Architectural Supports to Protect OS Kernels from Code-Injection Attacks

Hyungon Moon, Jinyong Lee, Dongil Hwang,
Seonhwa Jung, Jiwon Seo, Yunheung Paek
ECE and ISRC, Seoul National University
{hgmoon, jylee, dihwang, shjung, jwseo, ypaek}@sor.snu.ac.kr

ABSTRACT

The kernel code injection is a common behavior of kernel -compromising attacks where the attackers aim to gain their goals by manipulating an OS kernel. Several security mechanisms have been proposed to mitigate such threats, but they all suffer from non-negligible performance overhead. This paper introduces a hardware reference monitor, called Kargos, which can detect the kernel code injection attacks with nearly zero performance cost. Kargos monitors the behaviors of an OS kernel from outside the CPU through the standard bus interconnect and debug interface available with most major microprocessors. By watching the execution traces and memory access events in the monitored target system, Kargos uncovers attempts to execute malicious code with the kernel privilege. According to our experiments, Kargos detected all the kernel code injection attacks that we tested, yet just increasing the computational loads on the target CPU by less than 1% on average.

1. INTRODUCTION

These days, more and more attackers endeavor to compromise an OS kernel on which most of the applications in a system rely. Manipulating the kernel, attackers are capable of affecting the kernel's behavior in almost all aspects, such as the way how kernel objects are accessed, what data is sent through network, or what permission a file is accessed with. Unfortunately, like other programs, OS kernels could have vulnerabilities with which an adversary can acquire the capability to access their memory arbitrarily. For example, the Linux kernel, which is the kernel of the most dominant operating system in the smartphone market [1], is considered to have unknown vulnerabilities in that new ones are reported every year [2, 3]. Although they have been patched already, the adversaries would exploit a new one that is not published yet, to compromise a fully patched system.

A powerful way to compromise a victim kernel with the capability is the *code-injection attack*, so several mechanisms have already been proposed to detect it with architectural supports. With the access to the kernel memory, attackers can deceive the victim kernel into executing malicious code by placing it in a kernel memory region, corrupting some function pointers and manipulating the page table. To help OS kernels in mitigating the attack, modern processors are equipped with architectural supports, such as *Supervisor Mode Execution Prevention* (SMEP) [4] or *Privileged eXecute Never* (PXN) [5]. These supports add a field in page table entries, and make the Memory Management Unit (MMU) use the field to decide if an instruction can be executed with the kernel privilege or not. Utilizing these, several mechanisms have successfully detected the attacks by protecting the integrity of the page tables containing the configurations [6, 7, 8, 9, 10].

Although these mechanisms have successfully defeated the code injection attacks, they inevitably introduce non-negligible performance overhead. In order to detect the code-injection attacks with the new field in page table entries, they should mediate all updates to the page tables and ensure that only the pages with the legitimate kernel code are configured to be executable in the kernel mode. If they omit a single entry, an attacker can corrupt it to inject the malicious code into the kernel by marking it to be executable with kernel privilege and map the page to a physical memory region containing the malicious code.

In this paper, we present Kargos, a hardware-based reference monitor that detects the code-injection attacks without mediating the accesses to the entire page table. Instead of marking each page with its permission, Kargos examines the target addresses of indirect branch instructions to detect the first control-flow transfer to a malicious code block, while the CPU runs in the kernel mode. In this way, the monitor can ensure that the kernel never executes with the kernel privilege the instructions from outside the predefined kernel code pages. In addition, Kargos checks if the virtual pages of the kernel code regions are mapped to the corresponding physical code regions correctly. Otherwise, the attacker would be able to remap the kernel code pages into a physical memory region filled with the malicious code [11]. Combining these two, the monitor can detect any execution of an instruction that is not fetched from the legitimate kernel code region while the CPU is in the kernel mode. For the sake of explanation throughout this paper, we hereafter refer to the virtual pages storing the kernel code as the *virtual code regions* and the corresponding physical memory regions storing the kernel code as the *physical code regions*.

Even though Kargos needs to examine the target addresses of the indirect branch instructions, it is not necessary to install our Kargos inside the CPU core. Instead, our prototype is placed outside the CPU and acquires the values from the *Program Trace Interface* (PTI), which most modern CPU possess [12, 13]; in order to help debugging and profiling programs, the interface can be configured to continuously emit a stream of packets. Parsing them, Kargos can incessantly observe the target addresses of indirect branches.

Kargos can also secure the translations of the kernel code ad-

HASP 2016, June 18 2016, ,

ISBN 978-1-4503-4769-3/16/06. . . $15.00

DOI: http://dx.doi.org/10.1145/2948618.2948623

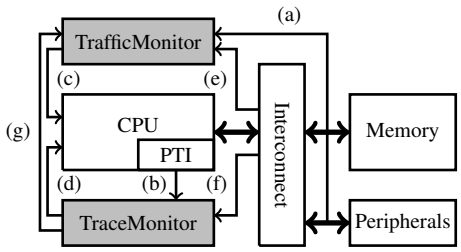

Figure 1: This figure shows how the hardware modules are connected to the other modules in the system. TrafficMonitor can examine the accesses to the memory or the memory mapped peripherals with the connections (a), and TraceMonitor is fed with the PTI packets through (b). Once one of them recognizes any violation, they interrupt the CPU to deal with the violation with (c) and (d). The CPU can access the hardware modules through (e) and (f), but the modules would accept these accesses selectively. TraceMonitor forwards some indirect branch target addresses to TrafficMonitor through (g).

dresses without modifying the CPU core. CPUs usually use the values of some special registers and the contents of some page table entries for address translations. For instance, ARM processors have *Translation Table Base Registers* (TTBRs) that contain the base address of page global directory, and several entries of the directory are used for translating the kernel code pages. For this reason, it is enough to protect these values to ensure that the address of kernel code pages are translated correctly. To mediate the modifications to the special registers, we first add the instructions that check the correctness of the updated values, to the code blocks which update the registers to include. In addition, Kargos ensure that these code blocks are always executed as designed, by checking if they are executed atomically. Using the outputs of PTI, Kargos can enforce some code blocks, including the ones to update the special registers, to be executed atomically. At last, such code blocks can also be designed to notify Kargos of the new address of the page global directory. With the value, Kargos can always monitor every access to the directory entries for the kernel code pages with *bus snooping* mechanism [14, 15].

To show the effectiveness and efficiency of Kargos, we have completely constructed its prototype in hardware to monitor the Linux kernel of Android 4.2.2 that runs on an ARM-based system. According to our experiments, Kargos has caught all the kernel code injection attacks that we tested, yet incurring only about 1% performance overhead than the original system being left vulnerable to the attacks. In addition, all components of our prototype are implemented in a physically secure hardware platform operating independently of the target system such that Kargos can work as planned even if we assume that an adversary is able to access any memory regions in the target system including the kernel code and page tables.

The rest of this paper is composed as follows. In Section 2, we describe the threat model that Kargos aims to mitigate and the assumptions. On top of these, we present the design of Kargos in Section 3 and evaluate our mechanism in Section 4. After discussing the limitations and future work in Section 5, we compare our work with the related work in Section 6 and conclude this paper in Section 7.

2. PROBLEM DEFINITION

This section describes the threat model and the assumptions for Kargos, in order to define the scope of this work.

2.1 Threat Model

In this work, we consider adversaries who inject their code and hijack the kernel control-flow to execute the injected code in the privileged mode. The adversaries are hereby assumed to know of an OS kernel vulnerability (e.g., CVE-2014-3153 [2]) which they can exploit to access the victim's memory arbitrarily. With this capability, they are able to put their own code into the kernel memory and redirect a control-flow of the kernel to the code. The OS kernel may try to defeat the attack using architectural support like PXN or SMEP, but such powerful adversaries can circumvent them by overwriting the page table entries, unless the entire page table is protected from such a corruption.

On the other hand, we assume that adversaries do not have physical access to a victim machine and the machine contains no malicious hardware. In other words, we rule out any kind of physical attacks as most previous work on kernel-independent security solutions do. On top of the assumptions about benign hardware, we add one more that our target system employs *secure boot* (or *trusted boot*) [16] to load the correct OS kernel image at bootstrap. Thanks to the secure boot, we assert that Kargos can safely collect, before the OS starts, the information about the kernel necessary for its monitoring job.

2.2 Assumptions

Kargos does not require the target system to have another privilege higher than the OS kernel for virtualization support, or the special CPU architecture for harboring the secure world. However, it still has some requirements. First of all, the target system CPU is assumed to have PTI, which in fact corresponds to the *program trace macrocell* (PTM) in ARM processors [12] or the *processor trace* in Intel x86 processors [13]. Luckily, modern processors today normally employ such hardware debug features, so we deem that this is a reasonable assumption. In addition, the target addresses coming out of PTI are assumed to be virtual addresses, which is indeed true for most PTIs in real machines. Lastly, the CPU may have a capability of controlling the interface but only by either executing some special instructions or accessing memory-mapped registers of the interface.

3. DESIGN

To detect the kernel code injection attacks, Kargos watches the memory traffic and the PTI outputs to examine the memory accesses and the execution traces of the target system. In addition, the target system kernel is augmented to notify some events, such as special register updates or mode switches, and Kargos ensures that the augmented code blocks are executed correctly by checking if they are executed atomically or not. For the rest of this section, we first briefly describe the architectural supports which Kargos is equipped with, then more details will follow as to how Kargos detects the kernel code injection attacks with these supports.

3.1 Architectural Supports

As shown in Figure 1, Kargos is composed of two modules, *TraceMonitor* and *TrafficMonitor*. Each module has its own memory-mapped control interface connected to the CPU through the interconnect. At boot time, the kernel can initialize the modules through the interfaces, and can also pass various information such as the special register values during runtime. Nevertheless, no attacker can corrupt the configurations of the modules because each interface can be locked at boot time. Once locked, no software component running on the CPU can unlock them unless the entire system reboots. Because a reboot would make the system load a clean kernel image, an attacker would not be able to corrupt the configurations of the modules.

3.1.1 Trace Monitoring

The first architectural support is the *trace monitoring*, which enables Kargos to examine the indirect branch target addresses. Being originally designed for an external debugger to completely reconstruct the execution flow, PTI provides packets of information about the executeion trace. Specifically, an external debugger can calculate the target addresses of the indirect branches from the packets. As mentioned in Section 1, Kargos only needs the target addresses of indirect branch instructions, as the targets of direct branches can be statically analyzed and acquired. For this reason, TraceMonitor parses only the packets required to calculate the indirect branch target addresses.

Figure 2 shows the main components of TraceMonitor. The *Packet Parser* generates a stream of indirect branch target addresses from the outputs of PTI, and the other four components take this stream as an input and examine it. Among the components, *Boundary Checker* is what detects the jump to the malicious code while the CPU is in the kernel mode. It has a set of registers that contain the addresses of bases and bounds of the kernel code regions. By comparing the incoming indirect branch addresses with these values, TraceMonitor can determine if the CPU tries to run code fetched from outside the virtual code regions.

The second module is the *Mode Tracker*, which is mainly responsible for recognizing the mode switches. Located outside the CPU, neither TraceMonitor nor TrafficMonitor can directly access the CPU to acquire the mode information. For this reason, we implemented our system in a way that TraceMonitor can track the CPU mode switches using some special execution traces. More details about how we could generate such special execution traces securely will be presented in Section 3.2.2. From the special traces, TraceMonitor can acquire the mode information securely.

The third module is *Reporting Helper*. To help TrafficMonitor to distinguish malicious *reports* of special register updates from the benign ones, this module selectively forwards some indirect branch target addresses to TrafficMonitor. In specific, the module forwards the target addresses of jumps to and from the code blocks which update the special registers for the address translations. Section **??** describes how TrafficMonitor uses these addresses to handle the incoming reports.

The last module of the TraceMonitor is *Atomicity Checker*, which enables the kernel to implement *Atomic Code Blocks*. Once atomic code blocks are realized in the kernel, this module is configured with the location information of the atomic code blocks, and checks if the CPU jumps to the middle of an atomic code block. Unlike the Boundary Checker, this module is not configured with the range of each code block mainly due to the scalability; the kernel can be built to be composed of a small number of code regions without difficulty, but could have an arbitrary number of atomic code blocks. For this reason, the Atomicity Checker uses (1) the ranges of regions containing the atomic code blocks and (2) the alignments to define the locations and boundaries of the code blocks. For each incoming indirect branch target address, Atomicity Checker first checks whether the target of the jump is in one of the memory pages containing the atomic code blocks not. If it is a jump to an atomic code block, Atomicity Checker checks if the address is correctly aligned, to prevent the CPU from jumping to the middle of an atomic code block. For instance, if the size of atomic code blocks in the region is 256 bytes, Atomicity Checker would check whether the lower eight bits of the target address are zero or not. In this way, the kernel can define a set of memory regions containing atomic code blocks and Atomicity Checker can ensure that the code blocks are executed atomically.

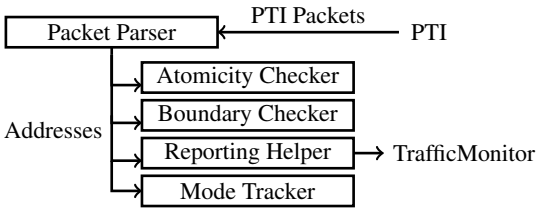

Figure 2: This figure shows the main components of TraceMonitor. The components that forward the addresses from PTI to TraffcMonitor and that generate an interrupt are omitted, and the functionality of each component is described in Section 3.1.1.

3.1.2 Traffic Monitoring

The second support for Kargos is the *traffic monitoring*, with which it can detect the malicious modifications to the kernel memory. Following the design presented in earlier work [14, 15], TrafficMonitor snoops the traffic between the *interconnect* and the memory, to detect any write accesses to the main memory, as depicted in Figure 1. Because any write accesses from the CPU to the caches make the corresponding cache line dirty and the dirty line will eventually be evicted to the memory, attackers cannot modify the memory contents without generating the corresponding traffic between the interconnect and the memory. In addition, because it is implemented in hardware, TrafficMonitor never misses any of the traffic. Note that TrafficMonitor can examine the memory accesses from the peripherals as well, including the Direct Memory Access (DMA), as there is no way to access the memory without using the link between the memory and the interconnect.

To define the protected memory regions, TrafficMonitor has a set of base and bound registers. For each memory access to the main memory, TrafficMonitor uses the values of these registers to check if the access is a write to the protected region. For instance, the registers should contain the bases and bounds for the physical code regions to detect the malicious modifications to the kernel code including the atomic code blocks for special register updates, and those of the page global directory entries for the virtual code regions to protect them from being remapped. While the registers for the kernel code regions can be configured at boot time, the others should also be updated at runtime using the reports from the kernel because it changes the active page table by updating the corresponding special registers in the CPU.

In addition to the memory protection, TrafficMonitor is also capable of distinguishing fake reports from the genuine ones, which should be generated by a particular code block. Otherwise, the attackers would be able to generate a fake report to deceive Kargos into protecting a wrong memory region. As they are assumed to have an arbitrary access to the kernel memory, they may also be able to alter the values of the control registers in the devices. If so, such attackers would be able to deceive our system without difficulty, by accessing the control registers for the reports.

To ensure that a report is generated by the corresponding code block, TrafficMonitor generates a nonce which the code block should include as a part of the report. In addition, TrafficMonitor updates the corresponding registers with the incoming report only when it recognizes the execution of the corresponding code block, through the indirect branch target addresses from TraceMonitor. If the code block is designed to raise an alarm when it fails to fetch the nonce, attackers would not be able to deceive TrafficMonitor into updating the bases and bounds of the protected region with a fake report. Section 3.2.1 includes how Kargos uses this protocol and how the kernel implements a correct code block to generate the report.

3.2 Detection Mechanism

This section describes how Kargos detects the code-injection attacks using these architectural supports, beginning with the four rules that the system should comply with at runtime and that Kargos checks to detect the attacks.

- **R1.** The physical code regions of the kernel should never be modified.
- **R2.** The CPU jumps to an address in the virtual code regions when it enters privileged mode.
- **R3.** All the targets of indirect jumps lie in the virtual code regions while the CPU is in privileged mode.
- **R4.** All of the virtual code regions are mapped to the physical code regions. In other words, the CPU translates an address of a virtual code region into an address of a physical code region.

The target system CPU would not execute in privileged mode the injected code as long as it complies with these four rules. Owing to **R3**, if the CPU executes an instruction from the virtual code regions in privileged mode, the next one shall also be from the region. As it is possible to statically calculate all of the target addresses of direct branches and examine the instructions at the boundaries of the kernel image, we can ensure the following; if the CPU executes an instruction from the virtual code regions that is not an indirect branch, then the next instruction will be fetched from the virtual code regions. As the CPU must fetch the first instruction from the virtual code regions when entering privileged mode (**R2**), the CPU cannot execute any instruction from outside the virtual code regions when in privileged mode. Consequently, to execute injected code without violating **R2** and **R3**, an attacker should either make changes to the physical code regions to which the virtual ones are mapped, or remap a virtual page in the virtual code regions into an arbitrary physical page, which contains the injected code, that is outside the physical code regions. As the former violates **R1** and the latter does **R4**, we can conclude that it is not possible to perform a kernel code injection attack that complies with all the rules at the same time. Kargos can therefore recognize any sort of code-injection attacks if it is capable of detecting any attempt to breach the rules.

3.2.1 Securing the Kernel Entrance

To secure kernel entrances (**R2**), it is sufficient to ensure that the target system CPU jumps to the *gateway* code blocks whenever it enters privileged mode. Designed to be executed at that moment, these code blocks preserve the CPU states of the applications, examine the status registers to determine the reasons for mode switches, and establish the execution environment for the kernel. To calculate the addresses of the gateway code blocks, CPUs usually use the values of certain special registers, which we call *gateway registers*. For example, the Linux kernel for ARM CPUs has six gateway code blocks to handle system calls, interrupts and exceptions. To calculate the addresses of the blocks, the ARM CPUs use the values of the *system control register* (SCTLR) and the *vector base address register* (VBAR).

Kargos protects the gateway register values, which the attackers should modify in order to deceive the CPU into jumping to malicious code when it enters privileged mode. As the user applications should not be able to modify these values, the kernel in general has to execute certain special instructions, which can only be executed in privileged mode. To update the gateway registers of the ARM CPUs, for example, the kernel should execute the `MRC` instructions with some predefined operands [5]. If we can force the kernel to regulate all executions of the special instructions, attackers would not be able to corrupt the gateway registers. We augmented the kernel in a way that checks all the operands in the special instructions prior to its execution. This is done by executing the checking instructions before the execution of the special instructions.

```
1    sub   pc, pc, #4
2    check_entries r0
3    bne   handler
4    ldr   r4, nonce_addr
5  wait_for_nonce:
6    ldr   r5, [r4]
7    cmp   r5, #0
8    beq   wait_for_nonce
9    str   r5, [r4]
10   str   r0, [r4]
11   dsb   sy
12   mcr   p15, 0, r1, c13, c0, 1
13   isb
14   mcr   p15, 0, r0, c2, c0, 0
15   isb
16   pop   {r4, r5, r6, r7}
17   orr   lr, lr, #0xc
18   bx    lr
```

Figure 3: This figure shows how the kernel follows the reporting protocol. Before checking the entries, the kernel executes an indirect branch instruction with which TrafficMonitor recognizes the execution of this block. If the checking operation fails, the kernel invokes a predefined handler at line 3. From line 6 through line 8, the kernel waits for a non-zero nonce to become available, writing the nonce and the report the address of the new page global directory and the nonce to the corresponding control register.

However, the kernel cannot check all of the executions without additional hardware support, as it cannot prevent the CPUs from jumping to a particular address, especially to the special instructions. While the CPUs usually allow the kernels to disable exceptions or interrupts for atomic operations, they do not provide a means to prevent control-flow transfers to a particular set of addresses. If the kernel does not prevent these jumps to the special instructions, an attacker would be able to bypass the checking instructions and execute the special instructions, even if the checking instructions are directly followed by the special instruction. To deal with this type of conceivable attack, Kargos uses the architectural support for the atomic code blocks, which is presented in Section 3.1.1. With the support, Kargos can guarantee that the special instructions always follow the checking instructions directly.

3.2.2 Detection of the Malicious Jumps

In order to recognize violations of **R3**, Kargos should be able to examine the indirect jumps. This is achieved with the help of TraceMonitor. At boot time, the kernel configures TraceMonitor with the ranges of the virtual code regions so that it can distinguish jumps to these code regions from the ones to the other regions. However, TraceMonitor must also determine whether the CPU is in privileged mode or not so that it can ignores all jumps when the CPU runs in the user mode. As presented in Section 3.1.1, TraceMonitor is able to recognize the mode switches, if the kernel follows a special control flow when and only when the CPU enters or exits privileged mode. From such distinguishing traces of indirect jumps, Mode Tracker would be able to recognize the mode switches. Given that the kernel has already been enforced to execute the gateway code blocks when it enters the privilege mode, Mode Tracker considers the jumps to those code blocks as signatures of mode switches from the user to privileged mode.

Similarly, OS kernels in general have several special code blocks that restore the CPU states of user applications and switch the CPU mode to the user mode. However, jumps to these *exit* code blocks

```
1  msr      SPSR_fsxc, r1
2  and      r3, r1, #31
3  cmp      r3, #16
4  subeq    pc, pc, #4
5  restore_context
6  movs     pc, lr  ; this may switch the mode
```

Figure 4: This shows how the kernel is modified to generate a special control-flow in an exit code block. After updating the SPSR with the value in `r1`, we added three instructions to check the value in `r1` and generate an indirect jump if the mode field is set to the user mode.

cannot be the signs of mode switches from privileged to user because the execution of the block may not always cause such a mode change. For example, the exit code blocks of the Linux kernel for ARM CPUs use `movs` instructions to switch the CPU mode to the user mode only when the mode field of the *saved program state register* (SPSR) is set to user. If not, the `movs` instruction does not change the CPU mode to user. For this reason, we augmented the kernel to follow a special control flow only when it returns to the user mode, whereas it follows the original control flow otherwise. In detail, the kernel checks the value of the SPSR before executing the `movs` instruction and executes an additional indirect jump instruction only when the mode field of the SPSR is set to user. This additional instruction generates a trace that Mode Tracker can consider as the sign of the mode change event. With this additional signature, Mode Tracker can recognize the mode switch from privileged to user. Figure 4 shows how the signature for a Linux kernel running on an ARM processor is generated.

3.2.3 Protection of the Mappings

Although we could avoid protecting all page tables in the target system, Kargos has to protect the page table entries which map the virtual code regions into the physical code regions, due to the *Address Translation Redirection Attack* (ATRA) [11]. To protect these entries (**R4**), we make use of TrafficMonitor to examine every access to the entries. However, examining all these entries in the target system is not desirable as most systems maintain a set of such entries for each process. Instead, Kargos protects only the entries of the page tables currently in use. In specific, the kernel checks the page table entries of a process whenever it becomes active. The kernel can check all such events by executing the checking instructions before executing the special instruction to update the active page table. Using the hardware support for atomic code blocks, we can also ensure that the kernel never uses a page table without checking the mappings from virtual into physical code regions. In the atomic code blocks that update the registers, the kernel also provides TrafficMonitor with the physical addresses of the new entries that should be protected from the modifications. In addition, the code block is implemented to comply with the protocol presented in Section 3.1.2 to be resilient to the fake reporting attack. Figure 3 shows how the code block generates a report which TrafficMonitor would accept as a genuine one.

3.2.4 Code Protection

To detect violations of **R1**, the kernel should examine all memory accesses to the physical code regions. Although kernels can rely on the MMU to examine the accesses and detect malicious modifications, this requires the kernel to have the means to protect the integrity of all page tables in the system. Otherwise, attackers would corrupt the page tables to deceive the MMU and modify the physical code regions without being detected [6, 8]. Kargos does not require the entire page tables to be protected, as it can detect the write accesses to the physical code regions with their physical addresses, using TrafficMonitor.

Name	Baseline	Kargos
null syscall	0.98μs	1.07μs (0.92%)
open/close	18.39μs	18.15μs (-1.28%)
select	4.58μs	4.57μs (-0.11%)
sig. handler install	2.81μs	2.82μs (0.11%)
sig. handler overhead	9.91μs	10.55μs (6.42%)
pipe	40.89μs	43.23μs (5.72%)
fork+exit	2853.15μs	2838.60μs (-0.51%)
fork+execve	9279.8μs	9159.16μs (-1.3%)
page fault	4.34μs	4.45μs (3.63%)
mmap	84.7μs	84.9μs (0.24%)

Table 1: LMBench results

4. EVALUATION

To evaluate the effectiveness and the efficiency of Kargos, we have implemented a full-system prototype on the Xilinx ZC 702 evaluation board which includes Xilinx Zynq Z-7020 [17], on which an ARM-based system can be developed. the hardware modules are developed in Verilog HDL and mapped on the FPGA. Since the target system employs ARM NIC-301 AXI network interconnect, all the modules in Kargos are also designed to comply with the corresponding ARM AMBA 3.0 specification. Mainly due to the speed limit of FPGA, we configured Kargos to operate at 80 MHz, and also scaled down the clock speed of the target system to 222 MHz, complying with the performance ratio between the host and the co-processors in most SoC platforms such as *application processors* of modern smartphones [18].

On top of this SoC, we ran Android 4.2.2 with the Linux kernel 3.8.0 from the iVeia's git server [19] as the operating system. While we used the Android framework as it is, we modified the Linux kernel in order to implement the atomic code blocks for the special instructions. Specifically, we enclosed and relocated two types of instructions to secure the kernel entrances and the address translations, as presented in Section 3.2.1 and Section 3.2.3, respectively. To secure the entrances, we created six atomic code blocks for the instruction modifying SCTLR. As the baseline system does not use VBAR to calculate the addresses, we did not need to protect it from malicious modifications. To protect the mappings, we enclosed four instructions modifying *Translation Table Base Register* (TTBR), which contains the base address of the page global directory in use. Because the kernel does not contain any special instructions for accessing PTM, we did not consider the case.

4.1 Performance

To evaluate the performance overhead that Kargos introduces, we initially used LMBench [20] to measure the performance of the operating system services. We used the script that is included in the benchmark suite to assess the performance impact on the latencies of the operating system services, as shown in Table 1. The reported values are the averages of 10 runs. As shown in the table, the performance impact of Kargos was negligible.

In addition to the microbenchmarks, LMBench, we also ran SPECint 2006 to evaluate the impact of our scheme on user-level applications. As shown in Table 2, the performance impact on SPEC is negligible. This suggests that our scheme would not degrade CPU-intensive workloads running as user-level applications.

Lastly, we ran five real-world Android benchmarking applications as shown in Table 3. All the results presented here represent the average values over 10 runs. As can be seen in the table, Kargos places less than 1% computational loads on average upon the target system, thanks to our architectural supports.

4.2 Security

As described in Section 2, we consider an attacker who exploits

Name	Baseline	Kargos
400.perlbench	12097.99s	12121.52s (0.19%)
401.bzip2	7284.54s	7274.29s (-0.14%)
403.gcc	2420.82s	2429.91s (0.38%)
445.gobmk	13412.38s	13542.57s (0.97%)
456.hmmer	15327.28s	15385.06s (0.38%)
458.sjeng	17000.11s	17051.94s (0.3%)
462.libquantum	42659.18s	42753.94s (0.22%)
464.h264ref	18785.86s	18841.65s (0.3%)
471.omnetpp	10334.19s	10382.46s (0.47%)
473.astar	7717.71s	7684.35s (-0.43%)
483.xalancbmk	11235.73s	11257.41s (0.19%)

Table 2: SPECint 2006 results. Among twelve benchmarks in SPECint 2006, our baseline system could not run 429.mcf due to the lack of memory.

Name	Baseline	Kargos
RL	607.90	610.82 (0.48%)
CF-Bench	531.80	527.80 (0.75%)
GeekBench	67.20	67.00 (0.30%)
Linpack-single	9.01	8.96 (0.64%)
Vellamo-metal	121.80	121.40 (0.30%)

Table 3: Android benchmark results.

a vulnerability of the kernel to acquire the capability of accessing the victim system memory arbitrarily in order to perform the kernel code injection attacks. For subverting our target system, such attackers can exploit a real-world vulnerability of Linux kernel, which has been reported as CVE-2014-3153 [2]. To test the effectiveness of Kargos, we wrote three Proof-of-Concept (PoC) attacks exploiting the vulnerability because we could not find any publically available kernel code injection attack on ARM-based systems leveraging the vulnerability. The first attack aims to execute its own code through modifying the physical memory regions, and the second one writes to its own page table to remap the virtual code regions. As both of them generate write attempts to the memory regions that the TrafficMonitor is monitoring, Kargos was able to detect the attacks without difficulty. The last attack is designed to hijack the kernel execution flow to already-injected code without modifying the kernel code regions or the page tables for the virtual code regions. As the hijacking inevitably causes a jump to addresses outside the virtual code regions, the TraceMonitor raises an alarm when it receives the trace that corresponds to the jump from the PTI. It is worth noting that the only difference between our PoC attacks and the publically available examples of kernel rootkits on *packetstormsecurity.com*, which inject their code to the kernel, are the way how they achieve the capability of accessing the kernel memory. While the public examples are implemented as kernel modules to manipulate the kernel code and data, ours exploit a real-world vulnerability of the kernel to achieve the capability.

5. LIMITATIONS AND FUTURE WORK

Code Reuse Attacks. While code injection has been a common way of control-flow hijacking, it has been lately discovered [21, 22] that adversaries can hijack the kernel execution flow without executing injected code in privileged mode at all. Instead, the adversaries can try a different breed of attacks, known as *code reuse attacks* (CRAs), where they reuse a set of kernel code snippets to implement the functionality they want, and redirect the execution flow to their chain of code snippets. Being designed to detect the execution of injected code, Kargos is not capable of catching CRAs. However, this limitation should not undermine the efficacy of Kargos for two reasons. First, the attackers cannot undermine Kargos even if they have successfully executed their CRA payload. Second, Kargos would be able to combined with the previously proposed mechanisms to mitigate the CRAs using the trace interfaces [23, 24]. While these mechanisms rely on the OS kernel to make use of the PTIs, Kargos would enable them to use the PTIs without relying on the kernel and to monitor the traces of the kernel.

Kernel Modules. Modern kernels are allowed to load the *kernel modules* to extend their code at runtime. Although the current design of Kargos assumes that the code regions remain unchanged, the design can be enhanced to allow the target kernel to load the modules. In addition, it would be possible for the enhanced Kargos to decline an attempt to extend the kernel with a maliciously crafted module as long as Kargos has a set of cryptographic hashes of the known good modules. In order to load a kernel module, the kernel should send a request to Kargos to adjust the current virtual code regions, physical code regions and page tables. Otherwise the execution of the extended code will raise an alarm. Upon receiving the request, Kargos will firstly include the module image to the physical code regions to detect the attempt to modify the image. Consequently, the image becomes neither writable nor executable during the verification process through the hash. Once the module is confirmed to be a known-good one, Kargos then checks the image again to find if it contains the special instructions. After this verification process, Kargos can safely extend the virtual code regions to permit the CPU to execute the kernel module.

6. RELATED WORK

Page Table Protection As mentioned in Section 1, several mechanisms [6, 7, 8, 9, 10] have been proposed to protect the page table utilizing the recent hardware supports such as PXN or SMEP. By protecting the page tables, they could detect the kernel code injection attacks effectively, but they inevitably introduced non-negligible performance overhead. Compared to these mechanisms, Kargos can defeat the attacks with smaller performance overhead, by avoiding the protection the entire pare tables.

Hypervisor-based Approaches With the higher privilege than the OS kernels, hypervisors in general are capable of investigating the kernel memory and intervening the kernel events. Using these capabilities, many mechanisms have been proposed to protect the kernel with the hypervisors. Among them, SecVisor [25] and NICKLE [26] are closely related to our work in that their main goals are also to detect kernel code injections. SecVisor protected and augmented the page tables whenever the CPU enters or exits the kernel to simulate the PXN/SMEP, as those supports were not available then. NICKLE emulated the *Havard Architecture*, where the code memory and data memory are strictly separated, by intervening the instruction fetches. Due to the lack of hardware supports, these mechanisms have higher performance overhead when compared with Kargos or the recent mechanisms using PXN/SMEP.

Snapshot Analyses Another direction of the mechanisms to protect the kernel has been the *Snapshot Analysis*. Using either a dedicated hardware or hypervisor, it is possible to implement a monitor which acquires the snapshots of the kernel memory periodically and analyzes them to detect the anomalies. The earlier ones have focused on protecting the kernel code from being corrupted [27], but recent ones have checked the *Control-Flow Integrity* (CFI) [28, 29], by examining the function pointers in the kernel memory. These mechanisms would be able to detect the code-injection attacks if they corrupt the function pointers, but very recent work [30] suggested that a *dynamic hook*, which hijacks the execution flow without persistently altering any control data,

would successfully evade these control data detection schemes. Although OSck [28] has a potential to detect dynamic hooks, they did not present a specific scheme that can handle all kinds of dynamic hooks on the kernel.

Bus Snooping As mentioned in Section 3.1.2, the idea of bus snooping is not new and already proposed as a means to watch the accesses to the kernel memory [14, 31, 32]. However, they have focused only on the detection of the malicious modifications to the kernel memory, and could not detect the code-injection attacks effectively. In particular, they could detect code injection attacks that corrupt the physical code regions or some function pointers that they are monitoring. Unlike these mechanisms, Kargos can detect any type of kernel code injection attacks.

Debug Interfaces The idea of using the debug interface (i.e., PTI in our work) for different purposes other than its intended usage has already been introduced and explored in several other studies, especially in the field of fault-tolerant computing [33, 34]. Fidalgo et al. [33] proposed a mechanism that can inject faults to the target system by accessing internal resources such as registers and memory via the interface. Another study presented an on-line fault detection technique that utilizes the interface to retrieve runtime information in a non-intrusive way [34]. Lately Lee et al. [35] discussed how the interface can be used to enhance the security of a system. They used the interface to transfer the internal cache data from the CPU into their external security device for memory traffic monitoring. Lastly, as mentioned in Section 5, the mechanisms to detect CRAs using the PTI outputs have also been proposed [23, 24]. These studies are conceptually similar to ours in that they exploit information flow that comes out of the debug interface without affecting the architecture of the CPU. However, none of these suggested the use of the interface to protect the OS kernel against the code injection attacks.

7. CONCLUSION

This paper presented Kargos, the first mechanism to detect all kernel code injection attacks without mediating all accesses to the page tables. While existing solutions either detect only certain forms of the attacks or consume considerably the computing resources on the CPU, Kargos detects all the attacks with nearly zero performance cost. In our design, we provide mechanisms, called the atomic code blocks and the reporting protocol, for Kargos to protect the integrity of special register in the CPU and to extract the values of them without relying on the service of the OS kernel. Unlike the hypervisor or any other software that runs on the CPU, our external hardware components technically has no means to access these registers directly.

8. ACKNOWLEDGEMENTS

This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. R0190-16-2010, Development on the SW/HW modules of Processor Monitor for System Intrusion Detection), the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A1A10051792), the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2016-R0992-16-1006) supervised by the IITP(Institute for Information & communications Technology Promotion), and the Brain Korea 21 Plus Project in 2016. This reseach is also supported by Software R&D Center of Samsung Electronics.

9. REFERENCES

[1] I. IDC Research, "Smartphone os market share, 2015 q2." http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2015.

[2] "Cve-2014-3153." Online, May 2014. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3153.

[3] "Cve-2015-3636." Online, May 2015. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3636.

[4] I. INC., "Intel r 64 and ia-32 architectures software developer's manual," *Volume 3b: System Programming Guide (Part 2)*, 2013.

[5] A. R. M. (ARM), *ARM Architecture Reference Manual, ARM v7-A and V7-R edition, Tech. rep.* ARM, 2012.

[6] 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 SIGSAC Conference on Computer and Communications Security*, 2014.

[7] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," in *IEEE Mobile Security Technologies Workshop*, 2014.

[8] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Architectural Support for Programming Languages and Operating Systems*, ACM, 2015.

[9] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm," in *Network & Distributed System Security Symposium (NDSS)*, 2016.

[10] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou, "Secpod: a framework for virtualization-based security systems," in *Usenix Annual Technical Conference*, USENIX Association, 2015.

[11] 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 *ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014.

[12] ARM, *CoreSight PTM-A9 Technical Reference Manual*, 2011.

[13] Intel, *Intel64 and IA-32 Architectures Software Developer's Manual*, 2014.

[14] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: toward snoop-based kernel integrity monitor," in *CM conference on Computer and communications security*, ACM, 2012.

[15] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, "Cpu transparent protection of os kernel and hypervisor integrity with programmable dram," in *International Symposium on Computer Architecture*, 2013.

[16] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *IEEE Symposium on Security and Privacy*, 1997.

[17] Xilinx, "Zynq-7000 all programmable soc technical reference manual," 2013.

[18] S. E. co. LTD, "Exynos 4." http://www.samsung.com/global/business/semiconductor/product/application/detail?productId=7745&iaId=844, 2012.

[19] "Building android 4.2.2 bsp on zc702." http://www.wiki.xilinx.com/Building+Android+4.2.2+BSP+on+ZC702.

[20] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, 1996.

[21] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert, "Persistent data-only malware: Function hooks without code," in *Network & Distributed System Security Symposium (NDSS)*, Feb. 2014.

[22] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms.," in *USENIX Security Symposium*, 2009.

[23] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, "Towards a practical solution to detect code reuse attacks on arm mobile devices," in *Workshop on Hardware and Architectural Support for Security and Privacy*, ACM, 2015.

[24] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Integration of rop/jop monitoring ips in an arm-based soc," in *Design, Automation & Test in Europe*, 2016.

[25] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," *ACM Symposium on Operating System Principles*, 2007.

[26] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Recent Advances in Intrusion Detection*, Springer, 2008.

[27] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor.," in *USENIX Security Symposium*, 2004.

[28] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2011.

[29] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *ACM conference on Computer and communications security*, ACM, 2007.

[30] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz, "Dynamic hooks: hiding control flow changes within non-control data," in *USENIX conference on Security Symposium*, USENIX Association, 2014.

[31] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, "Cpu transparent protection of os kernel and hypervisor integrity with programmable dram," in *International Symposium on Computer Architecture*, 2013.

[32] 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 *22nd USENIX Security Symposium*, 2013.

[33] A. V. Fidalgo *et al.*, "Real-time fault injection using enhanced on-chip debug infrastructures," *Microprocessors and Microsystems*, 2011.

[34] M. Portela-García *et al.*, "On the use of embedded debug features for permanent and transient fault resilience in microprocessors," *Microprocessors and Microsystems*, 2012.

[35] J. Lee, Y. Lee, H. Moon, I. Heo, and Y. Paek, "Extrax: Security extension to extract cache resident information for snoop-based external monitors," in *Design, Automation & Test in Europe*, 2015.