Hypernel: A Hardware-Assisted Framework for Kernel Protection without Nested Paging

Donghyun Kwon
Seoul National University
dhkwon@sor.snu.ac.kr

Kuenwhee Oh
KAIST
kuenwhee.oh@cs.kaist.ac.kr

Junmo Park
Seoul National University
jmpark@sor.snu.ac.kr

Seungyong Yang
KAIST
syyang@kaist.ac.kr

Yeongpil Cho
Soongsil University
ypcho@kaist.ac.kr

Brent Byunghoon Kang
KAIST
brentkang@kaist.ac.kr

Yunheung Paek
Seoul National University
ypaek@snu.ac.kr

ABSTRACT

Large OS kernels always suffer from attacks due to their numerous inherent vulnerabilities. To protect the kernel, hypervisors have been employed by many security solutions. However, relying on a hypervisor has a detrimental impact on the system performance due mainly to *nested paging*. In this paper, we present *Hypernel*, a security framework combining hardware and software components to address this problem. *Hypersec*, the software component, provides an isolated execution environment for security solutions, and the hardware monitor component enables a word-granularity monitoring capability on the kernel memory. Our evaluation shows that Hypernel efficiently fulfills the role of a security framework, while imposing mere 3.1% of runtime overhead on the system.

1 INTRODUCTION

Virtually all prominent operating system (OS) kernels, such as Windows and Linux, follow the *monolithic* design principle in which all of the huge kernel code is supposed to run in a single address space. As a result, if any vulnerability in a certain part of the kernel code (e.g., buggy device drivers) is exploited by an adversary, it would lead to subverting the entire kernel security. Considering that the OS kernel is usually assumed to be the trusted computing base (TCB) of the system, this means that even a single vulnerability could be exploited by the adversary to seize the system. To make the matter worse, as the number of exploitable kernel vulnerabilities is steadily increasing in recent years [2], it is becoming harder and harder to protect the kernel from such exploits.

To address that, many depend on the *hypervisor-level* support for their security solutions [14, 20, 22, 26]. In modern computer systems, the hypervisor acts as an intermediary between the underlying hardware and OS kernel. Leveraging such a unique role of the hypervisor in the system, hypervisor-level solutions protect kernels against rootkits [27], and moreover, isolate security critical appliances [8] from the kernels that might be housing malware. To realize security measures, they commonly utilize three functionalities of a hypervisor: *instruction trapping*, *hypercall* and *nested paging*. Instruction trapping allows the hypervisor to monitor sensitive kernel events like an update of system control registers. Hypercalls, normally a means for the kernel to invoke the hypervisor, are injected by hypervisor-level monitors into sensitive regions of the kernel code to observe security policies and procedures. Most importantly, nested paging is an indispensable tool for kernel protection that can be used to audit kernel memory activities without being interfered by the kernel. Together, a hypervisor-level security solution can monitor and secure privileged services and data transactions within its TCB completely isolated from the kernel.

Unfortunately, employing a hypervisor for kernel protection comes at a cost. According to previous studies [9, 25], even on a processor with hardware virtualization, activating a hypervisor alone may incur up to about 30% of runtime overhead of applications due primarily to nested paging that requires two stages of address translation for every memory access. To avoid this two-stage paging overhead while embracing the same security strength of a hypervisor-level security solution, our newly proposed security framework for kernel protection, called *Hypernel*, does not rely on nested paging. Hypernel instead relies on special hardware as well as a new software module, called *Hypersec*. The role of Hypersec is to provide security appliances or monitors with a secure execution environment isolated from the kernel which we do not trust. For this, Hypernel leverages the hardware extension for virtualization [18, 19] to ensure that only Hypersec is designated to modify predetermined security critical kernel and hardware resources, such as the kernel page table and hardware configuration registers.

Relying on a hypervisor for kernel protection also suffers non-negligible runtime overhead due to redundant hypercalls and traps. The reason for such redundancy is the granularity gap of protection as described in [27]. For kernel data protection, the hypervisor, in general, enforces an access control at the page granularity, meaning that if a page contains sensitive data, the whole page would also be marked sensitive. Unluckily, many sensitive data like kernel invariants tend to be scattered all over the memory space and much sparsely organized within a page. Thus very often in practice, sensitive and non-sensitive kernel data are allocated together in the

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196061

same page, resulting in many redundant traps/hypercalls to handle permission faults caused by accesses to non-sensitive data that happen to reside in a page with values to monitor. In our work, to eliminate this redundancy, Hypernel is equipped with a hardware module, called the *memory bus monitor* (MBM), that is designed to eavesdrop on the system bus between the host processor and main memory. With the help of MBM, Hypernel enforces and does access control at a finer granularity, i.e., word granularity: MBM monitors writes to every memory word and raises an interrupt upon finding any write attempt to sensitive data being defended by Hypersec.

As described in Sections 5 and 6, we try to validate our approach empirically by implementing a prototype of Hypernel on the Versatile Express Juno r1 Platform [4] running AArch64 Linux kernel 3.10. Our MBM is synthesized on the LogicTile Express daugtherboard [3] and added onto the Juno platform. Then we have evaluated the prototype of Hypernel with a set of benchmarks and found that Hypernel increases the execution times of applications by about 8% even at worst case, achieving much lower performance overhead than existing hypervisor-level solutions. As reported in Section 7, our word-granularity access control scheme with MBM has eliminated a majority of the original trap events that would have occurred in an existing page-granularity scheme.

2 RELATED WORK

Many people endeavor to build security frameworks for kernel protection. Their approaches can be categorized into three as follows.

Privileged software-based. Using a hypervisor has long been the most prominent way of protecting the OS kernel because of its role as an intermediary with the underlying hardware. Early approaches were realized on top of commercial hypervisors [14, 20, 22, 26], but they incurred heavy performance burden and had a possibility of being vulnerable by itself [1] due to their large code bases. Whereas the vulnerability issue was solved later by reducing the TCB [25], the fundamental performance issue arising from nested paging still lingered. Some people relied on the trap-and-emulate features of ARM TrustZone in their attempts to help their security monitors supervise the entire physical memory space without nested paging [5, 13]. However, these features do not provide the monitors with sufficient capabilities to intervene with the kernel operations and system resource control, resulting in their security solutions being filled with extensive kernel modifications to inject hooks in the code. Moreover, the protection granularity gap issue still haunted those TrustZone-based solutions because of their dependence on the page-granularity scheme to seize control of the memory. Meanwhile, in some early work, the runtime overhead caused by the protection granularity gap was resolved [27], but this demands a significant engineering effort for kernel code analysis and patches.

Kernel instrumentation. Some approaches strived to quarantine a majority of the untrusted part of a monolithic kernel from a small TCB which accesses the sensitive resources [6, 7, 11]. Specifically, in these approaches, a monolithic kernel is first partitioned into the trusted inner and untrusted outer kernel domains. Then the divided domains are asymmetrically given memory access permissions in a way that the inner one is allowed to access the outer one but not vice versa. By doing so, the inner kernel has an unrestricted monitoring capability over the outer one. Such an approach excels at performance thanks to no additional layer being inserted into the system. However, segregating the kernel requires an extensive analysis of the kernel structure, and a naïve implementation might fall into the protection granularity gap problem like aforementioned privileged software-based approaches.

Hardware-based. There were several efforts to extend the hardware to efficiently and securely monitor critical kernel data. Hardware security monitors can be classified into internal and external monitors. An internal security monitor requires a completely new design of an instruction set architecture (ISA) or a microarchitecture to maintain kernel security [12, 28, 29]. It allows software developers to easily secure their programs with high efficiency via processor-level security features. However, forcing a new processor design just to improve security is in practice very hard and slow to be accepted by commodity products that depend on a vast amount of legacy software built on top of existing processors. An external security monitor is usually immune from such a problem since it resides outside the processor (e.g., in the memory bus [17] or even on an expansion card slot [16]). Since its design does not entail modifications of existing processor architectures, it could be integrated more easily into a commercial system running legacy software. However, external monitors are unavoidably subject to the *semantic gap* problem as they have a limited capability to see the internal states of the host processor. Thereby, they were bypassed even by a relatively straightforward scheme [15]. In contrast, Hypernel suffers much less from this problem because Hypersec is securely running inside the processor with the help of the hardware extension for virtualization, and thus Hypernel has virtually the same security strength and ability to monitor the host processor internal as the privileged software-based approaches.

3 BACKGROUND

Hypernel exploits some architectural features: privilege levels and virtualization extension. These features are widely supported by many architectures such as x86 and ARM. Therefore, Hypernel can be implemented in these architectures. Currently, however, the prototype of Hypernel make use of the features of 64-bit ARM architecture (a.k.a AArch64). Thus, we will briefly explain these features of AArch64 in this section.

Exception levels. In AArch64, the privilege level is called Exception Level (`EL`). The exception level with the higher value of x in `EL`x has more privilege than the lower one. As shown in Figure 1, `EL0` and `EL1` are exception levels used by user applications and the kernel, respectively. The hypervisor runs on `EL2`.

Virtualization extension. ARM virtualization extensions provide several features for hypervisors to work efficiently in the `EL2`. First, the hypervisor can resort instruction trapping feature by setting the `HCR_EL2` register. By using this register, the hypervisor can trap several kernel events such as an update of Translation Table Base Register (`TTBR`) register. Also, It supports a hypercall instruction called `HVC`, which allows the kernel to enter hypervisor by executing it. The last feature is *nested paging* [1] which is used for virtual address translation in `EL0` and `EL1`. The memory management unit (MMU) which support nested paging simultaneously manages two levels

[1] Nested paging is also known as *second-level address translation* (SLAT), *Extended Page Table* (EPT) and *Rapid Virtualization Indexing* (RVI).

of page tables: the stage-1 page table and the stage-2 page table. The stage-1 page table, which is also the page table used in EL1, is the page table used by the kernel and translates a virtual address into an intermediate physical address (IPA). After that, the stage-2 page table translates the IPA into the physical address. Apart from this, the EL2 page table is used by the hypervisor, for virtual address translation in EL2. Figure 1 shows these page tables and their corresponding exception levels.

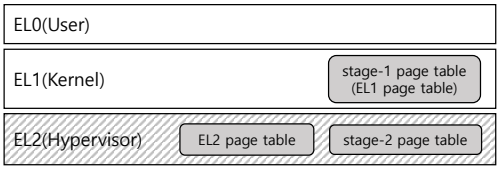

Figure 1: Exception Levels in AArch64

4 THREAT MODEL AND ASSUMPTION

In this paper, we follow the threat model and assumptions not vastly different from an existing security framework for kernel protection. We assume a secure boot mechanism such as AEGIS [23] or UEFI [24] is available to safely load and initialize Hypernel and the kernel. Apart from that, we assume that an attacker could successfully exploit any existing kernel vulnerabilities to alter the kernel memory. However, we do not consider any physical attacks, denial-of-service (DoS) attacks, and side channel attacks.

5 DESIGN

This section describes the design details of Hypernel, focusing on how to build an efficient security framework without nested paging.

5.1 Design Overview

Figure 2 shows the overall design of Hypernel which provides a *normal space* for the kernel and user applications running on it, and a *secure space* for security applications. To isolate the secure space from the normal space, the secure space is designed to operate in the hypervisor privilege, which is higher than the normal space's kernel/user privilege. A software module called *Hypersec* is installed on the secure space and provides an isolated execution environment for security applications. Note that Hypernel does not employ nested paging, unlike existing hypervisor-based security frameworks. Its details will be presented in Section 5.2. To efficiently intercept suspicious behaviors of the kernel, Hypernel includes an external hardware, called the *memory bus monitor* (MBM). Thanks to this hardware, Hypernel can monitor the device memory with finer granularities than in existing hypervisor-based approaches. The details will follow in Section 5.3.

5.2 Isolated Execution Environment

To provide security applications with an execution environment isolated from the kernel, we need a mechanism that offers asymmetric memory views to the normal and the secure spaces. In other words, from the secure space, it must be possible to access the normal memory region while the normal space is blocked from accessing the secure region. Existing hypervisor-based approaches used nested paging to prevent accesses to security applications' memory space via the stage-2 page table management. However,

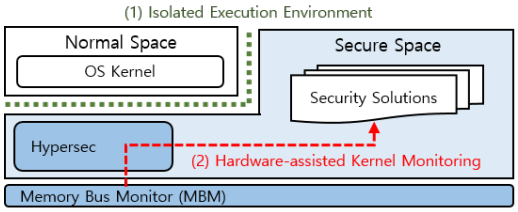

Figure 2: An overview of Hypernel framework.

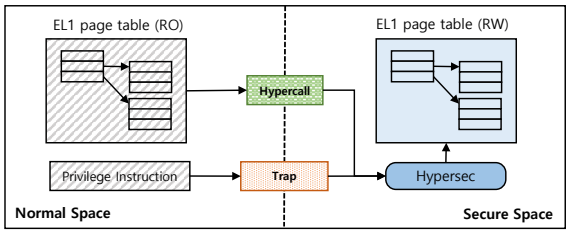

Figure 3: Isolated execution environment w/o nested paging.

this requires two stages of page table walk per address translation, obviously consuming extra execution time. Instead, Hypersec directly monitors the kernel page table and enforces no page table entries are created to map the memory region of the secure space.

5.2.1 Verifying the OS Kernel Page Table. Since the kernel manages the page table of the normal space, there is a risk that a compromised kernel might tamper with its page table. For example, the attacker might try to access the secure space by modifying the kernel page table to create a mapping for the memory region of the secure space. Therefore, to protect the kernel page table from the vulnerable kernel, we replaced the kernel code which updates the kernel page table with a hypercall, a la TZ-RKP [5]. This modification allows Hypersec, which resides in the secure space, to handle it instead. At this time, Hypersec verifies the request for updating the kernel page table. Specifically, Hypersec applies the W⊕X policy to the kernel memory region, sets the memory region used for the kernel page table as read-only, and makes the memory region for the secure space inaccessible.

5.2.2 Trapping Privileged Instruction Execution. Privileged instructions are executed to update or read the configuration of the system. For example, a privileged instruction could be used to change the value of the page table base register (e. g. `CR3` for x86, `TTBR` for ARM). However, there is a risk that an attacker could disable the isolated execution environment provided by Hypernel using these privileged instructions. That is, an attacker could use a maliciously generated page table for address translation, or even disable address translation. Therefore, Hypernel uses the feature for trapping privileged instructions provided by the virtualization extension to prevent such malicious execution of privileged instructions. As shown in Figure 3, when a privileged instruction is executed, a trap is generated, and Hypersec asserts that the settings necessary for Hypernel to operate normally are not changed.

5.3 Hardware-assisted Kernel Monitoring

It is essential to monitor the kernel memory space to protect kernel integrity. To achieve this, previous hypervisor-based approaches using nested paging mark stage-2 page table entries for regions to

monitor as read-only. Then, if a permission fault occurs due to an attempt to update the region, the hypervisor intervenes and verify the validity of the memory write event. However, aside from the aforementioned page table walk overhead, monitoring via nested paging is also prone to overhead caused by the protection granularity gap issue [27]. To avoid these problems, Hypernel has the MBM, a hardware module connected to the system bus between the CPU and main memory. The main functionality of the MBM, like those described in previous works [17], is to monitor fine-grained memory regions than page-granularity. However, this bus monitor hardware has a problem that it cannot know the information inside a processor. For example, since this approach cannot be aware of the memory addresses of dynamically allocated kernel data objects, it can only be used for monitoring limited kernel objects. Even worse, It is also known to be vulnerable to address translation relocation attack (ATRA) [15] because the mapping information between physical addresses and virtual addresses is not exposed to the monitor. However, in Hypernel, as Hypersec can provide the internal state of a processor as described in 5.2, we could use the MBM without such problems.

Figure 4 shows how the security application monitors the kernel memory using Hypersec and MBM in Hypernel. The green line indicates the workflow when a new memory region to monitor is registered, and the red shows the workflow when a write event occurs in the monitored memory region. The security application informs Hypersec with new regions to be monitored via the hooks inserted into the kernel code. When the hook (hypercall) is executed (❶), Hypersec receives the ID of the security application (SID), the base address and the size of the region as arguments. Hypersec then translates the virtual address of the monitored region to the physical address and delivers it to the MBM. At this time, the monitored region is represented at the word granularity through a bitmap which maps one word (8 bytes) to one bit. Hypersec enables the bits corresponding to the requested region in bitmap (❷). Also, to allow the MBM to monitor every memory event for that region, Hypersec modifies the kernel page table so that any cache entry for the page including the monitored region is not generated. Then, whenever a memory write event to the kernel memory space occurs (❸), the MBM verifies if there is a corresponding bit to the memory event in the bitmap (❹). If the bit is present, the MBM records the information of the event (address, value) in a ring buffer (❺) and raises an interrupt to notify Hypersec (❻). Hypersec handles the interrupt by fetching the event information from the output buffer (❼) and delivering it to security applications (❽). Note that since all the memory regions used by the MBM including the bitmap and the output buffer are located in the secure space, the kernel cannot undermine the MBM operation.

6 IMPLEMENTATION

We implemented the prototype of Hypernel on the ARM Versatile Express Juno r1 Development Platform [4], which uses the big.LITTLE architecture integrating a Cortex-A57 1.15 GHz dual-core processor with a Cortex-A53 650 MHz quad-core processor. We also implemented MBM on the LogicTile Express 20mg daughterboard [3] and added it to the Juno platform. To enable MBM

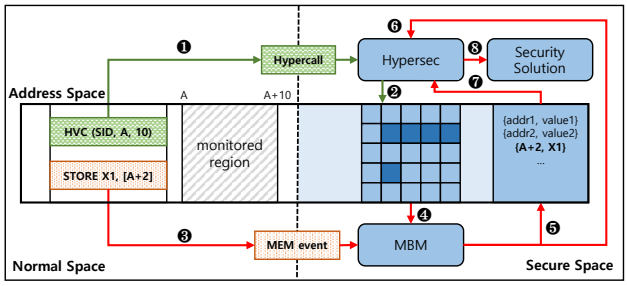

Figure 4: The operation routine of kernel monitoring.

to monitor the memory traffic of the processors of the Juno platform, the bootloader is modified so that the system uses the 128MB SDRAM on the daughterboard instead of the DRAM installed on the Juno platform. We used Linux 3.10.79 as the OS kernel. Hypersec consists of about 1.5 KLoC, and approximately 200 SLoC of Linux kernel code is modified to help interoperation between Hypernel and the kernel. The MBM was implemented using about 55,000 gates on the daughterboard.

6.1 Hypersec

Hypersec is initialized during the system booting process. The main purpose of this process is to set up the system control registers of the EL2. First, Hypersec builds an EL2 page table to be used in the secure space. Currently, the EL2 page table employs the linear mapping strategy so that each EL2 virtual address is equal to the physical address. Hypersec then goes on to initialize SP_EL2, which is the stack pointer register for EL2. Hypersec sets the exception vector in EL2 so that Hypersec can handle synchronous exceptions (hypercalls, traps) which occur while the kernel operates. Finally, Hypersec enables the TVM bit of the HCR_EL2 register to trap attempted modifications of registers related to virtual memory configuration. For example, Hypersec monitors the update of TTBR1_EL1, which holds the base address of the EL1 page table, to enforce that the kernel only uses the valid EL1 page table.

6.2 Kernel Instrumentation

We modified the kernel code to prevent the kernel from tampering with the EL1 page tables in the same way as existing works [11, 21]. First, we modified the kernel to make it allocate memory blocks in 4KB pages instead of 2MB sections. Normally the Linux kernel for AArch64 allocates memory blocks in the kernel linear region in 2MB sections, while the size of each page table is 4KB. This discrepancy might result in a memory section containing both page tables and other irrelevant data. Therefore, if we directly enforce the read-only policy on the vanilla kernel, we have to enforce it on each section containing such page tables, leading to a protection granularity gap issue. To prevent this issue, we instead forced the kernel to allocate memory spaces in 4KB pages. Since each page table is also 4KB, we can effectively enforce read-only policy on page table pages without triggering Hypersec intervention for other data. Then we modified the kernel to force it to write onto the kernel page table via hypercalls instead of directly modifying the page table. Hypersec verifies the write action and writes on behalf of the caller if the request is valid. Meanwhile, we inserted a hypercall

in the kernel interrupt handler to allow Hypersec to handle this interrupt.

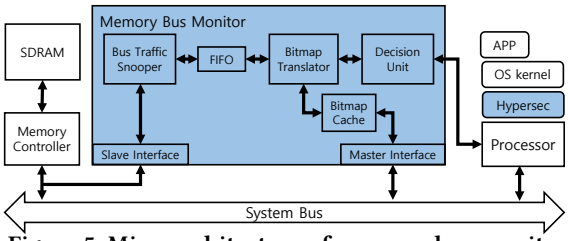

Figure 5: Microarchitecture of memory bus monitor

6.3 Memory Bus Monitor

Figure 5 shows the microarchitecture of MBM. The MBM consists of several hardware modules: the bus traffic snooper, the bitmap translator, the decision unit and the bitmap cache. The bus traffic snooper, a hardware module that monitors the memory bus traffic, captures the write address/value pairs. The captured write address/value pairs are temporarily stored in the FIFO buffer. When the bitmap translator is in the idle state, it loads the captured data from the FIFO buffer and calculates the corresponding bitmap address. Then, the bitmap translator reads the bitmap data from the main memory. Since, however, accessing the main memory and fetching the bitmap data for every write event in the same region is inefficient, we implemented a bitmap cache in MBM. The bitmap cache follows the read-allocate cache policy and is updated when a memory write event to the bitmap is detected. After loading the bitmap data, the bitmap translator delivers it to the decision unit. Then, the decision unit checks if a bit of the bitmap data, which represents whether the write event should be monitored or not, is enabled. If it is, the decision unit sends an interrupt to the host CPU. Hypersec finally intercepts the interrupt and handles the event.

7 EVALUATION

In this section, we evaluated the performance overhead caused by Hypernel and the efficiency of its kernel monitoring mechanism. First, we compared the performance overhead of Hypernel against the KVM hypervisor [10] to see how efficiently Hypernel provides the isolated execution environment. Next, to measure the efficiency of the kernel monitoring mechanism of Hypernel, we adopted a security solution which monitors sensitive kernel data on Hypernel, and we compared the number of inspected memory events under our word-granularity monitoring scheme with page-granularity monitoring scheme during the kernel execution.

7.1 Performance Evaluation

To evaluate the performance overhead of Hypernel, we conducted experiments on the following three cases: native, KVM-guest, and Hypernel. **Native** means that the base kernel runs without Hypernel. **KVM-guest** is the case where a base kernel is running on a KVM virtual machine (VM). In **Hypernel**, the kernel is working with our Hypernel. Sadly, due to the memory requirement that the host kernel should support the VM, KVM cannot work in the implementation of our Hypernel prototype. Thus, we changed the empirical environment to use DRAM (2GB) of the motherboard instead of the SDRAM in the daughterboard. Surely in this setting, only Hypersec is working in the case of Hypernel. However, Hypersec provides the isolated execution environment, and so we used this experimental setup to measure the performance overhead. Meanwhile, for the consistency of the results, the performance experiment was conducted on the Cortex-A57 big core.

Table 1: Execution time of kernel operations (μs).

Test	Native	KVM-guest	Hypernel
syscall stat	1.92	1.83	1.94
signal install	0.68	0.75	0.68
signal ovh	2.96	3.38	2.98
pipe lat	10.07	11.45	10.68
socket lat	13.76	16.08	14.51
fork+exit	271.68	337.84	314.77
fork+execv	285.53	351.81	340.70
page fault	1.57	1.98	1.89
mmap	24.60	28.40	27.50

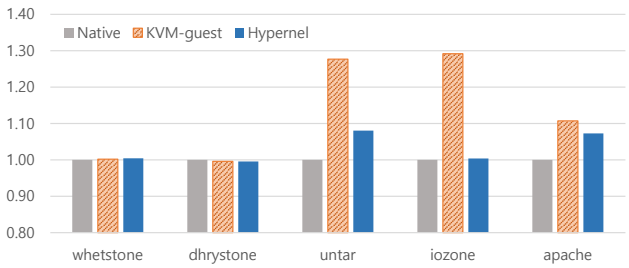

Figure 6: Application benchmarks results

7.1.1 Individual Kernel Operations. Since Hypernel is a security framework for kernel protection, it works closely with the kernel to control the system. To measure the cost of each kernel operation with Hypernel, we experimented the LMbench test suite. As listed in Table 1, both KVM and Hypernel add extra cycles to each kernel operation, as can be expected. In comparison between KVM and Hypernel, the execution times of kernel operations are basically comparable. In fact, the kernel runs slightly faster for most cases with Hypernel than with KVM. On average, the kernel gets slower by 15.5% and 8.8%, respectively with KVM and Hypernel.

7.1.2 Performance of Application Benchmarks. Now, we would like to show the actual performance impact of Hypernel on applications. Figure 6 compares the performance results measured for three test cases. On average, KVM-guest and Hypernel incur 13.5% and 3.1% of the performance overhead, respectively. KVM-guest shows worse performance than Hypernel due to the system virtualization including nested paging, which occurs at every time a virtual address is translated to physical and machine addresses. On the other hand, for Hypernel, we were able to reduce the overhead by adding security checks only for certain kernel operations, such as the page table management.

Table 2: Comparison of the number of trap events.

benchmark	Hypernel page-granularity	Hypernel word-granularity
whetstone	525	48(9.2%)
dhrystone	637	39(6.1%)
untar	2173870	96467(4.4%)
iozone	1510	117(7.7%)
apache	48650	1754(4.4%)

7.2 Efficiency of Kernel Monitoring

Hypernel proposed a hardware-assisted kernel monitoring scheme to overcome the inefficiency of the existing page-granularity monitoring scheme. To evaluate our scheme, we conducted the following experiment. First, we created two versions of security solutions. One security solution monitors only the sensitive fields of the target kernel data objects (`cred`, `dentry`)[2] and verifies the integrity of these fields. The other solution also validates these fields, but it monitors the entire fields of target kernel data objects. Then we recorded the number of interrupts generated by the MBM while the application benchmarks are running for each configuration. Through the results of second security solution, we were able to estimate the number of unnecessary faults that would have occurred if a security framework working on page-granularity was used. This kind of estimation was possible since the number of interrupts that occur when monitoring the entire object would be the same as the number of faults that occur when the target kernel data objects are aggregated in specific pages, and the security framework monitors these pages by configuring as read-only. As shown in Table 2, it was confirmed that Hypernel could efficiently protect sensitive data through only about 6.2% of trap events compared to the existing page-granularity mechanisms.

8 DISCUSSION

Formal verification of Hypersec. As a software component, Hypersec itself might have software vulnerabilities and be subject to attacks exploiting them. The usual solution for this problem is to formally verify the code that it works as intended and there is no vulnerability to be exploited. Since large programs could be hardly verified, it is important to shrink the code base of a security solution small enough to be verifiable. The entire code of Hypersec has about 1500 SLoC. This is comparable with 4,000 SLoC of the C component of the Nested Kernel prototype [11] and 6,000 SLoC of XMHF's core component [25] which are seen to be verified properly.

DMA attacks to Hypernel. To improve performance, some peripheral devices can access the physical memory using direct memory access (DMA). Thus, Hypernel must thwart the adversary's attempt to tamper with the memory region of the secure space through DMA. Although the countermeasure is not currently implemented in Hypernel, the previous work [6, 21] shows that such a malicious attempt can be easily circumvented by leveraging IOMMU[3]. Furthermore, since our MBM can watch the bus traffic between the CPU and main memory, we expect that Hypernel can detect such an attack with additional engineering efforts.

[2]These values are relevant to Linux user credential and Virtual File System, respectively. Modifying the `cred` structure allows the attacker to elevate any process to have `root` permission, while seizing the control of a `dentry` enables the attacker to access its inode and manipulate it so that operations against the directory and its contents are not working correctly.

[3]IOMMU is called as System MMU in the ARM architecture.

9 CONCLUSION

This paper presents Hypernel, whose goal is to protect the kernel without nested paging. To attain this goal, Hypernel comes with Hypersec, the privileged software that with being assisted by hardware virtualization, efficiently provides an isolated execution environment for security appliances and monitors. Hypernel is also armed with MBM, the special hardware module that enables a word-granularity access control, thereby eliminating the redundant overhead which ordinary security solutions with nested paging should bear. The evaluation results confirm the effectiveness of Hypernel as a security framework in terms of performance.

REFERENCES

[1] Xen: Vulnerability statistics. http://www.cvedetails.com/vendor/6276/XEN.html.

[2] Linux Linux Kernel : CVE security vulnerabilities, versions and detailed reports, 2017.

[3] ARM. Logictile express 20mg daughter board.

[4] ARM. Versatile express juno r1 development platform. In *ARM 100122_0100_00_en*, 2015.

[5] A. M. Azab. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, pages 90–102, 2014.

[6] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[7] Y. Cho, D. Kown, H. Yi, and Y. Paek. Dynamic virtual address range adjustment for intra-level privilege separation on arm. 2017.

[8] Y. Cho, J.-B. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *USENIX Annual Technical Conference*, pages 565–578, 2016.

[9] C. Dall, S.-W. Li, J. T. Lim, J. Nieh, and G. Koloventzos. Arm virtualization: performance and architectural implications. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 304–316. IEEE Press, 2016.

[10] C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

[11] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 50, pages 191–206, 2015.

[12] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 487–502, New York, NY, USA, 2015. ACM.

[13] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proceedings of the Mobile Security Technologies 2014 Workshop*, 2014.

[14] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 279–290, New York, NY, USA, 2011. ACM.

[15] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang. Atra: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 167–178, New York, NY, USA, 2014. ACM.

[16] L. Koromilas, G. Vasiliadis, E. Athanasopoulos, and S. Ioannidis. GRIM: Leveraging GPUs for Kernel Integrity Monitoring. In *Proceedings of the 19th International Symposium of Research in Attacks, Intrusions and Defenses*, pages 3–23, 2016.

[17] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 511–526, Washington, D.C., 2013. USENIX.

[18] R. Mijat and A. Nightingale. Virtualization is coming to a platform near you. *ARM white paper*, 20, 2011.

[19] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[20] N. L. Petroni Jr and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115. ACM, 2007.

[21] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[22] A. Srivastava and J. Giffin. Efficient Protection of Kernel Data Structures via Object Partitioning. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 429–438, Orlando, Florida, USA, 2012.

[23] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.

[24] E. Unified. Unified extensible firmware interface specification. *Version*, 2:1827–1882, 2014.

[25] A. Vasudevan, S. Chaki, Limin Jia, J. McCune, J. Newsome, and A. Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 430–444. IEEE, May 2013.

[26] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou. Secpod: a framework for virtualization-based security systems. In *USENIX Annual Technical Conference*, pages 347–360, 2015.

[27] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554. ACM, 2009.

[28] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[29] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 457–468. IEEE, 2014.