Revisiting the ARM Debug Facility for OS Kernel Security

Jinsoo Jang
KAIST
jisjang@kaist.ac.kr

Brent Byunghoon Kang
KAIST
brentkang@kaist.ac.kr

ABSTRACT

Hardware debugging facilities, such as watchpoints, have been used for software development and analysis. In this paper, we expanded the use of watchpoints as a hardware security primitive for enhancing the runtime security of mobile devices. By analyzing the watchpoints in detail, we derived useful watchpoint properties that can be exploited to build security applications. Based on our analysis, we designed example applications for hardening the OS kernel by exploiting watchpoints. The proposed applications were implemented on a Juno development board with 64-bit ARM architecture (ARMv8). Hardening the kernel by fully enabling the proposed schemes was found to impose reasonable overhead, i.e., 3% with SPEC CPU2006.

1 INTRODUCTION

In general, modern electronic devices, such as mobile phones and PCs, support hardware debugging features (e.g., breakpoints and watchpoints). These features are defined during the system-on-chip (SoC) design process, and once integrated, are used for developing low-level software (e.g., firmware and bootloaders) as part of manufacturing the final product. Debugging features are also available at device runtime, and have been used to facilitate software development (e.g., GDB), to analyze malware [24], and to find critical bugs that cause race conditions [9, 16]. Thus, previous studies have capitalized on debugging features for software development and analysis.

In this paper, we describe the leveraging of hardware debugging features from a different aspect. We first analyze in detail a debugging facility, specifically a watchpoint on the ARM architecture, and abstract useful security features such as privileged-aware monitoring. We then show how to leverage these features to enhance mobile OS security.

As example security applications, we designed a watchpoint-based kernel execute-only memory (XOM) and privileged access never (PAN) attributes. XOM [12] aims to prevent the leakage of a code location and thus hinder a code reuse attack. The 64-bit ARM architecture supports this feature only for user memory; the user-level XOM can be enabled through the configuration of the page table attribute. To activate this feature in the kernel space as well, we utilize watchpoints, which do not incur any exception for an instruction fetch from a watchpoint-monitored region. In particular, kernel-level XOM is emulated by configuring the watchpoint such that read access to the kernel text region generates a watchpoint exception.

PAN is a security feature that prevents a kernel bug from arbitrarily reading and using a potentially malicious user memory. In the ARM architecture, PAN is limitedly supported by high-end processors. We designed PAN by exploiting the following watchpoint properties: (1) the watchpoint can be configured to monitor privileged access only, and (2) is sensitive to an unprivileged instruction execution. Specifically, we set the watchpoint monitoring to trap only the kernel-privileged access to the user space during kernel execution. Then, to ensure that legitimate kernel functions, such as copy_from_user, can seamlessly access the user memory, we patch such functions with unprivileged load and store instructions that never generate watchpoint exceptions against the region set with kernel-privileged access monitoring.

We implemented our prototype watchpoint-based kernel protection mechanisms on a Juno development board [6]. The exception vector and macro for accessing the user space were patched to implant the watchpoint configuration operations and use the unprivileged instructions, respectively. During the performance evaluation, we combined each solution including the existing Linux security feature (i.e., TTBR-based PAN) and measured the performance of each case. We observed an overhead of less than 3% using SPEC CPU2006.

2 RELATED WORK

Mobile device security. By using various features of ARM architecture, novel approaches have been proposed to secure mobile devices. The privileged access never (PAN) and privileged execute never (PXN) flags [1] have been leveraged by Linux to restrict privileged access to and execution of the user memory, respectively. Sentry [17] takes advantage of the IRAM and cache to protect the device from DRAM attacks, such as cold boot attacks. The memory domain and the domain access control register (DACR) are leveraged by ARMlock [29] and Shreds [14] to enable hardware-based fault isolation and realize the in-process memory isolation, respectively. User-level execute only memory (XOM) for mobile devices was developed using page permission manipulation [15] and a software-based load-masking technique [13]. ARM TrustZone, a security extension of the processor, has been intensively explored for critical application logic protection [22], communication channel security [21], and OS kernel attack [23] and integrity protection [11]. The hardware-assisted virtualization has also been used for device security enhancement: OS kernel protection [7] and application shielding [20]. Finally, hardware security primitives such as the physical unclonable function (PUF) have been leveraged to prevent hardware-based attacks and to establish the root of trust of the device [26]. In our work, we use hardware watchpoints for the

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317897

Table 1: Watchpoint control register flag setting for monitoring a particular mode in the non-secure and secure states.

SSC	PAC	Security state	Watchpoint for
01	10	Non-secure	User
01	01	Non-secure	Kernel
10	10	Secure	User
10	01	Secure	Kernel

creation of useful security applications, such as kernel-level XOM and PAN emulation.

3 WATCHPOINT ANATOMY FOR SECURITY

In this section, we provide a brief introduction to watchpoints, a self-hosted monitoring feature. Further, watchpoint aspects that are useful for building security applications for mobile device runtime protection are explored.

3.1 ARM privilege and security model

Before we discuss watchpoints, we briefly introduce the ARM security states and privilege model. Modern ARM processors support the TrustZone [1] security extension, which separates the processor security state into non-secure and secure states. Each state can be entered by configuring the non-secure (NS) flag in the SCR_EL3 register. For example, clearing and setting the NS flag indicates that the current CPU security state is secure and non-secure, respectively. The user, kernel, and monitor modes exist in the secure state. By contrast, the user, kernel, hypervisor, and monitor modes are available in the non-secure state. Self-hosted debugging is supported for all modes in both states except for the monitor mode, which only supports external debugging.

3.2 Watchpoint exception routing

By default, the debug exceptions are routed to and handled by the kernel. In addition, to generate the exceptions in the same mode that handles the exceptions, the kernel debug enable (KDE) flag in MDSCR_EL1 must be set and the debug exception mask flag (D) in the process state (PSTATE) must be cleared. For example, to generate and handle the debug exceptions in the kernel mode in the non-secure state, the following configuration should be adopted: MDSCR_EL1.KDE=1, SCR_EL3.NS=1 and PSTATE.D=0.

3.3 Watchpoint privilege and type

The mode privilege where the exception can be generated is defined by the combination of the security state control (SSC) and the privilege of access control (PAC) flags in the watchpoint control register (DBGWCR). Except for the monitor mode, any mode in both security states can generate the exceptions depending on the configuration. Table 1 describes the example combination of flags for generating the debug exceptions explicitly in a single mode in the non-secure and the secure states. Besides, the load store control (LSC) flag in DBGWCR, which is two bits in size, determines the type of monitoring as read (0b01), write (0b10), or both (0b11).

3.4 Useful security properties

Here, we describe the useful features of watchpoints that inspired us to design the kernel security applications using watchpoints. The properties stated below can be leveraged to build security applications.

(P1) Privilege-aware monitoring. The watchpoint exception is privilege-sensitive. For example, even if a certain address range is accessible from the user and kernel, we can configure the watchpoint such that only kernel-privileged access generates the exceptions.

(P2) Compatibility with unprivileged operations. In association with P1, an unprivileged instruction execution recognizes the privilege setting of the watchpoint-monitored area. For example, the execution of unprivileged load (e.g., *ldtr* instruction on ARM) against a certain region that is configured with kernel-privileged access monitoring does not generate any exception.

(P3) Instruction fetch agnostic monitoring. Because the watchpoint is a type of data breakpoint, instruction fetch from the watchpoint-monitored region does not incur any exception, which is suitable for implementing execute only memory (XOM).

4 OS KERNEL HARDENING

We propose two watchpoint utilizations for enhancing OS kernel security (Figure 1): (1) emulation of PAN and (2) enabling kernel-level XOM. In this section, we briefly present the motivation, design, and limitations of our approaches.

4.1 PAN emulation

PAN is a hardware-supported security feature that aims to restrict access to user memory from the kernel. On ARM, this security feature is available from ARMv8.1 [1], but at the time of writing this paper, most commercial processors do not support this feature; only Cortex-A75 and Cortex-A55, two out of 14 available commercial Cortex-A processors, solely provides this feature [3].

To support this security feature for legacy devices, Linux emulates its behavior [4]. For example, on ARMv8 (64-bit ARM architecture), Linux forces the translation table base register for the user space (TTBR0) to point to an invalid page table during kernel execution. This makes the entire user space memory inaccessible to the kernel. However, to support kernel operations that legitimately access the user memory (e.g., `copy_from_user`), the TTBR0 is temporarily remapped to the valid page table address, making the entire user space accessible again. This property could be a potential vulnerability because a kernel bug could be exploited when the protection is disabled.

Watchpoint-based PAN. We can ensure seamless PAN enforcement by using watchpoints, the protection of which is never deactivated during the kernel mode execution. The combination of the watchpoint properties *P1* and *P2* enables this feature. In particular, when the processor switches to the kernel mode, we configure the watchpoints such that they monitor any access (read/write) to the user space with kernel privilege (*P1*). For legitimate operation support, we replace the load (LDR) and store (STR) instructions with unprivileged load (LDTR) and store (STTR) instructions in the kernel functions that access the user space. Because only the

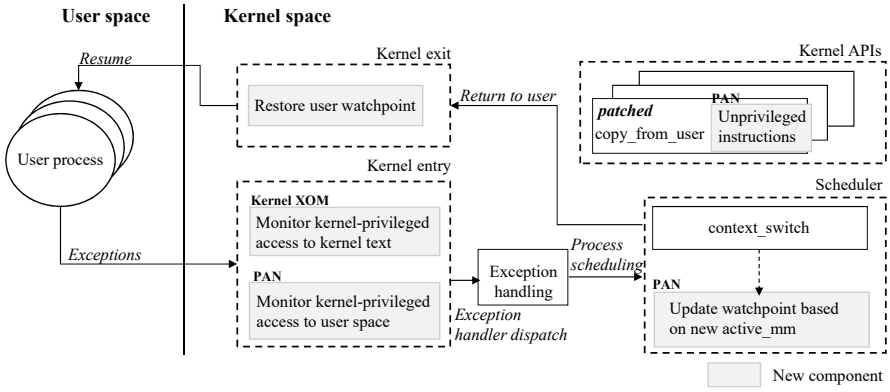

Figure 1: Watchpoint-based kernel security applications. For kernel XOM, the watchpoints are configured in kernel entry to monitor kernel-privileged access to the kernel text. PAN emulation requires the watchpoints to be updated twice, i.e., in kernel entry and scheduling. PAN configures the watchpoints on the basis of the current process active_mm to monitor kernel-privileged access to the user space. Further, kernel functions that directly access the user spaces are patched to use unprivileged load and store instructions for compatibility with PAN enforcement.

kernel-privileged access to the user space is monitored, the unprivileged instructions do not incur any watchpoint exceptions (*P2*). Cache maintenance instructions executed for the user memory do not generate exceptions as well. This is an additional benefit of our approach compared to TTBR0-based PAN. Because the cache on ARMv8 is physically indexed physically tagged (PIPT) or virtually indexed physically tagged (VIPT) [2], performing the cache operations requires TTBR0 to be properly (re)mapped.

The monitoring area is defined by referring to the virtual memory area (VMA) of each process. We traverse `mmap` in the `active_mm` kernel data structure to get all the VMAs for the current process. The pairs of `vm_start` and `vm_end` are members of `mmap` and indicate the start and end addresses of each VMA. They are stored in the linked list in ascending order, and all of them need to be monitored for PAN enforcement. To minimize the number of watchpoints required for the monitoring, we configure the watchpoints with maximum size of monitoring, which is 2 GB. Note that the virtual address width is 39 bits in our Linux system (e.g., 0X0 - 0x7FFFFFFFFF for user space). Hence, we generate the first watchpoint monitoring address value by masking (i.e., AND operation) the first `vm_start` with 0x7F80000000 to make the monitoring address aligned with the monitoring size, which is a watchpoint setup requirement [1]. The watchpoint control register value is also generated with the 2 GB setting. We keep traversing the pairs until we get to the `vm_start` that does not fall within the 2 GB area. At this point, we create new watchpoint monitoring address and watchpoint control values in the same manner. We repeat this procedure until no pairs are left to be traversed.

The aforementioned procedure is conducted whenever a new process is scheduled to reflect the current `active_mm` update. The generated address and control values are not only set in watchpoint registers but also stored in our *wp_pan* data structure array added in the `thread_info` kernel structure. *wp_pan* defines 32-bit and 64-bit long variables to store the generated values of watchpoint control (DBGWCR) and address (DBGWVR), respectively. Then, the values in *wp_pan* are used when the CPU mode switches from user to kernel in order to reactivate PAN. In other words, the watchpoint registers are configured twice for kernel entry and process scheduling. Further, note that although the kernel thread does not access the user space, it borrows the previous process user space mapping as its own user memory map. Hence, PAN enforcement based on `active_mm` is still effective for the kernel threads.

4.2 Kernel XOM

XOM was introduced to overcome code memory disclosure attacks [27] by forcing a certain memory region to be either readable or executable, but not both. Toward this end, previous works have used hardware features, such as memory paging. For instance, XnR [12] emulates XOM by managing the page faults on user memory pages that are intentionally set as non-present. Norax [15] uses the access permission bit in page table entry to force the user memory to be only executable and not readable. On x86, the extended page table (EPT) was exploited to enable XOM by removing read permission from certain memory pages of virtual machines [18, 28].

However, the techniques used in these works are not sufficient for enabling kernel-level XOM on ARM for the following reasons: (1) XOM emulation might cause frequent kernel page faults and thus incur an extremely high performance overhead. (2) Removing read permission of kernel pages causes a prefetch abort exception on ARM architecture [1] regardless of whether the page is mapped by the OS page table or EPT; hence, paging nuances cannot be used for supporting kernel-level XOM on ARM.

Software-only approaches have also been introduced. LR2 [13] demonstrates that XOM can be created by splitting the virtual memory into two halves and masking the address bit to isolate the read operation in one half (i.e., data). However, this approach might require significant changes in the OS kernel layout. kR^X [25] implements the kernel XOM by using SFI and Intel MPX [5], which is only available on x86.

Watchpoint-based kernel XOM. As discussed in Section 3.4, an instruction fetched from the memory region that is monitored by the watchpoint does not incur any exceptions (*P3*). This property

is beneficial when we create execute-only memory for the kernel space. We just need to configure a single watchpoint to monitor the kernel text area. The watchpoint value register is set to the starting address of the kernel text. Read access monitoring is sufficient for XOM, but write access can also be set for kernel integrity protection. Configuration and activation of the watchpoint are conducted when the CPU mode is switched from user to kernel. Owing to the privilege-aware monitoring (*P1*), we can configure the watchpoints to monitor only kernel-privileged accesses, which implies that the watchpoint does not need to be reconfigured (deactivated) for returning to the user mode as long as the user process does not use the watchpoint.

According to ARM [1], the monitoring start address should be aligned with the size of minitoring. Based on this requirement, the monitored kernel text area needs to be aligned with its size. Further, the size needs to be a power of two. Our Linux kernel size (Linux kernel version 14.12 with Linaro 17.10) is approximately 8.7 MB, which requires 16 MB to be monitored in order to cover the entire text area and satisfy the requirement. Because the text and data sections are adjacently located in memory by default, the text and data need to be decoupled. Toward this end, we enforce the alignment with 0x1000000 between the text and the data by manipulating a linker script.

On ARM architecture, the size of the immediate value for an individual instruction is limited (e.g., 12 bits for LDR instruction); thus, a 64-bit value cannot be fully expressed by a single instruction. To address this problem, a compiler uses a literal pool [8], which stores data in code sections. In the Linux kernel, we find such cases. For example, the `get_symbol_pos` function reads the address value stored in the literal pool to lookup the kernel symbol information in the data segment. To support such benign operations, we patch the kernel code such that XOM is temporarily disabled during execution. This can be simply conducted by flipping the 'enable bit' in the watchpoint control register.

5 IMPLEMENTATION

We implemented our prototype on Juno board, offering Cortex-A57 and Cortex-A53 based on big.LITTLE architecture. We used Linaro software platform, which supports ARM Trusted firmware and 64-bit Linux (4.12.0).

For XOM enforcement for the kernel text, the `kernel_entry` macro in `entry.S` is patched. The macro is used by the exception vector. As the exceptions can occur in both user and kernel modes, the macro handles both cases. For the entry from user to kernel, we clear PSTATE.D and configure watchpoints based on the kernel text address and size. On the other hand, we only clear PSTATE.D for exception occurrence during the kernel mode execution to keep activating XOM (note that debug exceptions are automatically disabled upon exception occurrence). To separate the text and the data segments, we modified the `vmlinux.lds.S` linker script to enforce the 16 MB alignment between the text and the data.

PAN implementation requires the `kernel_entry` macro to be patched similarly to that of XOM. Further, the `context_switch` function in `sched/core.c` is patched to update the watchpoint configuration based on `active_mm` of the newly scheduled process. The kernel functions that aim to access the user space, such as `copy_from_user` and `get_user`, are patched to use the unprivileged load and store instructions in association with the PAN enforcement. Note that the Linux kernel uses the functions to access the kernel spaces as well (e.g., `get_user` invocation in `dump_mem`). To this end, we selectively use one of the privileged and unprivileged instructions depending on the `addr_limit` value of the current process, which can dynamically be adjusted using `set_ds()` in the kernel.

Table 2: System call performance with TTBR-based PAN and watchpoint-based PAN (in μs).

System call	Linux	TTBR_PAN	WP_PAN
gethostname	269.7	302.4 (1.12×)	309.5 (1.14×)
settimer	229.6	253.5 (1.10×)	244.1 (1.06×)
getdomain	216.3	255.3 (1.18×)	248.7 (1.14×)
pipe	1191.6	1221.1 (1.02×)	1222.6 (1.02×)
sched_getaffinity	225	263.9 (1.17×)	252.3 (1.12×)
getcwd	306.7	327.3 (1.06×)	322.7 (1.05×)
semctl	266.1	298.9 (1.12×)	303.8 (1.14×)
ptrace	195.1	202.2 (1.03×)	202.7 (1.03×)

6 SECURITY ANALYSIS

In our PAN and XOM implementation, the watchpoint configuration is conducted by the kernel. As the solutions expect kernel-privileged attacks, resilience to attacks that attempt to compromise the watchpoints should be carefully considered. In this regard, the security of XOM and PAN can be enhanced by adopting fine-grained ASLR [19]; the location of the code chunk that configures the watchpoints can be obfuscated by applying ASLR. Meanwhile, as an alternative to adopting ASLR, we can use the kernel integrity monitors [10, 11], which are widely deployed on commercial mobile devices, to protect the watchpoint-based security solutions. The integrity monitors trap, verify, and emulate security critical operations. Similarly, we can protect the watchpoint configuration as part of such security critical operations. The feasibility of coordinating our approaches with existing defensive measures will be explored in the future.

7 PERFORMANCE EVALUATION

LMBench. To evaluate the kernel-level solution overhead, we ran LMBench for each solution and their combinations: WP_XOM, TTBR_PAN, WP_PAN, WP_XOM + TTBR_PAN, and WP_XOM + WP_PAN. In our experiment, WP_XOM uses one watchpoint and WP_PAN requires a maximum of three watchpoints to cover all VMAs for a certain active process. Figure 2 shows the run time of each case normalized to Linux. WP_XOM imposes less overhead compared to other cases, the maximum of which is 17% in the `write` system call test. It indicates the tendency that the tests with less elapsed time, such as simple syscall, are observed with more overhead.

Among the PAN solutions, WP_PAN outperforms TTBR_PAN in most test cases. The maximum overhead of WP_PAN was at most 27% for the `read` system call. However, the performance of TTBR_PAN, specifically in signal handling, was much worse. More than 200% and 300% overhead was incurred by enabling TTBR_PAN and WP+XOM + TTBR_PAN, respectively. We believe that this is due to the fact that the kernel needs to frequently access the user

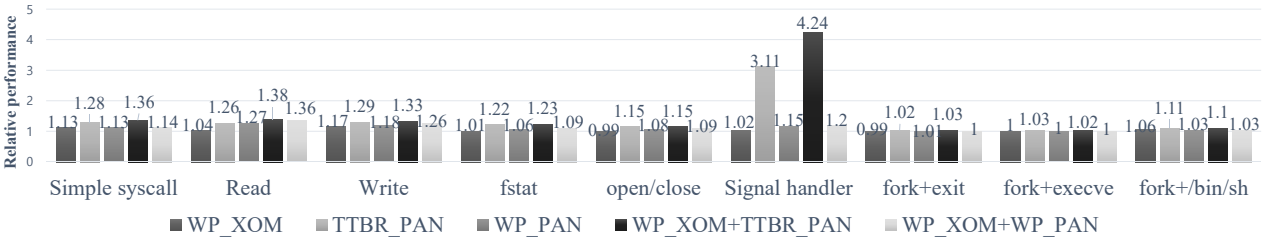

Figure 2: Performance measured by LMBench with watchpoint-based XOM and PAN, TTBR-based PAN, and their combinations. The result is normalized to Linux (lower is better). In most cases, watchpoint-based PAN outperforms TTBR-based PAN. Specifically, in handling signals, the overhead of TTBR-based PAN surges up owing to the high ratio of accessing the user space in its entire operation.

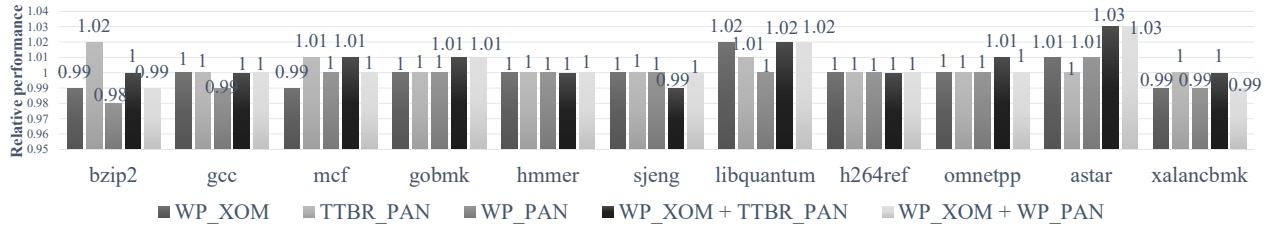

Figure 3: SPEC CPU2006 with watchpoint-based XOM and PAN, TTBR-based PAN, and their combinations. The result is normalized to Linux (lower is better). A maximum overhead of 3% was imposed for the XOM and PAN combination with the ASTAR test case.

memory for the preparation of signal handler invocation (e.g., copying current kernel stack into user memory) and thus reconfigure the TTBR repeatedly. By contrast, owing to the harmony between watchpoint and unprivileged instructions (*P2*), WP_PAN can naturally access the user memory without additional operations. Finally, combining WP_XOM with PAN solutions increases the overhead slightly but it is negligible in the tests with long elapsed time, such as `fork+exit`, `fork+execve`, and `fork+/bin/sh`.

System calls. To further investigate the performance difference between the PAN solutions, we ran additional tests that invoke several system calls that require the kernel services to explicitly access the user memory. For example, kernel directly writes the user memory via the `copy_to_user` function to support the `gethostname` system call. Table 2 shows the time elapsed for 100 iterations of each system call. In contrast to the signal handling test in LMBench, the performance difference in the additional tests between TTBR_PAN and WP_PAN was not that significant (within 5%). We suspect that the portion of accessing user memory in the additional tests is even smaller than that of signal handling; thus, the impact of the overhead is limited.

SPEC CPU2006. Finally, we ran SPEC CPU2006 benchmarks for the five aforementioned cases. Figure 3 shows the benchmark results normalized to Linux. Compared to the results obtained with LMBench, the overhead measured by CPU2006 was even lower because the performance impact of security measures was obscured by the relatively long runtime of the CPU2006 test programs. In most cases, the overhead was negligible (at most 3%). Further, we could not find distinguishable performance differences between the respective cases. The proposed measures impose overhead based on the occurrence of general system events such as the context switch and interrupt. Thus, as long as the test program does not intentionally (or frequently) create exceptions or yield CPU time, we will observe similar overhead between the tests. Finally, the overhead imposed by PAN was slightly higher than that by XOM because of the higher complexity of the PAN enforcement operation (e.g., traversal of `active_mm`).

8 DISCUSSION

8.1 Feasibility

In our approach, the number of supported watchpoints is important to ensure the effectiveness of the protection. In a PAN emulation, we checked all process memory mappings by referring to the `maps` file in `/proc` and found that a maximum of three watchpoints, each monitoring 2 GB of memory, are sufficient for protecting our Linux environment. For application memory mapping requiring more than four watchpoints in a PAN emulation, we can temporarily use the TTBR-based approach. In addition, the OS memory allocation mechanism can be enhanced to condense the application memory allocation under the watchpoint monitoring.

Meanwhile, the current kernel XOM implementation can be reinforced with additional security artifacts such as code diversification, return address hiding, and kernel module protection [13, 18]. In particular, protecting the module text might require reorganizing the kernel memory map to properly locate the text under watchpoint monitoring. We expect four watchpoints to be sufficient for protecting additional static kernel objects.

8.2 Compatibility

GDB, a user-level debugger, can monitor data accesses by configuring the watchpoints. The configuration is triggered by a system call (e.g., ptrace), which directly updates the relevant registers. However, this mechanism might disable our kernel protection solutions. We resolve this issue in a similar manner as our PAN implementation. An array of watchpoint-related structures is added to `thread_info`. We then update the array with watchpoint configuration values that are generated during the system call handling instead of directly updating the watchpoint registers. By doing so, we can preserve the watchpoint setup for kernel solutions. Later, switching to user mode, the values in the array are configured as the watchpoint register values. For the kernel debugging (e.g., KGDB), our solutions need to be disabled by recompiling the kernel.

8.3 Scalability

As illustrated in Table 1, the watchpoints can be enabled during both a secure state and a non-secure state. Hence, our approach can be adopted in the trusted execution environment (TEE) for secure OS protection. Because the debugging registers are shared between two states, every processor mode switch between states might require saving and restoring the debugging register values, in addition to the watchpoint setting for secure OS protection. In addition, the watchpoints are supported in the hypervisor layer, which opens the door to the design of useful security facilities for an enhancement in hypervisor security. For example, the kernel integrity monitor running in the hypervisor layer [10] can protect its text region by configuring the watchpoints.

9 CONCLUSION

We analyzed in detail the properties of debugging facilities, namely, watchpoints, and showed how to leverage them as hardware security primitives. The practicality of our approach was demonstrated by designing and implementing two security applications: kernel XOM and PAN. Based on a performance evaluation, our approach was shown to incur negligible overhead. Specifically, watchpoint-based PAN outperformed TTBR-based PAN, which is one of the kernel security features in Linux.

ACKNOWLEDGMENTS

This work was supported by the NRF (NRF-2017R1A2B3006360), IITP (IITP-2017-0-01889), and ONR (N00014-18-1-2661) grants.

REFERENCES

[1] 2018. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. (May 2018). https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile
[2] 2018. ARM Cortex -A Series: Programmer's Guide for ARMv8-A. (May 2018). http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf
[3] 2018. Efficient Application Processors for Every Level of Performance. (May 2018). https://www.arm.com/products/processors/cortex-a
[4] 2018. Exploit Methods/Userspace data usage. (May 2018). https://kernsec.org/wiki/index.php/Exploit_Methods/Userspace_data_usage
[5] 2018. Introduction to Intel Memory Protection Extensions. (May 2018). https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions
[6] 2018. Juno ARM Development Platform SoC. (May 2018). https://www.arm.com/files/pdf/DDI0515D1a_juno_arm_development_platform_soc_trm.pdf
[7] 2018. Lifting the (Hyper) Visor: Bypassing Samsungąŕs Real-Time Kernel Protection. (May 2018). https://googleprojectzero.blogspot.com/2017/02/lifting-hyper-visor-bypassing-samsungs.html
[8] 2018. Literal pools. (May 2018). http://www.keil.com/support/man/docs/armasm/armasm_dom1359731147760.htm
[9] 2018. Racehound. (May 2018). https://github.com/kmrov/racehound
[10] 2018. Real-time Kernel Protection (RKP). (May 2018). https://www.samsungknox.com/pt-br/blog/real-time-kernel-protection-rkp
[11] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: real-time kernel protection from the ARM trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 90–102.
[12] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1342–1353.
[13] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. 2016. Leakage-Resilient Layout Randomization for Mobile Devices.. In *NDSS*.
[14] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-grained Execution Units with Private Memory. In *Security and Privacy, 2016. SP 2016. IEEE Symposium on*. IEEE.
[15] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling execute-only memory for COTS binaries on AArch64. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 304–319.
[16] Lee Chew and David Lie. 2010. Kivati: fast detection and prevention of atomicity violations. In *Proceedings of the 5th European conference on Computer systems*. ACM, 307–320.
[17] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2015. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, 177–189.
[18] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 763–780.
[19] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization.. In *USENIX Security Symposium*. 475–490.
[20] Jinsoo Jang and Brent Byunghoon Kang. 2018. Retrofitting the Partially Privileged Mode for TEE Communication Channel Protection. *IEEE Transactions on Dependable and Secure Computing* (05 2018).
[21] Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. [n. d.]. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15), San Diego, CA*.
[22] Kari Kostiainen, Jan-Erik Ekberg, N Asokan, and Aarne Rantala. 2009. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ACM, 104–115.
[23] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. [n. d.]. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17), San Diego, CA*.
[24] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards transparent tracing and debugging on arm. In *26th USENIX Security Symposium (USENIX Security 17)*.
[25] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. kRˆ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 420–436.
[26] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. 2014. A primer on hardware security: Models, methods, and metrics. *Proc. IEEE* 102, 8 (2014), 1283–1295.
[27] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 574–588.
[28] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 35–46.
[29] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 558–569.