Hardware-Assisted Rootkits: Abusing Performance Counters on the ARM and x86 Architectures

Matt Spisak
Endgame, Inc.
mspisak@endgame.com

Abstract

In this paper, a novel hardware-assisted rootkit is introduced, which leverages the performance monitoring unit (PMU) of a CPU. By configuring hardware performance counters to count specific architectural events, this research effort proves it is possible to transparently trap system calls and other interrupts driven entirely by the PMU. This offers an attacker the opportunity to redirect control flow to malicious code without requiring modifications to a kernel image.

The approach is demonstrated as a kernel-mode rootkit on both the ARM and Intel x86-64 architectures that is capable of intercepting system calls while evading current kernel patch protection implementations such as PatchGuard. A proof-of-concept Android rootkit is developed targeting ARM (Krait) chipsets found in millions of smartphones worldwide, and a similar Windows rootkit is developed for the Intel x86-64 architecture. The prototype PMU-assisted rootkit adds minimal overhead to Android, and less than 10% overhead to Windows OS. Further analysis into performance counters also reveals that the PMU can be used to trap returns from secure world on ARM as well as returns from System Management Mode on x86-64.

1 INTRODUCTION

Rootkit authors have been forced to adapt over the past decade as detection methods and countermeasures have been integrated into vendor operating systems and commercial security products. For example, the introduction of Kernel Patch Protection (KPP) and other kernel integrity measurement algorithms such as Microsoft PatchGuard for Windows [1], Samsung KNOX for Android devices [2], and Apple KPP for iOS 9+ devices [3], has mitigated traditional kernel rootkit designs that involve modifications to syscall dispatch tables, exception vector tables, or other critical code and data structures found in the OS. With KPP in place, attackers are often forced to move malicious code to less privileged user-mode, to elevate privileges enabling a hypervisor or TrustZone based rootkit, or to become more creative in their approach to achieving a kernel mode rootkit.

Early advances in rootkit design focused on low-level hooks to system calls and interrupts within the kernel. With the introduction of hardware virtualization extensions, hypervisor based rootkits became a popular area of study allowing malicious code to run underneath a guest operating system [4, 5]. Another class of OS agnostic rootkits also emerged that run in System Management Mode (SMM) on x86 [6] or within ARM TrustZone [7]. The latter two categories, which leverage virtualization extensions, SMM on x86, and security extensions on ARM, are perhaps more advanced in terms of added stealth; however, in some cases this may require an attacker to escalate privileges beyond the kernel level.

In this paper, a novel kernel-level rootkit is introduced, which utilizes the performance monitoring unit (PMU) of the CPU in order to trap specific architectural events. This provides an attacker the opportunity to redirect control flow at critical times without patching any part of a kernel image. Specifically, the contribution of this research includes:

- A generalized approach towards creating PMU-assisted rootkits. This approach is validated with a proof-of-concept Android rootkit targeting the ARM architecture that can covertly intercept SMS messages.
- The concept is extended to the Intel x86-64 architecture with a prototype Windows rootkit that is proven to evade PatchGuard.
- An additional application of performance counters is discussed for detecting and intercepting returns from ARM secure world or returns from x86 SMM.

The rest of the paper is organized as follows. Section 2 offers background information on the PMU for both the ARM and x86 architectures. Section 3 describes the implementation of the rootkit for both architectures tailored towards Android and Windows as well as highlighting several implementation challenges. Section 4 highlights results, limitations, and other applications of this research. Section 5 details prior art in rootkits and other uses of the PMU in security research. Finally, section 6 offers areas for future research and general conclusions to the paper.

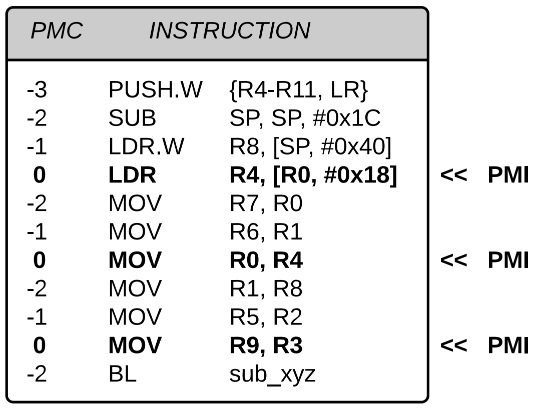
2 BACKGROUND

Nearly all modern CPU architectures contain a performance monitoring unit built into silicon. While features may vary across architectures, they all share a common goal of providing metrics and insight into the performance of the CPU. The foundation of the PMU is built around one or more performance counters (PMC), a set of performance events to be counted, and an interrupt mechanism to signal a counter overflow. For example, an event such as cache misses could be sampled in order to evaluate the performance of the system under test. Numerous commercial products such as DS-5 Development Studio from ARM, VTune Amplifier from Intel, Xcode Instruments from Apple, and Linux perf all leverage the PMU in order to provide engineers real-time feedback to help them diagnose bugs or identify bottlenecks in software. In another example, drivers that act as a CPU governor on many Android based mobile devices are tied directly to a PMU in order to make real-time decisions for conserving power.

2.1 PMU Overview

On both the ARM and Intel x86 architecture, the PMU can be controlled from software. In the x86 architecture this interface consists of a set of model-specific registers (MSR) accessible from ring 0, while on the ARM architecture software control of the PMU can be achieved thru a set of registers behind a system control coprocessor, CP15, or optionally from a memory-mapped interface. Intel's PMU features date back to the original Pentium while the ARM PMU was introduced in ARMv6 and can be found in ARM11, Cortex-R, and Cortex-A cores to include ARMv8 (64-bit). As the PMU is integrated into silicon, it operates transparently to any software running on the CPU. Multi-core processors have a separate PMU for each core, thus each core must be programmed individually.

The PMU's main task is to count events, based on event filters that are set in the event select registers. Each architecture specifies a list of events and filters that can be applied that includes both the event type (e.g. ITLB misses) as well as privilege mode. A complete list of



```
PMC    INSTRUCTION

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

Figure 1: A performance counter is configured to count instructions retired, and the interrupt handler resets the counter to overflow every 3 instructions.

available events can be found in the respective architecture reference manuals [8, 9]. Additionally, performance monitoring interrupts (PMI) can be enabled for each performance counter and will be triggered whenever a performance counter overflows. Whereas one possible use-case of the PMU is to periodically sample a counter based on timing intervals, the interrupt on overflow feature enables one to create a sampling period based on a specific number of events. For example, setting a counter to -3 would result in an interrupt after 3 instances of the selected event being counted by the PMU as shown in Figure 1. The interrupt handler is responsible to reset the counter value to the initial sampling period and the cycle repeats.

Four steps are required to properly utilize the PMU:

1. Register an interrupt service routine to handle performance monitoring interrupts (PMI).

2. Set the interrupt frequency by initializing a performance counter to the desired value.

3. Set the event selection filter, which specifies both the event to count and the privilege mode(s) in which to count.

4. Enable the counter and enable interrupts for the counter.

2.2 The ARM PMU

Technically, the PMU is an optional extension of the ARM architecture, yet it is extremely common on chipsets found in mobile phones and tablets. Nonetheless, on a given ARM device it is first necessary to ensure the performance monitoring extensions are implemented

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

Table 1: Debug settings and PMU version supported on ARM chipsets found in various smartphones and tablets.

and available for use. This can be achieved by querying the DBGAUTHSTATUS debug register in order to obtain the values of four debug signals: DBGEN, NIDEN, SPIDEN, and SPNIDEN. The PMU is considered to be a non-invasive debug feature and thus only requires either NIDEN or DBGEN to be high for counting in normal world, and SPIDEN or SPNIDEN to be high for counting events in secure world. Table 1 lists the debug settings found on several different mobile devices containing chipsets from various vendors.

The ARM reference manual defines a set of architectural performance events as well as numerous optional events. Further, the CP15 C9 register encodings are reserved for the PMU, and since only a subset are utilized it states that additional CP15 c9 encodings can be used to support IMPLEMENTATION DEFINED performance events. As an example, the Krait and Scorpion architectures, as developed by Qualcomm, extend this interface to include optional and custom events. A total of four CP15 c9 performance event selection registers are added in the Krait CPU representing four distinct regions of the CPU. One region for the vector floating-point unit codenamed Venum, and three others for each remaining unit of the CPU. These added events are then encoded based on the region of the CPU, an event code, and a group code. To utilize these Krait specific events, the Krait event selection register is set to the event of interest, and the standard ARM event selection register is set with another code to effectively link the PMU to utilize the appropriate Krait custom register.

Documentation is sparse on this architecture, but information can be gathered from the Android MSM kernel source [10, 11] as well as linux mailing lists [12]. After briefly analyzing an iOS kernelcache, it appears as though Apple is another vendor to extend the ARM PMU specifications in order to add custom performance events to their proprietary ARM-based processors found in iPhones and iPads. However, Apple hardware was not further analyzed as part of this research effort.

3 ROOTKIT DESIGN

The overall concept a PMU-assisted rootkit is to force a hardware interrupt on every single instance of a system call on a given CPU. Let S represent the set of all System Call events, and E the set of all performance events counted from one architecturally supported PMU event code. E is considered a candidate rootkit event if and only if, S is a subset of E, and the absolute complement of S is small. In other words, the goal is to identify a supported PMU event type that ideally counts syscall instructions and ONLY syscall instructions; however, other events may exist that include syscall events as a subset and limited additional events so as to keep performance overhead to a minimum. For example, while the PMU event ALL_RETIRED_BRANCHES might be a superset of all syscalls, the overhead from attempting to trap all branches on the system is too expensive and thus makes this event an unrealistic choice.

Once a candidate rootkit event is identified for the target architecture, a performance counter on each core is configured with an appropriate sampling period and interrupts are enabled for that counter. Typically, the desired sampling period will be -1 (0xFFFFFFFF), which will result in a PMC overflow, and subsequent PMI on every single instance of the event. Malicious code can then be inserted into an interrupt service routine (ISR) that is registered to handle interrupts triggered by the performance monitoring unit. This effectively enables malicious code to be transparently executed for every system call operation, in order for the ISR to inspect and

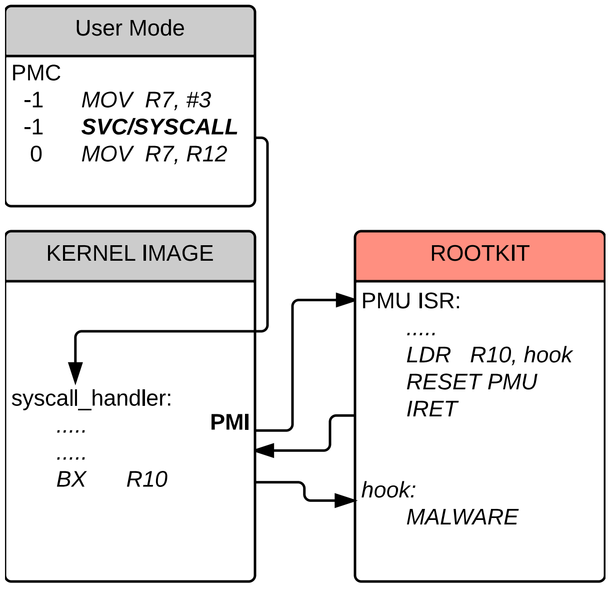


Figure 2: The PMU-assisted rootkit is invoked on counter overflow, and can optionally modify saved registers in order to redirect code execution after servicing the interrupt.

optionally redirect code execution upon returning from the interrupt. Installation of the rootkit only requires initializing one performance counter, and registering the attacker controlled ISR to handle performance interrupts. To perform this behavior system-wide, the malicious ISR is then responsible for clearing the overflow bit from the generated PMI, and to reset the counter back to the sampling period to ensure the next event is also trapped by the PMU. This overall approach is depicted in Figure 2.

As it turns out, favorable events were identified on both the ARM and x86 architectures to capture system calls and exceptions that may be useful to rootkit authors. Since hardware is taking care of interrupting system calls, the rootkit does not need to patch any tables or portions of a kernel image that are often closely guarded by kernel patch protection algorithms. Control of the PMU is OS-agnostic, which means that only the malicious ISR handler is coupled to the OS. Most high-level operating systems such as Android, Linux, and Windows all provide APIs to legitimately register for performance interrupts. In this section, first a prototype rootkit implementation tailored towards Android running on the ARM (Krait) architecture will be introduced followed by a discussion of a similar rootkit tailored towards Windows 7 running on the Intel x86-64 architecture.

3.1 The ARM Rootkit

3.1.1 Finding Inspiration In the ARM Manual

The original idea behind a PMU-based rootkit first occurred while building a dynamic binary instrumentation tool and reviewing the ARM Architecture Reference Manual. In an effort to standardize event codes for vendors that wish to extend required performance events, Appendix C of the manual contains a table with ARM recommendations for *IMPLEMENTATION DEFINED* event numbers [8]. One particular group of events standout, namely the events with an *EXC_* prefix *(EXC_UNDEF, EXC_SVC, EXC_PABORT, EXC_DABORT, EXC_IRQ, EXC_FIQ, EXC_SMC, EXC_HVC)*. These events represent the entire ARM exception vector table (EVT), and combined with the fact a hardware performance counter can be configured to overflow generating an interrupt on every occurrence of an event, the idea for a PMU-assisted rootkit was born.

On the ARM architecture, the supervisor call instruction (SVC) is used to switch to a privileged mode by generating an SVC exception. This instruction was previously termed Software Interrupt SWI. Android, and many other operating systems, utilize this vector in the exception table to handle system calls. As such, the *EXC_SVC* code matches the criteria discussed above as a perfect PMU rootkit event candidate.

The target chipset for a prototype ARM PMU rootkit was the Qualcomm APQ8084 Snapdragon processor found in the Motorola Nexus 6. This quad-core CPU was selected for several reasons. First, Qualcomm remains a leader in both application and baseband processor market share across the globe with their hardware found in more than a billion Android devices [13, 14, 15], and for this reason alone makes their products very attractive for research. Secondly, the APQ8084 CPU houses Qualcomm's custom Krait architecture. As highlighted in section 2, this CPU architecture extends the ARMv7 PMU specifications with a custom implementation that includes support for many additional performance events including counting SVC instructions. Finally, the author is a fan of Qualcomm technology and finds proprietary implementations all the more interesting to research. However, the approach is not unique to this chipset or architecture. Beyond the Krait, Scorpion, and possibly Kryo architectures, the event codes that are leveraged for the Android rootkit in this paper *(EXC_SVC and EXC_SMC)* are documented in both the ARMv7 and ARMv8 manuals [8, 16], are included by default on the cortex-A57 and cortex-A72 architectures [17, 18], and it is possible that other chipset manufactures have added similar functionality in extending their PMU.

Compared to a PMU-assisted rootkit, kernel-mode rootkits that patch portions of a kernel image are significantly easier to implement. All these traditional rootkits must do is override an address of a legitimate function in a syscall dispatch table and replace it with the address of an attacker controlled function. The idea of leveraging the performance monitoring unit to enable malicious code was found to be much harder to implement in practice. The main reason for this stems from the fact the rootkit is relying entirely on the PMU and its interrupt delivery in order to gain code execution. This introduces a number of challenges for successful and reliable syscall hooking via PMU interrupts, which will now be discussed in detail.

3.1.2 IRQ Number Identification

On the ARM architecture, performance monitoring interrupts are typically programmed in the Generic Interrupt Controller (GIC) to be delivered via IRQ. The Android kernel utilizes a radix tree data structure in order invoke registered IRQ handlers based on the IRQ number (IRQn). In order to register for performance interrupts the IRQn must be identified for the target platform, which can be achieved a couple different ways.

Device Tree Source Device tree source and source include files (*.dts* and *.dtsi*) can be found in the kernel source for a given device. These files provide a detailed description of the hardware configuration of different boards and SoCs. In these files it is not uncommon to find the definition for the PMU. For the Nexus 6, the PMU definition can be found in *msm/arch/arm/boot/dts/qcom/apq8084.dtsi*:

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

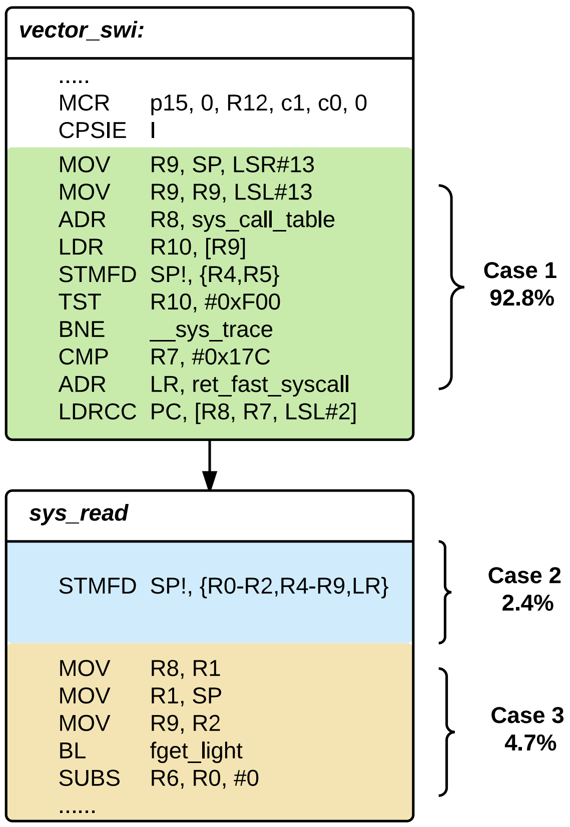
This syntax above describes critical interrupt information about the PMU. The first value of the interrupts tuple denotes it is a Private Peripheral Interrupt (PPI). The middle value represents the interrupt number, and the final value represents the IRQ type and flags. The interrupt tuple can be translated to the IRQn based on the layout of the GIC [19]. The first 16 interrupt ID numbers are considered Software Generated Interrupts (SGI), the next 16 are PPI's, and then an additional 988 values are available for Shared Peripheral Interrupts (SPI). For this device, the IRQn for the PMU = 7 + offset of PPIs (16) = 23. Once this value is obtained, invoking *request_percpu_irq()* will successfully register an ISR handler to the Krait PMU. In cases where this data is accessible from the Android device, it may be feasible to dynamically resolve the PMU IRQn from the rootkit installer.

Brute Force While most application processors contain a performance monitoring unit, not all COTS devices make use of their interrupts. Thus, PMU interrupt definitions are not always found in the device tree source. Such was the case while researching the Amazon Fire HD tablet with MT8135 CPU. For this device a small utility driver was created to help identify PMU IRQ numbers. To achieve this, an ISR handler was registered for every unused PPI and SPI on the system. Next, the PMU was programmed to periodically cause an overflow on a counter. Finally, */proc/interrupts* was analyzed to match the number of overflows forced by the driver with the number of interrupts seen per IRQn. There may exist chipsets that include a PMU, but have its interrupt disabled within the GIC. Identifying and enabling the appropriate interrupt within memory-mapped GIC registers is outside of the scope of this research.

3.1.3 Interrupt Shadow

The biggest challenge to implementing the ARM rootkit was interrupt instruction skid. Instruction skid refers to the number of instructions that are executed past an instruction that triggers a PMC overflow before execution is halted by a PMI. The interrupt shadow refers to these instructions collectively, and any PMU event that occurs within this shadow is missed. This problem extends to both x86 and ARM architectures. The ARM manual [8] sheds light on part of the reason for this effect, *"...the architecture does not define a point in the pipeline where the event counter is incremented."* Fortunately, SVC instructions occur far less frequently than many other event codes, and thus missing a single SVC instruction due to an interrupt shadow was found to be extremely rare. This was verified by checking the counter value in the ISR upon overflow. Normally, upon overflow the counter value will be at zero, if however the counter value is greater than zero then it means an event was missed due to interrupt shadow. The more frequent the interrupts, the more likely to miss trapping an event due to a shadow.

While interrupt shadow did not cause the rootkit to miss system calls, instruction skid still complicated system call hooking. The issue is made more difficult by the fact interrupts are disabled in the SVC exception handler, and then enabled just before accessing the system call dispatch table. Thus, the earliest point in which an SVC operation can be trapped via PMU in the Android kernel is the instruction following CPSIE within *vector_swi*. The landing area is quite small between when interrupts are enabled and a branch to a resolved system call address occurs. Therefore, the rootkit had to deal with three



```
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]

sys_read
    STMFD   SP!, {R0-R2,R4-R9,LR}

    MOV     R8, R1
    MOV     R1, SP
    MOV     R9, R2
    BL      fget_light
    SUBS    R6, R0, #0
    ......
```

Figure 3: The frequency associated with three possible values of PC upon trapping SVC instruction via PMU and isolating the read system call.

different cases. Upon the SVC instruction being trapped by the PMU on a *read* system call for example, the saved program counter (PC) was either 1) within *vector_swi* after the CPSIE instruction, 2) PC equals the address of the entry point of *sys_read*, or 3) PC is somewhere past the *sys_read* entry point.

The rootkit was developed to accommodate all three of these cases. The ISR first retrieved the saved user mode and SVC mode registers. The user mode registers were used to filter trapped SVCs by inspecting the syscall number in R7. The first two cases were straightforward to implement by emulating any remaining instructions within *vector_swi* and changing the saved PC register to the address of a malicious function. As depicted in figure 3, these two cases represent roughly 95% of 3 million trapped *read* system calls. The third case was handled by overwriting a return address on the stack, and allowing the system call routine to finish. This was acceptable since all hook points selected involved data being retrieved from the kernel. Specifically, the ISR walks up the saved stack pointer in search of the address of *ret_fast_syscall*, and modifies it to a trampoline address. Trampoline code restores the Link Register (LR) with *ret_fast_syscall* before branching to malicious code. This malicious function takes the return value as its only parameter, already stored in R0, and then retrieves the original function parameters from the saved user mode registers in order to copy buffers to and from kernel space to perform any desired modifications to data.

3.1.4 CPU Hot-Plugging

One final challenge on the Nexus 6, and presumably many other Android devices, was the presence of a CPU hot-plug feature for multi-core systems. This feature allows individual CPU cores to be brought online based on demand in an effort to conserve power. The hot-plugging feature is present by default on Android 5.0 on the Nexus 6. Performance monitoring unit registers do not persist when a core goes offline. The malicious ISR is still registered within the OS to handle interrupts, but the counter for a core coming online will no longer be set appropriately to count SVC instructions. Fortunately, the Linux kernel provides an easy interface, *register_hotcpu_notifier*, which will be invoked whenever a core changes state. No action is needed when a core goes offline, but the rootkit uses this callback to watch for the CPU_STARTING event and then initialize the PMU on that core to count and trap all SVC instructions. With this implementation, the rootkit was able to persist through CPU hot-plugging.

3.1.5 The PMU-Assisted Android Rootkit

The prototype Android rootkit was successfully created and tested on a Nexus 6 running Android Lollipop 5.0, build number LRX210. Since the main objective of a rootkit is to enable covert operations and to hide indicators of compromise, the system calls *getdents64* and *read* were selected for the proof-of-concept solution. The *sys_getdents64* function was selected in order to provide basic file hiding and process hiding on Android. To explore a more phone specific kernel malware feature, the *sys_read* system call was intercepted and examined only in the context of *qmuxd*. The *qmuxd* daemon on Qualcomm powered Android devices is responsible for managing communication sessions with the cellular modem. This daemon passes along data encapsulated in a proprietary QMI protocol, between the cellular baseband and the appropriate clients running within Android. By hooking *sys_read* within the context of *qmuxd* threads, it enabled the rootkit to intercept QMI packets being sent from the modem towards Android. As an example use-case of this hook, the rootkit inspects QMI packets and filters on the Wireless Messaging and Voice QMI services in order to intercept and modify incoming SMS messages, USSD messages, or incoming phone call notifications. While there are all kinds of interesting hooks and operations that could be conducted on a phone, one could envision this particular hook being utilized for covert communication, by preventing certain QMI packets from ever reaching higher-level services running in Android.

3.2 The x86 Rootkit

Having successfully proven a kernel-level rootkit for Android using the ARM performance monitoring unit, attention was turned towards the Intel x86 architecture to explore whether Intel's performance monitoring unit could be abused in a similar manner. Specifically, the focus was on the x86-64 architecture since PatchGuard is only supported on 64-bit versions of Microsoft Windows. Initial experimentation was performed on Linux Ubuntu 14.04.3 x64 for ease of research and development, and then later tested on Windows 7 SP1 64-bit. Research was performed on a Dell laptop with Intel core i3-4030U, and a MacBook Pro with Intel core i7, both based on the Haswell microarchitecture.

Identifying a candidate PMU event type on the x86 architecture was a little more challenging. Unlike the ARM architecture, which offers counting SVC instructions, no obvious event was identified for counting exclusively SYSCALL instructions on x86-64. Coming up with a solution for x86 required first dissecting the operations involved in Intel's SYSCALL instruction. From a high level, the SYSCALL instruction switches from ring-3 to ring-0 while branching to a system call handler in the kernel. More specifically, the instruction leverages three separate MSR's: the target code segment (CS) and stack segments (SS) are loaded from IA32_START, the address of the OS syscall handler is loaded into RIP from IA32_LSTAR, and the RFLAGS register is updated based on the complement of IA32_FMASK.

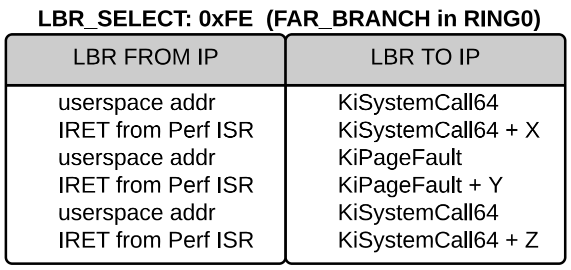
One architectural performance event available on Intel processors is the Branch Instructions Retired (BR_INST_RETIRED) event. Dating back to at least the 3rd generation of Intel Core an interesting conditional unit-mask exists, namely the number of far branches taken (BR_INST_RETIRED.FAR_BRANCH). This allows the PMU to count all retired far branches, and is of particular interest since the SYSCALL instruction essentially makes a far branch into the kernel system call handler, such as *KiSystemCall64* for Windows. Two different Intel x86 event codes were found capable of counting far branches, which both satisfy the criteria as a candidate rootkit event. The first is the BR_INST_RETIRED.FAR_BRANCH event. Since the SYSCALL instruction takes a far jump towards ring 0 kernel code, branches targeting user mode code can be ignored. This can be controlled by setting the appropriate mode inclusion bit in the IA32_PERFEVTSELx MSR. A second event code was identified capable of counting the same far branches as long as Intel's Last Branch Recording (LBR) was supported. The event code ROB_MISC_EVENTS.LBR_INSERTED will count every branch that is inserted into the LBR. Combining this with an LBR filter that only logs far branches occurring in ring 0 will result in identical results to the first event code. These two PMU configurations are displayed in figure 4.

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

Figure 4: Two Intel x86-64 PMU configurations capable of counting Far branches including the SYSCALL instruction

Either of these event codes can be used to count every single far branch destined for kernel code. However, setting the counter sampling period to -1, as was performed on the ARM architecture, was found to result in an interrupt loop. The cause of the interrupt loop is the iretq instruction that occurs upon completing service of the performance monitoring interrupt. This particular iretq instruction is returning to the kernel code that was interrupted by PMI, and since the iretq itself is a far branch, the PMC overflows and triggers another PMI causing a loop. Thus, trapping every far branch in the kernel on x86 requires resetting the performance counter to -2 from the ISR. This behavior is captured in figure 5.

Last branch recording is a convenient feature of Intel x86 hardware that enables filtering and recording specific types of branches. Retrieving the TO address from the top of the LBR stack while filtering on far branches is the easiest way to determine whether the current PMI was triggered on a SYSCALL. By comparing this address with the address from IA32_LSTAR, the rootkit could quickly determine whether a system call was the culprit of the PMI. For environments where the LBR is not accessible, such as within a virtualized guest OS, the rootkit was implemented to inspect RIP from the KTRAP_FRAME passed into the ISR in order to determine whether the address falls within *KiSystemCall64*. Using this methodology, the Windows 7 rootkit was developed and tested on physical hardware as well as within a VMWare guest OS.



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

Figure 5: A generalized LBR snapshot when logging only Far branches in ring 0

Since the SYSCALL instruction masks interrupts, a delayed interrupt instruction skid was observed in Windows as was the case on the ARM architecture. In spite of this, implementation of the Windows PMU-based rootkit was significantly easier than Android, given that many of the challenges such as IRQ number identification and CPU hot-plugging did not apply. The primary hurdle was dealing with the instruction skid occurring after the *sti* instruction within *KiSystemCall64*. However, the problem was much easier on Windows as the landing area between enabling interrupts and branching to the resolved syscall routine is much larger in Windows *KiSystemCall64* compared to the Android kernel. In fact, in one experiment more than 32 million syscalls were trapped by the Windows rootkit, and more than 99.9999% of them were interrupted prior to jumping into the syscall routine retrieved from the System Service Descriptor Table (SSDT). For this reason, cases 2 and 3 from the Android rootkit were ignored, making the Windows rootkit implementation much simpler.

Using these PMU settings, a Windows rootkit was developed by trapping far branches and redirecting system calls on a Windows 7 64-bit kernel protected by PatchGuard. The Windows system call *NtEnumerateValueKey* was chosen as one hook in order to hide registry settings, and was able to successfully cloak registry values without tripping PatchGuard. Additionally, the Windows system call *NtCreateFile* was hooked in order to monitor files. SYSCALL instructions were found to represent 68% of far branches trapped, and another 25% of trapped events were far branches to *KiPageFault*. The far branch event remains a solid candidate for rootkits as the performance overhead is still quite low, and interestingly this approach may extend to intercepting other entries of the IDT since most of the additional far branches caught were page faults or other hardware interrupts.

4 ANALYSIS

4.1 Overhead and Limitations

Efficacy testing was performed on the Android rootkit by validating PMU hooks were able to cloak threads from a process listing with the *ps* command, to hide a file on the file system while performing a directory listing with the *ls* command, and validating the ability to intercept and modify incoming SMS and USSD strings displayed to the user. On Windows, validation was performed by ensuring registry values were not visible from regedit, as well as ensuring PatchGuard did not trigger a BSOD for detecting any modifications to critical kernel code or data.

Performance overhead was studied on both architectures using the PassMark benchmarking tool [20] for Android and Windows. Additionally, the Dromaeo JavaScript benchmarking tool [21] was utilized as a secondary metric. In total, trapping system-wide SVC instructions via performance interrupts added 2% overhead to the Nexus 6 device. In most categories of PassMark for Android overhead was negligible, but the Memory tests had a 6% overhead likely due to hooking read. On x86, trapping all far branches in ring 0 added 8% overhead to Linux, and 10% overhead to Windows 7.

Existing KPP implementations ignore the performance monitoring unit and legitimately registered ISRs, thus the PMU rootkit was able to evade current integrity checks including from Microsoft's PatchGuard. However, detection of abuse of the performance monitoring unit is rather easy to perform. First, any kernel-level or higher privileged code is able to read and interpret PMU register settings. This could easily be integrated into anomaly detection software or KPP to periodically inspect PMU registers in search for abuse. A major increase in interrupt frequency is also evident when abusing the performance monitoring unit. The ability to detect an increase or even presence of PMI is another approach to detecting these rootkits. However, abusing the PMU is a double edged sword, and while this paper focuses on the offensive use-case of hardware performance counters, there are many more practical applications in the defensive realm. Thus, defenders must keep this mind before treating any PMU configuration anomalies as malicious.

As with any approach, there are limitations to a PMU-assisted rootkit. First, the rootkit provides no form of persistence. PMU registers do not persist across core resets, and registering an ISR for handling interrupts must be done upon each boot. Secondly, the approach will miss any event that occurs while interrupts are disabled. This did not impact reliability of intercepting system calls, but may apply to other hardware-assisted applications that utilize the PMU. Finally, the PMU-based rootkit is fragile. Any other code running with kernel-level privileges is capable or reading, writing, or tampering with the performance monitoring registers. Disabling an interrupt enable bit, changing an event code filter, or setting the counter value could prevent the rootkit from continuing to gain code execution on each system call. Thus, it may be necessary in a real world implementation to have a watchdog thread as part of the rootkit to detect PMU register tampering and reset these values for the rootkit to continue operation. Interestingly, the ARM architecture does support trapping access to the PMU to hypervisor mode, offering additional stealth opportunities provided an attacker has control of the hyp vector table.

4.2 Interactions with SMM and TrustZone

An interesting artifact of this research was observed once it appeared that various performance events were capable of trapping returns from System Management Mode on x86, and depending upon debug settings, trap returns from secure world on ARM. Potential applications for this are left to the reader's imagination, but knowledge of the periodicity of SMM and secure world switches could be beneficial to a rootkit especially if KPP is being performed while in these modes. In the case of ARM, there may exist a use-case for a rootkit to modify or capture data being returned from a particular Secure Monitor Call (SMC).

This topic was first investigated on ARM with the obvious *EXC_SMC* event code. Using this code all SMC calls could be counted triggering an interrupt on overflow. Even on systems without this specific event code, SMC instructions can often be counted by using the appropriate mode exclusion bits. For example, event code 0x3800000C will count all branches in secure PL0, PL1, or HYP modes. Since the SMC instruction itself begins execution in normal world, but transfers to secure monitor mode, the SPNIDEN debug signal may not be required to count these instructions. Moreover, a MediaTek chipset in the Amazon Fire, listed in Table 1, was found to support counting any event in secure world due to its debug settings. As a result, the PMU could be configured to trigger a PMI while in secure world resulting in the interrupt being serviced on return from an SMC call or a secure FIQ.

ARM recommends the FIQ to be reserved for secure world and IRQ for normal world. Because a secure FIQ is programmed with a much higher priority, even though an overflow of a performance counter would occur immediately upon running in secure world, the PMI will not be serviced until the FIQ is completed. Depending upon a vendors TrustZone implementation, there may exist denial-of-service attacks on an SMC call by causing an SMC handler to immediately be forced to return to normal world to service a PMU interrupt. It is thus important for a TrustZone implementation to ensure it first masks interrupts within the SMC vector handler. This is a possible area of future research, as further analysis was not performed as part of this effort.

On the x86 architecture, monitoring far branches also appeared to catch returns from SMM mode. This was especially apparent on hardware with LBR support. One convenient feature provided by Intel's PMU is the ability to freeze counting as well as freezing the LBR whenever a performance counter overflows. These two settings are controlled within the IA32_DEBUGCTL MSR. On the Haswell CPU inside the Dell Inspiron laptop, every return from SMM resulted in a few observable behaviors. Whenever the PMU ISR was invoked after executing in SMM, neither the LBR nor the counter was frozen. Because of this, the LBR top of stack (TOS) was always a branch to *hal!HalpPerfInterrupt* (IDT vector 0xFE). Additionally, inspecting the far branch prior to the TOS entry exposed a matching TO and FROM address. Finally, reading the far branch counter value revealed a positive count, whereas a normal far branch trap results in a counter value of zero due to the PMC freeze setting. All of these anomalies were found to reliably signal a return from SMM, further validated by noting an increment to the MSR_SMI_COUNT. There do exist errata for some Intel Xeon processors, which mention LBR data could be incorrect after return from SMM [22, 23]. Regardless of the exact cause and sequence of events, it seems that trapping far branches via PMU on this particular Haswell CPU is able to reliably detect and trap returns from SMM.

5 RELATED WORK

A handful of rootkit designs have been introduced targeting the ARM architecture in prior research. A Phrack article [24] covers a number of traditional approaches for Android based kernel-rootkits ranging from syscall table modifications via */dev/kmem*, to modifying the SVC/SWI exception vector to branch to an attacker controlled syscall handler. Cloaker [25] leverages the System Control Register (SCTLR) on the ARM architecture in order to move the exception vector table from high address to low address in order to map a malicious EVT at address 0x0 in order to intercept all exceptions. Rather than patching syscall table entries, Suterusu [26] performs inline hot patching of kernel functions on both the x86 and ARM architectures. Clock Locking Beats [27] explores the feasibility of hiding code by implementing a custom CPU governor capable of overclocking in order to add cycles for malware to run. Roth [7] introduces a Trust-

Zone based rootkit, which redirects IRQ interrupts, normally utilized by normal world, to secure monitor mode allowing TZ based malware to intercept.

Rootkits targeting the x86 architecture have been very well documented in prior literature including countless Phrack articles and books [28]. Beyond traditional rootkit patching techniques, Shadow Walker [29] added novel stealth techniques by hiding code pages after hooking the page fault handler. The Subversive rootkit [30] leverages x86 debug registers in order to trap and intercept system calls of interest. The Blue Pill rootkit [4] utilizes x86 virtualization extensions in order to create a hypervisor running underneath the target operating system, thus able to intercept interrupts among many other things. An SMM based keylogger [6] was introduced that patches the SMI handler in BIOS firmware, and a GPU based keylogger [31] was demonstrated that uses DMA to capture keystrokes.

The performance monitoring unit is the subject of several studies focused on hardware-assisted security and instrumentation. Most if not all of these studies have been focused on the Intel x86 PMU, with very little discussions on the ARM PMU. Vogl et al. [32] discusses using the PMU on the x86 architecture in order to conduct instruction level monitoring by trapping performance events such as branches or instructions to a hypervisor for monitoring. Several studies focus on utilizing the PMU for return-oriented-programing (ROP) detection. Wicherski [33] uses the PMU to trap mispredicted RET instructions occurring in Ring 0 in order to detect ROP activity within the kernel, and Li et al. [34] extends this approach to check for stack pivoting on similar mispredicted return branches. CFIMon [35] explores using the PMU and Branch Trace Store (BTS) on Intel architecture to apply control-flow integrity on branches. Meanwhile, NumChecker [36] uses performance counters to detect rootkits by measuring event frequency during system calls.

6 CONCLUSION

In this paper, a novel kernel-level rootkit is discussed and demonstrated on the ARM and Intel x86 architectures. On the ARM architecture, SVC instructions are trapped by configuring the performance monitoring unit to count and overflow on every SVC instruction. A prototype Android rootkit tailored towards the Krait architecture is implemented capable of hiding files, processes, and SMS messages. Similarly, by leveraging the far branch event supported on the Intel x86 architecture, a proof of concept solution is demonstrated in Windows 7 64-bit capable of hiding registry entries. Further, utilizing the PMU does not require patching or modifying any critical kernel structures and thus allows it to evade Kernel Patch Protection algorithms such as Microsoft PatchGuard. Performance overhead is shown to be low on Android, and at an acceptable level on Windows and Linux. Further, it was shown how the same approach can be used to trap returns from secure world on the ARM architecture, as well as returns from System Management Mode on the x86 architecture.

Several areas exist for potential future work in researching offensive applications of the performance monitoring unit. First, extending the solution to ARMv8 processors with their built-in support for counting SVC instructions on some Cortex-A cores is one area of future research. Additionally, exploring Apple and other custom ARM system-on-a-chip solutions in search of support for interesting PMU features. On the x86 architecture, further research is needed to better understand all events and interrupts that can be caught monitoring far branches. For example, in testing the prototype Windows PMU driver is able to trap and redirect code execution on debug exceptions prior to a breakpoint being hit opening up possibilities for anti-debugging features. Further, it was noted that capturing page faults was the second most common event beyond SYSCALL on x86. Thus, exploring the feasibility of Shadow Walker type solutions is another potential area of future research. Intel SGX is an emerging area of research that could also present interesting interactions with the performance monitoring unit.

7 Acknowledgments

A number of individuals need to be recognized and thanked for enabling this research to occur. Thus, a special thank you is given to Cody Pierce of Endgame, for his support, for educating me on the Windows Kernel, and for encouraging me to publish this research; to Kenny Fitch of Endgame, who was instrumental in enabling the Windows research by identifying how to register for PMU interrupts on Windows without upsetting PatchGuard; to Eric Miller of Endgame, for his support, sharing devices, and providing feedback to ideas; to Jamie Butler of Endgame, for educating me on some Windows and x86 rootkit techniques as well as reviewing this paper; and last and most importantly, thank you to my wife and kids for their love and support during what probably seemed like my working two jobs.

References

[1] Skywing, “PatchGuard Reloaded: A Brief Analysis of PatchGuard Version 3,” *Uninformed*, vol. 8, 2007.

[2] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, “Hypervision

across worlds: Real-time kernel protection from the ARM TrustZone secure world," in *ACM CCS '14*, pp. 90–102, 2014.

[3] T. Pangu, "Hacking from iOS8 to iOS9." http://blog.pangu.io/wp-content/uploads/2015/11/POC2015_RUXCON2015.pdf. Ruxcon 2015.

[4] J. Rutkowska, "Subverting Vista Kernel For Fun And Profit." https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf. BlackHat 2006.

[5] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, "SubVirt: Implementing Malware with Virtual Machines," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pp. 314–327, IEEE Computer Society, 2006.

[6] F. Wecherowski, "A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers," *Phrack, Issue 66*, 2009.

[7] T. Roth, "Next Generation Mobile Rootkits." Hack In Paris 2013.

[8] ARM, *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*.

[9] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C, 3D): System Programming Guide*.

[10] https://us.codeaurora.org/cgit/quic/la/kernel/msm/tree/arch/arm/kernel/perf_event_msm.c?h=msm-3.4#n278.

[11] https://android.googlesource.com/kernel/msm/+/android-6.0.1_r0.61/arch/arm/kernel/perf_event_msm_krait.c.

[12] "ARM: perf: Add support for Scorpion PMUs." https://lkml.org/lkml/2015/2/11/654. Linux Kernel Mailing List, 2015.

[13] https://developer.qualcomm.com/get-started/android-development.

[14] "Qualcomm Retains Lion's Share Of LTE Baseband Market; Further Gains Expected In 2016." http://www.forbes.com/sites/greatspeculations/2016/02/24/qualcomm-retains-lions-share-of-lte-baseband-market-further-gains-expected-in-2016/.

[15] "Strategy Analytics: Smartphone Apps Processor Revenue Declined 4 Percent in 2015 to Reach $20.1 Billion." http://www.prnewswire.com/news-releases/strategy-analytics-smartphone-apps-processor-revenue-declined-4-percent-in-2015-to-reach-201-billion-300221407.html.

[16] ARM, *ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile*.

[17] ARM, *ARM Cortex-A57 MPCore Processor: Technical Reference Manual*.

[18] ARM, *ARM Cortex-A72 MPCore Processor: Technical Reference Manual*.

[19] ARM, *ARM Generic Interrupt Controller Architecture Specification: GIC architecture version 3.0 and version 4.0*.

[20] PassMark PerformanceTest Mobile. http://www.passmark.com/products/pt_mobile.htm.

[21] Dromaeo JavaScript Performance Testing. http://dromaeo.com/.

[22] Intel, *Intel Xeon Processor E5 v2 Product Family, Specification Update, September 2015*.

[23] Intel, *Intel Xeon Processor E3-1200 Product Family, Specification Update, August 2015*.

[24] dong-hoon you, "Android platform based linux kernel rootkit," *Phrack, Issue 68*, 2011.

[25] 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.

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

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

[28] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[29] S. Sparks and J. Butler, "SHADOW WALKER: Raising The Bar For Rootkit Detection." BlackHat Japan 2005.

[30] falken, "Subversive rootkit." https://github.com/falk3n/subversive.

[31] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "You Can Type, but You Can't Hide: a Stealthy GPU-based Keylogger," in *EuroSec*, 2013.

[32] 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.

[33] G. Wicherski, “Taming ROP on Sandy Bridge: Using Performance Counters to Detect Kernel Return-Oriented-Programming.” SyScan 2013.

[34] 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.

[35] 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.

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