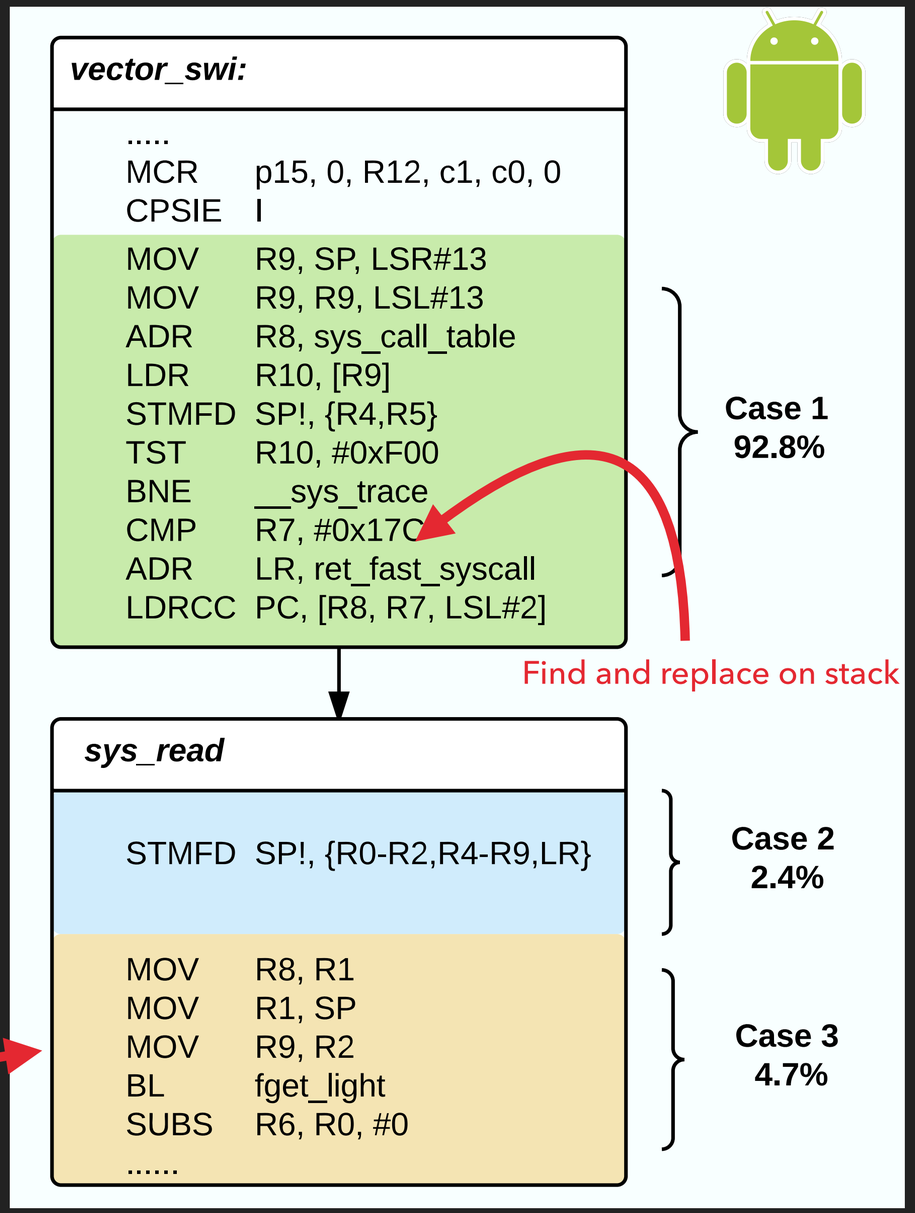
- Replace saved PC with address of hook

```
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

- We will 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 appropriate post_hook function
- post_hook can retrieve original params from saved user mode registers, and modify as necessary



Case 3: Beyond entry point

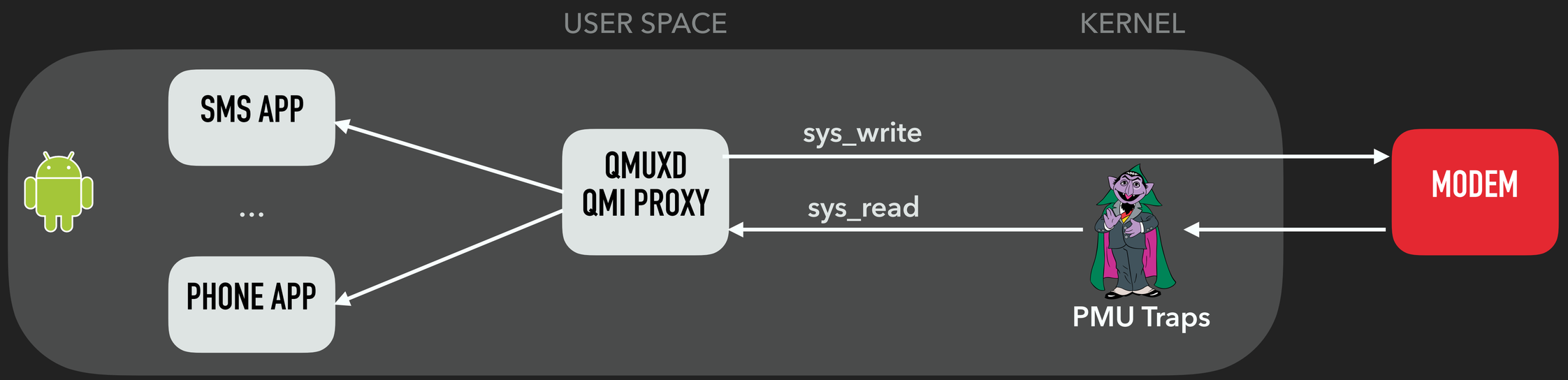
DEMO: PMU ROOTKIT

PROCESS AND FILE HIDING WITH SYS_GETDENTS64 PMU SVC TRAPS

MOTOROLA NEXUS 6
QCOM APQ8084 (KRAIT) CPU

FUN WITH QMI

- Linux rootkits are boring. This is a phone…
- Hook sys_read in context of qmuxd in order to intercept all QMI comms from modem to Android (using only the PMU)



DEMO: PMU ROOTKIT

INTERCEPTING QMI WITH SYS_READ PMU SVC TRAPS

MOTOROLA NEXUS 6
QCOM APQ8084 (KRAIT) CPU

ANALYSIS AND LIMITATIONS

- PMU trap on SVC instructions adds less than 5% overhead (2-3%)
- Should evade current kernel integrity monitor algorithms
- PMU registers do not persist a core reset
- Any other code at PL1/EL1 or higher can read/write the registers

DETECTION STRATEGIES

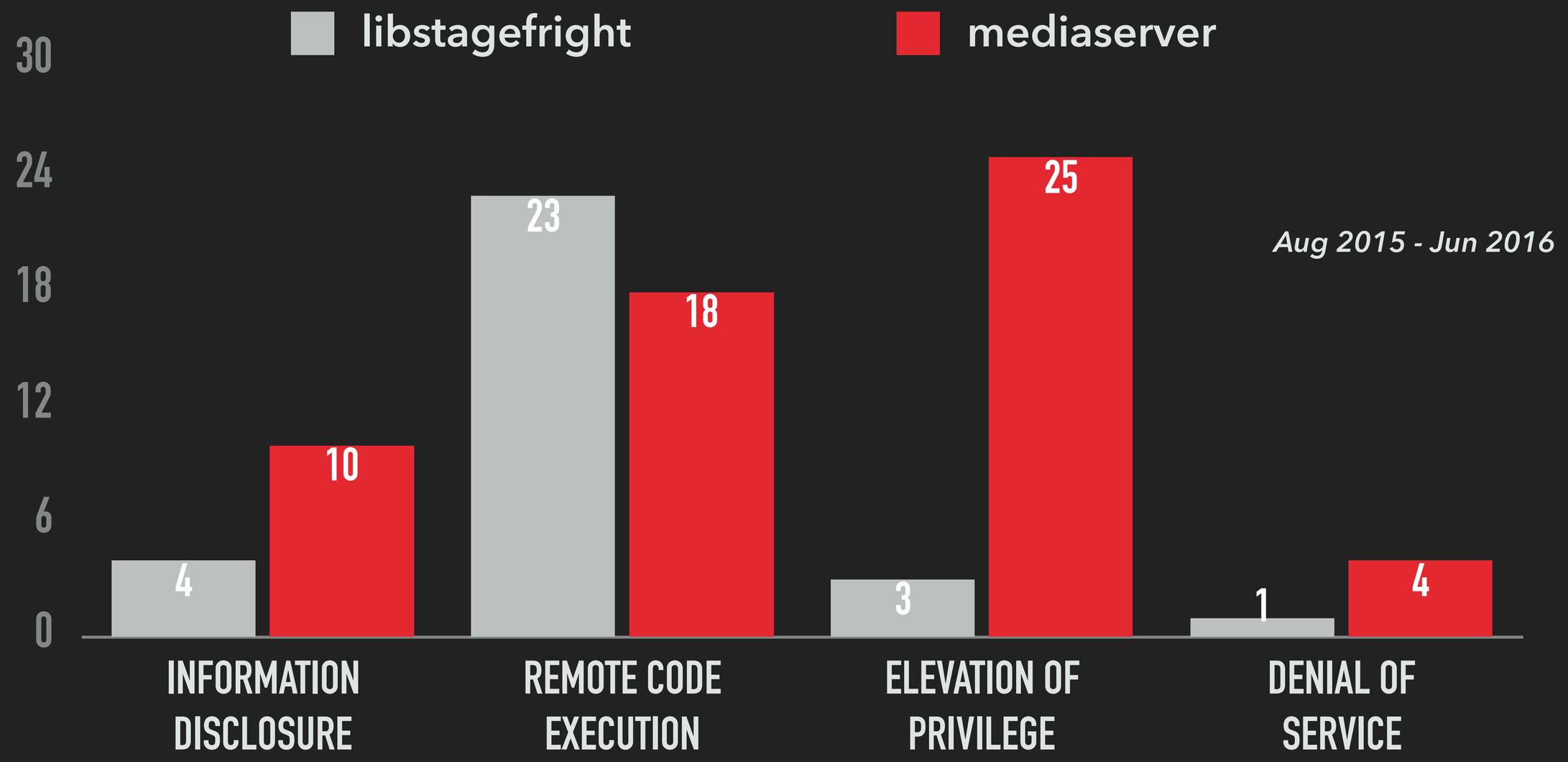
- /proc/interrupts –> easy to modify and cloak
- Reading PMU registers looking for someone counting SVCs
  - Access to PMU registers can be trapped to HYP mode
  - Not all usage of PMU in this way is malicious…
- irq_handler_entry/irq_handler_exit tracepoints
- Validate IRQ handler addresses by iterating radix tree structure
  - PMU Traps on Data & Prefetch Aborts for ShadowWalker?

CASE STUDY: PMU DEFENSE

EXPLOIT DETECTION FROM THE KERNEL

- Trap SVC instructions to perform syscall monitoring
  - Detect ROP behavior (e.g. EMET / ROPGuard checks)
  - Doesn't increase attack surface to protected user space binaries
  - Much easier to implement than Rootkit since no re-direction required
  - Protect COTS binaries (i.e no source/compiler required)
  - No modifications to kernel image - just need ISR registered

ANDROID CVE'S IN MEDIA



DEMO: PMU DEFENSE

BLOCKING STAGEFRIGHT ROP CHAIN FROM THE KERNEL

CVE-2015-3864
POC's courtesy Mark Brand, Google & NorthBit's Metaphor

LG NEXUS 5
QCOM MSM8974 (KRAIT) CPU

FUTURE WORK

- Port instrumentation approach to basebands
- Analyze Apple hardware for PMU features and explore iOS kernel tracing

ACKNOWLEDGEMENTS

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

QUESTIONS?

OR FEEDBACK

mspisak at endgame.com
@matspisak
ENDGAME.

REFERENCES

S. Vogl and C. Eckert, "*Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture*," in *Proceedings of EuroSec'12, 5th European Workshop on System Security*, ACM Press, Apr. 2012.

G. Wicherski, "*Taming ROP on Sandy Bridge: Using Performance Counters to Detect Kernel Return-Oriented Programming.*" SyScan 2013.

X. Li and M. Crouse, "*Transparent ROP Detection using CPU Performance Counters.*"https://www.trailofbits.com/threads/2014/transparent_rop_detection_using_cpu_perfcounters.pdf . Threads 2014.

X. Wang and R. Karri, "*NumChecker: detecting kernel control-flow modifying rootkits by using hardware performance counters*," in *DAC*, ACM, 2013.

Y. Xia, Y. Liu, H. Chen, and B. Zang, "*CFIMon: Detecting violation of control flow integrity using performance counters*," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, IEEE Computer Society, 2012.

M. Coppola, "*Suterusu Rootkit:Inline Kernel Function Hooking on x86 and ARM.*" https://github.com/mncoppola/suterusu

J. M. Thomas, "*Clock Locking Beats: Exploring the Android Kernel and Processor Interactions.*" https://github.com/monk-dot/ClockLockingBeats

T. Roth, "Next Generation Mobile Rootkits." Hack In Paris 2013. https://hackinparis.com/data/slides/2013/Slidesthomasroth.pdf

F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "*Cloaker: Hardware supported rootkit concealment*," in *Proceedings - IEEE Symposium on Security and Privacy*, pp. 296–310, 2008.