The DragonBeam Framework: Hardware-Protected Security Modules for In-Place Intrusion Detection

Man-Ki Yoon
University of Illinois at Urbana-Champaign
mkyoon@illinois.edu

Mihai Christodorescu
Qualcomm Research
mihai@qti.qualcomm.com

Lui Sha Sibin Mohan
University of Illinois at Urbana-Champaign
{lrs,sibin}@illinois.edu

Abstract

The sophistication of malicious adversaries is increasing every day and most defenses are often easily overcome by such attackers. Many existing defensive mechanisms often make differing assumptions about the underlying systems and use varied architectures to implement their solutions. This often leads to fragmentation among solutions and could even open up additional vulnerabilities in the system.

We present the *DragonBeam Framework* that enables system designers to *implement their own monitoring methods and analyses engines* to detect intrusions in modern operating systems. It is built upon a novel hardware/software mechanism. Depending on the type of monitoring that is implemented using this framework, the impact on the monitored system is very low. This is demonstrated by the use cases presented in this paper that also showcase how the DragonBeam framework can be used to detect different types of attack.

Categories and Subject Descriptors K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

1. Introduction

As attackers expand their reach into well protected systems, no layer is safe from intrusions. The targets of attacks in recent years have ranged from applications to middleware services, operating system (OS) kernels and device drivers, hypervisors and even firmware. The sophistication of such attacks, then, makes it harder to identify and build a trusted computing base (TCB) to develop security mechanisms. The common approach has been to move security monitoring (e.g., the reference monitor) functionality into a secure domain (say, a virtual machine separated from the virtual machine that must be secured) [14, 26]. This suffers from the existence of a *semantic gap* between the interface used for monitoring and the interface useful for security decisions. Another problem is that different security mechanisms use a variety of architectures to implement their solutions. Trying to combine one or more of these to improve the overall security of the system could result in a spaghetti of architectural mechanisms that, in itself, might open up new vulnerabilities. Hence, there is a need to provide system designers with a cohesive framework for implementing their monitoring and analysis techniques.

Our approach to solving such problems is to start with secure hardware and then to *bootstrap security into higher layers*. We propose a system where the *secure hardware* is the first level TCB and introduces security monitoring components into the system layer above (in this case, the OS kernel). By running the monitoring component in the layer that is the target for attackers, we gain significant visibility into local operations as well as effects of attacks – thus avoiding the semantic gap. The secure hardware ensures the runtime integrity of the upper-layer security monitoring component(s) by *protecting it* and *continuously validating its liveness and behavior*. Designers can then implement methods/hooks (to monitor the system resources/components that they care about) in our security monitoring component. The gathered information can be offloaded to a redundant hardware component where the designers can apply their own analysis techniques on the collected data. We call this the *DragonBeam framework*.

The DragonBeam framework is a set of software and hardware mechanisms that allow us to develop and maintain a *two-level monitoring system* that consists of: (i) a *kernel module* that resides in the OS kernel – it carries out any desired security checks and measurements and (ii) capabilities to monitor the *behavior* of this module and to establish its integrity by a combination of (a) integrity measurement and (b) runtime challenge-response protocol – these latter components actually execute on a trusted computing base that resides on a different core of the processor. One advantage of using such methods is that the DragonBeam framework *does not* require modifications to the OS or the applications. In addition, since the DragonBeam architecture provides a separate core for executing the TCB-related components, (i) the overhead for the continuous security interactions is low and (ii) the effects on the critical executions paths is limited (and often negligible).

SYSTOR '16, June 6–8, 2016, Haifa, Israel.

http://dx.doi.org/10.1145/2928275.2928290

The methods presented in this paper lay the groundwork for DragonBeam to become a *generic framework for developing/implementing security solutions*. System designers can implement their favorite monitoring and/or data capturing methods in the kernel module that can then either analyze the data itself or relay it to additional analysis components on the TCB. These components can perform more extensive analyses as required.

Our paper makes the following contributions:

1. We introduce a novel framework, DragonBeam, for implementing monitoring and intrusion detection solutions – it provides hardware-guaranteed integrity for the security monitoring system. The latter can be extended to other system layers to create a multi-level monitoring system rooted in the secure hardware. An overview of the framework is presented in Section 2 while details are in Section 3.
2. We have implemented the framework and carried out evaluations on an FPGA softcore processor (Section 5). This helps us in gauging the real hardware costs for implementing such a system. As we see later in the paper, the additional hardware costs are less than 1%.
3. Two use cases as well as a performance evaluation are used to illustrate how to use the framework in Section 6. These use cases not only demonstrate the ease of use but also highlight the flexibility of our approach by showcasing different types of attack detection methods. The evaluation shows negligibly small performance overheads.

1.1 Threat Model and Assumptions

We aim to make our threat model as broad as possible which is in line with recent developments [10, 22]. An attacker can breach any part of the software stack (OS kernels, middleware, runtime libraries, applications to name a few) and can even have full control of any software running on the main (monitored) system. We assume that the attacker does not perform physical attacks against the hardware. Thus trust can be placed in the hardware components, and in particular in our TCB. We further assume that the whole system (both software and hardware) is secure during boot time as well as immediately after the boot sequence is complete; also updates to the TCB require physical access.

While our threat model is quite broad, an attacker may attempt to carry out malicious activities between operations that verify the integrity of the DragonBeam framework – the attacker could corrupt some components of our framework and restore it just before the next check. Such transient attacks [18, 38] cannot completely be ruled out in external monitoring mechanisms [29] and the DragonBeam framework is no exception. On the other hand, some of the mechanisms presented in this paper, viz., the randomization techniques (Sections 3.4, 6.1 and 6.2), will help mitigate such attacks.

2. Overview

The main idea that we propose in this paper is that of a *hardware/software mechanism* to detect intrusions. This is achieved by using a two-level monitoring framework that we call DragonBeam. This architecture takes advantage of the redundancy available in computing resources on a modern multicore architecture – we trade off performance to improve overall security by using one of the cores to *monitor* the other core(s). This mechanism can monitor the operating system, the applications executing on the monitored core, or both. Figure 1 presents the DragonBeam architecture, which we illustrate with use cases in the remainder of this section.

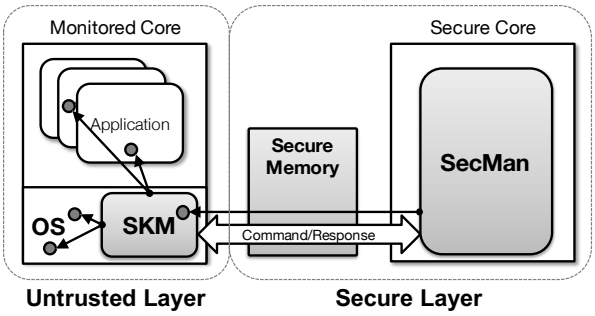

Figure 1. Overview of the DragonBeam architecture.

2.1 High-level Architecture

Figure 1 shows the DragonBeam framework, in which a trusted on-chip entity, the *secure core*, continuously monitors the runtime behavior of another (potentially untrustworthy) entity, the *monitored core*. The secure core is part of our trusted computing base. Since both cores are on the same die, minor hardware modifications are required to extract the relevant information directly from the monitored core. This increases the trustworthiness of the monitored signal since they are transparent to any code that executes on the monitored core. On the other hand, the amount of information that can be gathered is somewhat limited by the amount of hardware changes (essentially probes) that can be made. Such changes are often intrusive and might require significant efforts in design and verification.

To solve these problems and to increase the amount of information that the secure core can gather, we introduce a *software module* that executes inside the monitored core as a kernel module. We call it the *Secure Kernel Module* (SKM). The SKM is responsible for capturing information about the applications as well as the OS that executes on the monitored core and passes this information (with help from a hardware unit) to the secure core. The SKM is controlled by a software module that executes on the secure core called the *SKM Manager* (SecMan). The SecMan also ensures that the SKM is not attacked or prevented from executing or carrying out its intended functions. The SKM acts as a bridge to gather information about the behavior of the OS and/or applications on the monitored core; this information is then sent to the secure core for analysis.

Hence, the overall architecture of our solutions is composed of *three* major components, viz.,

1. the Secure Kernel Module (SKM), which runs in the untrusted operating system,

2. the SKM Manager (SecMan), which protects the integrity of the SKM, and
3. the secure core, which runs the SecMan.

We now elaborate on the SKM and SecMan components in the following sections.

2.1.1 Secure Kernel Module (SKM)

The SKM is a *kernel module* that resides on the monitored core. It is controlled by the SecMan and carries out a variety of security functions, chief among which is to *gather information about the execution behavior* of important OS components and applications. It could either actively analyze or passively send to the secure core the information that is gathered. The specific tasks depend on (i) the particular issue that is being tracked and (ii) what the designer of the system has decided. For instance, it could (a) perform integrity checks on the kernel code and/or data structures, (b) monitor the behavior of some critical applications, (c) actively monitor what processes/modules are executing, and (d) what low level resources are being requested by what processes and how they are being used among other things. The SKM executes in the most privileged mode as do other kernel components. Hence it has access to all of the kernel data structures. This helps it detect anomalous activities on the monitored core.

The DragonBeam framework does not prescribe in any way the actual functionality of the SKM. The goal of the framework is to ensure that the SKM operates without any external effects on its code or data, even when the attacker has root-level access to the system. The execution of the SKM is closely tracked by the SecMan – this prevents the situation that the SKM itself is taken over or prevented from executing. The SKM is developed by the system designers themselves and also has a fixed, limited, functionality. Hence, it is easier to verify that the SKM itself does not expose any security vulnerabilities.

2.1.2 Secure Kernel Module Manager (SecMan)

The SecMan actively manages and monitors the execution of the SKM. The SecMan and SKM follow a *command and response protocol* that has been developed by the system designer. Consider the following example to illustrate this: (a) the SecMan tells the SKM to capture the current state of the process list in the kernel; (b) the SKM wakes up and gathers this information; (c) the SKM communicates this information back to the SecMan and finally, (d) analysis modules on the secure core will compare the state of the process table to ones that were captured previously to check for unexpected processes – this might allow us to detect the execution of malicious process on the monitored core. The overall protocol will include a fixed set of commands that bound the operations of the SKM. At runtime, the SecMan will issue one of these commands (or a sequence of them) and the SKM will execute them in the order received.

Another important function of the SecMan is to *guarantee the integrity and the liveness of the SKM itself* via code hash and a heartbeat mechanism. This is why we call this a two-level monitoring architecture – the SKM monitors the behavior of kernel components and applications while the SecMan monitors the execution of the SKM itself.

As shown in Figure 1, the secure core is on-chip hardware supplemented with a *secure memory* module that facilities secure communication between the monitored and the secure core. The secure memory relays commands from the SecMan to the SKM and also the data from the SKM (i.e., results from executing the commands) to the SecMan. The main aim is to carry out these operations in a trusted manner. This secure memory can only be accessed by either the SKM or the secure core. This ensures that the commands and responses cannot be intercepted, corrupted, or faked by an adversary.

2.2 Sample Use Cases

We now present simple use cases for this architecture to illustrate the power of this two-level monitoring mechanism.

The integrity of the system call table in the kernel is very important. Some kernel rootkits often overwrite the entries in the system call table to hijack the execution of benign processes and hide the presence of malicious processes or files [6]. One way to detect such rootkits is to regularly check the state of the system call table – the state can be checked against what was stored at the secure-startup. Hence, the SecMan can send commands to the SKM to capture the current state of the system call table and copy it into the secure memory. If the recently captured state of the table deviates from what is expected, then the secure core can take corrective action or raise alarms. Similar operations can be carried out to verify the integrity of the interrupt vector and to find malicious kernel module or user processes.

2.3 Requirements and Challenges

For the rest of this paper we will address the following requirements and challenges in implementing the two-level SKM-based monitoring architecture:

1. The SKM must not be compromised even if the kernel on the monitored core is successfully attacked; also, the SecMan must be able to detect if the SKM is no longer operating as expected.
2. The SKM should act promptly in response to commands from the SecMan.
3. The secure memory must not be corrupted by an adversary, even when the adversary has gained root access on the monitored core. The access controller for the secure memory must ensure that only the SKM and the secure core have access to the secure memory.

3. Detailed Architecture

We now present more details about DragonBeam framework.

3.1 DragonBeam Framework Operations

We will explain the details behind each step in the DragonBeam framework using the example of detecting problems in the system call table (explained in Section 2.2). An overview is presented in Figure 2. The steps are:

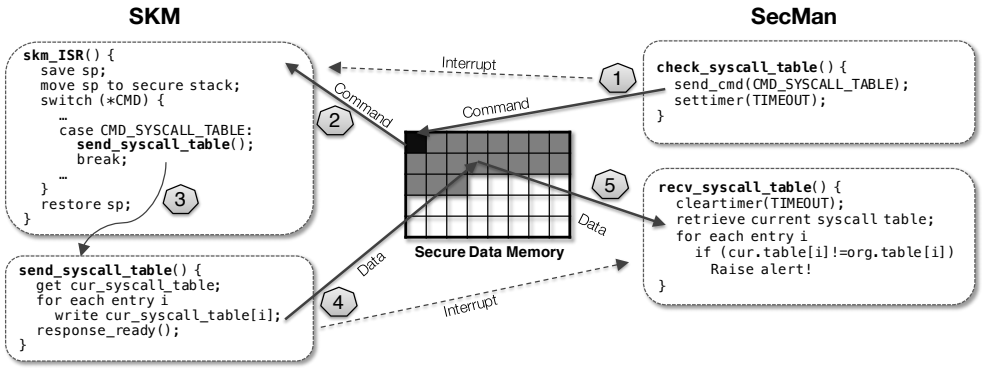

Figure 2. Overview of the execution of an example security task (integrity check of system call table) in the DragonBeam framework. The SecMan issues a request (1) to the SKM via an interrupt. The SKM collects the data needed (i.e., the contents of the system call table) and passes the data back to the SecMan via the secure memory in steps (3) and (4). Finally the SecMan verifies the integrity of the received data (5).

1. The SecMan sends a command (via the secure memory) to check the system call table.
2. The above process results in an inter-core interrupt that is handled by an interrupt handler on the monitored core.[1] The interrupt-service routine, `skm_ISR()`, is the main body of the SKM and handles the command sent by the SecMan. No entity on the main core (other than the SKM) can know the command because *access to the secure memory is restricted*.
3. The SKM calls the appropriate function to carry out the required task.
4. The function carries out its operations by placing the required information in the secure memory.
5. The secure core receives an interrupt that the data it requested is now available in the secure memory. On receiving the interrupt, the SecMan reads out the information from the secure memory region and then sends it to the appropriate module in the secure core that can analyze this information.

The above process repeats for every command that is sent by the SecMan to the SKM. Let us now look at the details that enable the above process.

3.2 SKM Registration

The SKM is loaded onto the monitored core during the booting phase of the OS. Our assumption here is that the system is in a clean state at boot time (we could even use a secure-boot mechanism such as IMA [31]).

The first task for the SKM (at load time) is to request the SecMan to register it. Figure 3 shows the SKM registration process. For the SecMan to test the veracity of the SKM's request for registration, it uses a hash of the latter's `.text` section. Since we do not trust the SKM yet, we do not wait for it to send its hash; rather, we calculate it *directly from the main memory.* This prevents malicious modules from copying the SKM's hash in order to pass off as legitimate modules.

[1] We will use 'monitored core', 'application core' and 'main core' interchangeably.

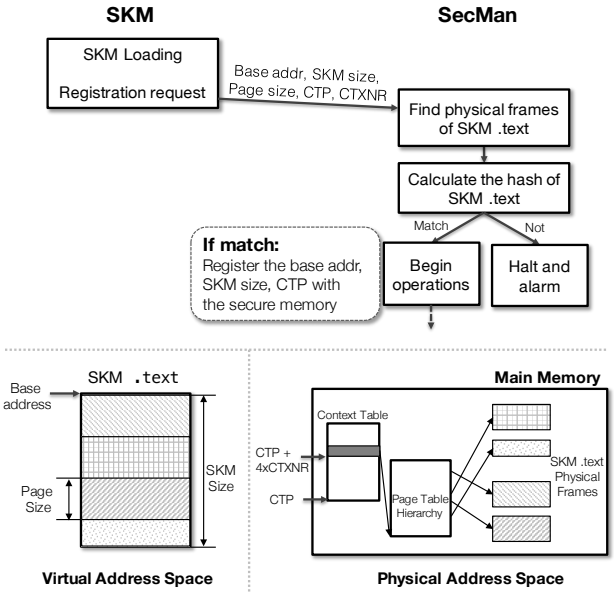

Figure 3. SKM registration during secure boot.

The SecMan needs the following information about the SKM to compute this hash value: (a) the virtual base address, (b) the size of the `.text` section, (c) the page size and the (d) page table information—all for the SKM. In SPARC processors (that our prototype is based on; see Section 5), the last piece of information corresponds to the *context table pointer* (`CTP`) and the *context number* (`CTXNR`) [34]. The `CTP` points to the root of the page table tree and the `CTXNR` is used to index the context table (i.e., the page table) of the current process (hence `CTXNR` is unique for each process). The SecMan then translates the `.text` section of the SKM as defined by the base (virtual) address and its size to a set of physical addresses that host the SKM code. The SecMan then calculates a hash of the physical frames that store the SKM code using, for instance, the SHA-1 algorithm [13] (or whatever hashing algorithm that the systems designers favor). If the newly computed hash matches what was calculated at the design time, then the SecMan registers the base address for the SKM's `.text` sec-

tion, its size, the page size, CTP and CTXNR with itself. Also, the base address, size and CTP are registered with the secure memory controller. From then on, the secure memory can *only* be accessed by the code that is verified to be part of the SKM.

Note that a malicious module may make a registration request to the SecMan. However, with the registration process described above, the only way for the attacker to pass the registration process is to implement the malicious code in such a way that its .text section has the same hash value as the SKM's one – remember that the hash is *directly* calculated by the SecMan from the given information about the malicious module's .text section.

3.3 Secure Memory and Access Control

The secure memory enables secure and trusted communication between the SKM and the SecMan. The secure memory controller only allows the SKM on the main core (and any from the secure core) to access this on-chip memory. To implement this access control, we use the program counter value to identify *who initiated* a memory transaction to the secure memory. The program counter-based memory access control has been used in the context of embedded devices with no support of virtual memory [21, 24, 36]. In systems with virtual memory, however, the program counter cannot be used as a unique identifier. Hence we combine it with a coarse-grained check on the address mapping information.

Figure 4 presents the overview of the secure memory access control process. The secure memory controller accesses the program counter (a virtual address) register (PC) on the main core as well as the context table pointer register (CTP). It then checks if (i) the CTP register matches the one registered by the SecMan during the SKM registration phase and (ii) the value of the PC register is within the .text region of the SKM. Note that we do not use the context number register (CTXNR) because the kernel memory address mappings are identical across all contexts on SPARC [34] and thus the context number is ignored by the MMU during kernel address translations. If the above two conditions are satisfied, we can ensure that only the SKM can access the secure memory.

However, it is entirely possible that a smart adversary may have modified the page table referenced by the legitimate CTP register so that the virtual addresses indexed by [Base, Base+Size] are mapped to the malicious code instead of the SKM's physical addresses. Also, the adversary may have set up a whole new context table tree at a different location. The malicious module might then be able to access the secure memory by modifying the address mappings. To prevent such problems, we could check the physical addresses of the instructions trying to access the secure memory to verify if it falls within the SKM's physical frames. However, we avoid this option since it would involve an address translation each time the secure memory is accessed—this would result in huge performance overheads. Our solution (as will be elaborated in Section 3.4) is for the SecMan to (a) translate the SKM's .text section (indexed by [Base, Base+Size]) from virtual to physical addresses using the registered CTP and CTXNR register values and (b) calculate the hash of the resultant frames—following exactly the same process as the registration, only done *often during execution*, not just at the registration time. If the attacker has modified the page table so that the virtual address range [Base, Base+Size] points to its malicious code, the resultant hash value will not match the legitimate value. Otherwise, we can be sure that the range [Base, Base+Size] is pointing to the SKM. Also, an altered CTP value cannot be effective because this simply violates the secure memory access control. Of course, these do not eliminate the possibility where a carefully constructed malicious module changes the address mappings and restores it to the original between two hash check points. However, regular hashing with random time intervals can significantly reduce the possibility of success for such attempts.[2]

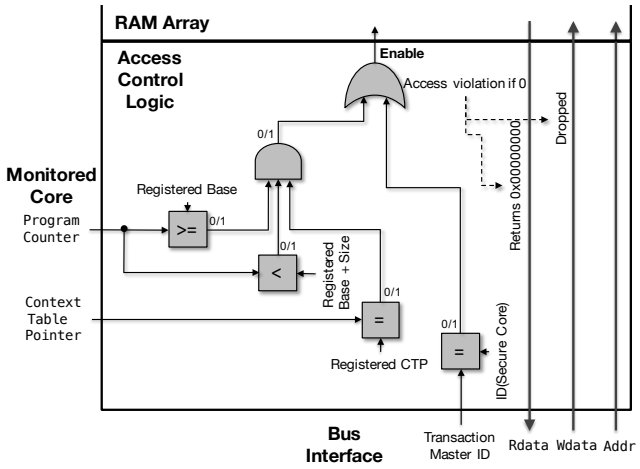

Figure 4. Secure memory access control.

If the memory controller detects an illegal access attempt, the controller returns 0 for read or drops write transactions.

3.4 Heartbeat and SKM Integrity Check

As mentioned earlier, we need to continuously check if (i) the SKM's code has not been tampered with; (ii) the SKM is actually starting up when it is commanded to do so by the SecMan and finally (iii) the SKM responds to requests in a timely manner. We have already discussed a technique to check for (i). The issues of liveness of the SKM, viz., (ii) and (iii), require a more active approach.

Our solution for this problem is to occasionally send out a special *heartbeat* command to the SKM. The SKM should respond to this command right away. This ensures that the SKM has not been deactivated by an adversary. No other process can respond to this command since the communication happens through the secure memory that can be accessed only by the SKM. When a heartbeat command is sent to the SKM, the SecMan starts a countdown timer. If it does not receive the expected response from the SKM then this is also indicative

[2] The attacker would want to do this type of transient attack in an attempt to impersonate the SKM and send fake response through the secure memory. The attacker would therefore try to alter the address mappings before the SKM is activated by a command from the SecMan. Such a threat can be significantly reduced if the SecMan performs the hashing right before sending the command to the SKM.

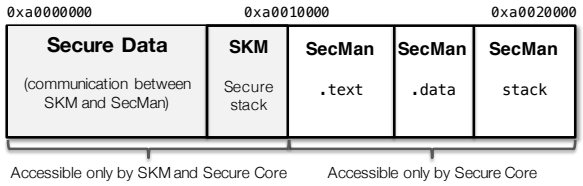

Figure 5. The secure memory map.

of an attack. The timer can be used for any/all commands that the SecMan sends across.

Another precaution that we can take is to send out these heartbeat commands, in addition to the SKM operations and hashing, at *random* intervals so that an adversary cannot guess the pattern of checks by the SecMan.

3.5 Secure Memory for SecMan and Secure Stack

To ensure the integrity of the SecMan we load its code and data onto a part of the secure memory region that can be accessible only by the secure core, as shown in Figure 5. Hence, an attacker that takes control of the main core cannot corrupt the SecMan. The SKM is loaded onto the main memory (along with the rest of the kernel),[3] and we monitor and protect it by the mechanisms described earlier.

The secure memory also hosts the *secure stack* for the SKM. When an interrupt is raised and handled by the SKM, the first step taken by the ISR is to change the stack pointer to the secure stack (see Step 2 in Figure 2). Hence, a malicious module cannot read or alter the data stored in the secure stack even if it knew where the stack is.

4. Security Guarantees of the DragonBeam Framework

The DragonBeam framework provides for the secure operation of a kernel module, in the presence of attackers with full access to the system, including kernel-level access. We outline here the potential attack vectors and the defenses that the DragonBeam framework provides.

SKM Code Integrity: An attacker may try to replace the SKM with a malicious module and thus have it loaded onto the system, either during the boot-phase or later during system operation. This is prevented by the SKM registration process which requires the exact hash value obtained at the design time. Also, the hash is directly computed by the SecMan using the page table information of the caller to dissuade such attempts. The attacker may also try to modify the SKM's ISR. However, since it is placed in the SKM's `.text` section, any attempts by the attacker to change the ISR will be detected by the hashing mechanism mentioned earlier.

SKM Control Flow: The attacker may try to change the state or parameters used by the SKM in order to cause a buffer overflow and change its execution flow. This can take many forms, from code-injection attacks, to return-to-library attacks and to return-oriented-programming attacks. All of these attacks rely on a software module processing its inputs in a vulnerable way; for the purposes of this paper we assume the SKM is well designed and implemented to avoid such problems. This is a realistic assumption as the SKM is supposed to be functionally self-contained and small.

SKM Availability: The attacker may try to disable the SKM by preventing it from being scheduled for execution on the CPU or disabling interrupts. Hence, to guarantee the liveness of the SKM, we use a heartbeat mechanism (Section 3.4). The attacker may try to impersonate the SKM by redirecting the interrupts to itself and by responding to the heartbeats. We prevent this situation by using the secure memory as a communication channel; no entity, other than the SKM, can write a response to the secure memory and thus the fake response cannot be delivered to the SecMan.

SecMan Integrity: The attacker may try to corrupt the SecMan directly so that none of its security functions are invoked in the first place. This cannot happen in our architecture since the attacker cannot access the memory region of the SecMan.

OS Integrity: An attacker may try to create a distinction between the code and data of the actual running kernel and the code and data inspected by the SKM. Indeed such attacks are possible – more so with the runtime splicing support present in kernels these days. Such a distinction between the running and observed kernel states might allow an attacker to perform malicious tasks – the SKM may not detect these anomalies if it is not well designed. For example, an attacker who wishes to hijack the system call table could modify the software interrupt table to point to a new system call dispatch handler, which in turn uses a new system call table placed elsewhere in memory. Thus, the SKM's task is not just to read and check the system call table but also to verify that the code that is supposed to use this table is valid and active. At a minimum the SKM must check the interrupt descriptor table, the code of the appropriate interrupt handler and, finally, the system call table used by that code. We leave the design of such checks to the designer of the SKM while we ensure the integrity and confidentiality of the checks via the DragonBeam framework.

5. Implementation

We implemented the DragonBeam framework on a Leon3 processor [2] for a Xilinx ZC702 FPGA [4]. Leon3 is a soft-core processor based on 32-bit, in-order, 7-stage pipeline SPARC V8 architecture [34]. From the soft-core implementation, we demonstrate the ease with which we can make the required modifications and (b) measure the hardware costs.

5.1 System Configuration

Figure 6 shows our DragonBeam framework implementation on Leon3 processor and Table 1 lists the details about the implementation. The system consists of two cores each of which runs at 83.3 MHz and the system has a main memory of 256 MB. Each core has L1 instruction (16KB) and data (16KB) caches. The MMU has split TLBs for data and instruction. The Leon3 processor also includes a single-port on-chip RAM,

[3] The SKM could be loaded onto the secure memory. However, this requires a modification of the OS, which we avoid.

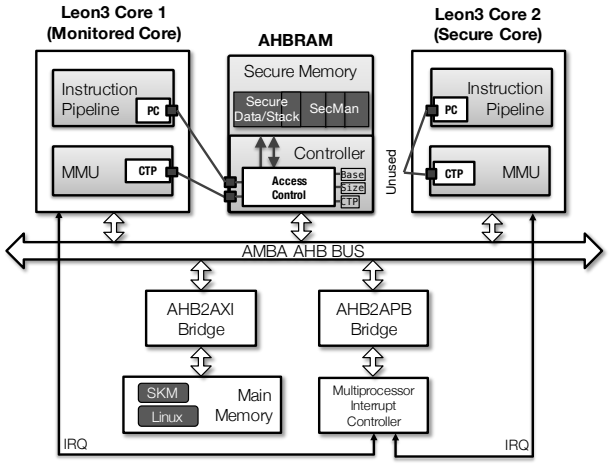

Figure 6. The DragonBeam framework implementation on Leon3.

AHBRAM, to which the cores can access through AMBA [1] AHB (Advanced High-Performance Bus) bus, as depicted in Figure 6. We instantiated it as an 128KB on-chip SRAM that is addressable at 0xa0000000. The first half is used as the secure communication channel between SKM and SecMan and also for the secure stack of the SKM, as shown in Figure 5. The bottom half is used by the SecMan. The entire region is set to be *uncacheable*, otherwise a non-SKM process can access the cached data without accessing the secure memory.

5.2 Secure Memory Implementation

We modified the control logic of AHBRAM, i.e., the on-chip SRAM, to implement the secure memory. As explained in Sections 3.2 and 3.3, the information about SKM's `.text` and page table are stored in the secure memory for the access control. For this, we designate the first 16 bytes of the secure memory (i.e., 0xa0000000–0xa0000010) as special memory-mapped registers in which the SecMan can write the information during the SKM registration phase. The controller checks the AHB master ID of the memory transaction and grants access to these registers only from the secure core. Hence, the information can be set only by the SecMan. The bottom half of the secure memory used by the SecMan (explained in Section 3.5) is protected in the same way.

After the SecMan has validated the SKM at registration time, the SecMan *locks* the memory that contains the control data (i.e., the first 16 bytes) by asserting the lock bit in the control register. From then on, access control to the secure data and secure stack regions becomes enabled. As explained in Section 3.3, the access control logic requires the current program counter (PC) and context table pointer (CTP) values from the monitored core. As shown in Figure 6, we extract the PC value from the fetch stage of the core's instruction pipeline and the CTP value from the MMU and feed them to the secure memory controller. It returns 0 or drops the transaction for an illegal read and write request, respectively.

5.3 Software Configuration

The monitored core runs an *unmodified* Linux 3.8 kernel residing on the main memory as shown in Figure 6. The SKM is implemented as a Linux Kernel Module and resides in the main memory. The SKM has about 350 lines of C code (including spaces) that implement the two use cases introduced in our evaluation (Section 6) along with interrupt handling and inter-core communication routines.

Table 1. Details about the Implementation Platform.

Implementation Artifact	Value
Platform	Leon3 on Xilinx ZC702 FPGA
Processor	SPARC V8 Dual Core @ 83.3 MHz each
Main Memory	256 MB
L1 cache	Split, 16 KB, LRU
TLB	Split, 8 Entries, 4KB page, LRU
Secure Memory	128 KB, Single-port
Monitored Core OS	Linux 3.8
Secure Core OS	None (Bare Metal Execution)

We implemented the SecMan as a *bare-metal executive* running on the secure core for the purposes of our proof-of-concept. A complete system can also have an OS and analysis modules running on the secure core. The SecMan has about 450 lines of C code (excluding the SHA-1 library) a majority of which is for interrupt and timer-related functionality. As mentioned above, the SecMan resides in the bottom half of the secure memory, accessible only by the secure core.

6. Evaluation

In this section we evaluate the DragonBeam framework along the following lines: (a) how it can be used by system designers to implement different detection mechanisms to catch malicious activities; (b) the overheads imposed by the DragonBeam framework on the main system; and (c) the costs for implementation in hardware.

6.1 Implementation of Detection Mechanisms

To demonstrate the effectiveness and versatility of our framework we implemented two existing detection mechanisms. Note that we are *not* proposing new detection techniques for any of the use cases presented here. We instead intend to demonstrate *how to use* our framework for the benefit of system designers. We also intend to show how SKM closes the semantic gap by running directly inside the untrusted OS and collecting information useful to security decisions.

6.1.1 Hidden Module Detection

Many kernel rootkits such as modhide [5], suterusu [3], etc. hide themselves from the kernel module list to avoid detection by anti-virus software. The hidden modules are invisible from even lsmod or \proc\modules, both of which read the list of currently loaded modules from the same kernel data structure. For the following experiments, we used the `suterusu` kernel rootkit that hides by deleting itself from the kernel module list.

Hidden kernel modules can be detected by scanning the memory region where modules are typically placed. The main idea is that every page that is present in this memory region should be allocated to one of the modules present in the kernel module list as such pages are not swapped out. If even one page cannot be matched up to a known module, then it is an indication of a hidden module. For instance, in our

experimental setup, the memory space where kernel modules reside lies in the range of 12 MB which hosts 3072 pages of size 4 KB each.

We implemented the detection method using our DragonBeam framework as follows:

1. For each page in the module memory space, we check if it has been loaded into memory by checking the requisite flag, viz., the present bit. We collect information on all pages in this region.
2. We traverse the module list. For each module, we obtain its base address and size – this corresponds to the list of pages used by the module. We mark off the pages associated with each module from the list of pages obtained above.
3. If any of the pages from step 1 have not been marked off at the end of step 2, then it is an indication of the existence of hidden modules in the system.

The SecMan sends commands to the SKM to execute the procedure described above. The SKM replies with the results of this checking procedure. To prevent attackers from evading the checking procedure mentioned above, the SecMan must send the commands at *random* points in time.

We used the `suterusu` kernel rootkit that tries to hide its presence in the system. Using our checking mechanism, the SKM found *two pages* that had been allocated for this rogue module's code and data. Other rootkits that operate in a similar manner will also be caught by this procedure.

6.1.2 System Call Table Integrity Check

Many rootkits *hijack* system calls [6] to intercept sensitive data, hide malicious processes or files, *etc.* One way to detect such attacks is to verify that the current state of the system call table matches the original state immediately after a secure boot. This will detect rootkits that rewrite entries in the table.

First, the SecMan asks the SKM to capture the *initial* state of the system call entry table. This information is passed via the secure memory to the SecMan. This happens right after the SKM has registered with the SecMan, at which point the OS state is still trustworthy. The SecMan stores this initial state in its internal memory region, part of the secure memory. During regular execution, the SecMan asks the SKM to send snapshots of the system call table. The SecMan compares the newly received state information with the one obtained initially. If it detects a change in the table, then that is an indication of a rootkit having hijacked certain system calls.

We used the `modhide1` [5] rootkit for our experiment – it hijacks the open system call to prevent the detection of a module being inserted into the kernel (e.g., cat \hide hides the module). Using the method described above, the SecMan was able to detect this rootkit's presence. More generally this approach can check the state of any critical kernel data such as the interrupt descriptor table and handlers (as described in Section 4), page tables and translation base register, etc.

6.2 Performance Evaluation

We now analyze the overhead imposed on the main core due to the execution of the DragonBeam mechanisms.

Table 2. Average latencies of SKM operations.

SKM Operation	Avg Latency (stdev)
Heartbeat	0.010 ms (4.100 us)
SKM `.text` hashing	2.679 ms (7.474 us)
System call table check	0.108 ms (5.727 us)
Hidden module detection	3.914 ms (4.772 us)

Table 2 shows the latencies of the heartbeat and hashing operations as well as the operations necessary for use cases explained in Section 6.1. The latency is measured between the time points when the SecMan sends a command to the SKM and to when the SecMan completes the analysis after receiving data from the SKM. Each entry in the table presents the average of 1000 measurements and the standard deviation. Note that the hashing operation does not involve the SKM and thus the latency is simply the time required to (i) copy SKM `.text` to the secure memory and (ii) perform the SHA-1 operation. The time spent for the system call table check is to copy a table of 343 entries (total size 1.3 KB in Linux 3.8 on SPARC) and then to compare it (with the stored version of the table) entry by entry. The hidden module detection operations take much longer because it checks (i.e., looking up each entry in the three-level page table) all the pages in the kernel module space (3072 pages).

Beyond microbenchmarks, we used the SPECINT2006 [15] benchmark suite[4] and measured execution times for two situations: when the SKM is enabled versus when it is disabled. Specifically, we executed each benchmark 20 times for each of the following scenarios: (a) when the SKM is not running, (b) heartbeat (c) SKM `.text` hashing (d) system call table integrity check and *(e)* hidden module detection is enabled. To obtain stable results, we used a fixed period (100 ms at the CPU clock speed of 83.3 MHz) for the operations. Hence, the SKM was sent commands to execute each operation about 1600–6400 times during one benchmark execution.

Figure 7 shows the average (geometric mean) ratios of the execution times for each benchmark when the SKM is enabled (with an SKM operation) compared to when the SKM is disabled. As the plot shows, the overheads associated with the SKM operations are very small. Among them, the hidden module detection incurs the biggest overhead (around 4%). This is because it is an *in-SKM procedure*, i.e., the analysis carried out inside the SKM module. We could reduce this overhead by offloading the analysis to the SecMan – i.e., the SKM can just send the list of modules with their base addresses and sizes to the SecMan. Since the latter can physically address the main memory it can obtain the page information directly and check the flags. Overall, the overheads are consistent with the latency of each SKM operation (except hashing) shown in Table 2. As mentioned above, the hashing operation does not directly involve the SKM, however it gen-

[4] We used five benchmarks – `bzip2, hmmer, libquantum, mcf, sjeng`– from the suite after excluding ones that are failed to cross-compile to the Leon 3 SPARC platform or ones that took very long time to execute one single execution trace (since the FPGA softcore is slower on average than regular processors). The average execution time for each of these benchmarks (without the DragonBeam framework) are 370, 198, 161, 511 and 639 seconds, respectively.

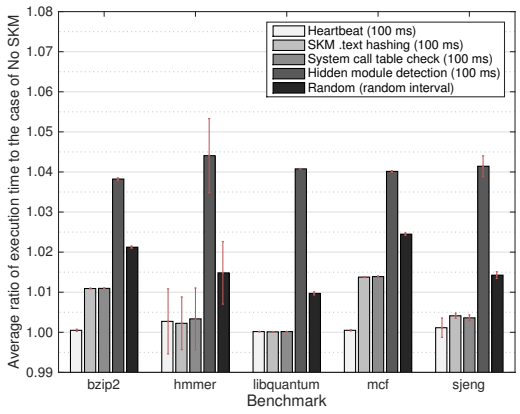

Figure 7. The average ratio of execution time to the case when SKM is disabled for different SKM operations.

erates a decent amount of bus traffic when copying the SKM's `.text` from the main memory to the secure memory. Hence, the operation indirectly imposes overheads on the main core by interfering with the memory traffic of the core. The system call table integrity check operation affects the main core's memory traffic in the same way because it copies the system call table between the main and secure memories. As we can see from Figure 7 the overheads imposed by these two operations vary more (as compared to the other two sets of bars) across the benchmarks. This is because of the different memory footprints of the benchmarks. `bzip2` and `mcf` have a substantially larger memory footprint than the other three [16] and hence they experience more overheads by such SKM operations that require large memory transfers.

Lastly, we performed a similar experiment with *random* operations that execute at *arbitrary intervals*. The SecMan sends a command randomly chosen from the four operations and then schedules the next event at some Δt after the current time. We configured Δt to be randomly drawn from $[0, 1, \ldots, 200]$ ms so that the median is 100 ms. However, we configured the SecMan to send the next command as soon as the SKM responds to the current one if $\Delta t < 10$ ms. That is, with 5% of probability, SKM operations can be *back-to-back*. These make it very difficult for an adversary to predict when and what kind of SKM operation(s) will occur next thus leading to a significantly reduced chance of success for transient attacks. The right-most bar of each benchmark group in Figure 7 shows the overheads imposed by this randomized check. The results are consistent with those of individual operation; the overhead is close to the average of the four operations.

These results show that our DragonBeam framework imposes very little overheads on the main system and it can still be an effective method to implement security mechanisms. One can also use the results when finding a proper combination of random periods (for different operations) according to the expected system load and allowable overheads.

6.3 Hardware Costs

Finally, we evaluate the hardware cost of the proposed architecture. The top half of Table 3 shows the cost of the hardware change in terms of the FPGA resource utilization (based on Xilinx ZC702 board [4]). The number in each cell is the number of resources used and the last column shows the extra resource due to the DragonBeam framework. The table shows that the DragonBeam framework adds a very small amount of hardware resources. This was possible because the only hardware changes in the framework are (i) the logic to fetch the 30-bit `CTP` (context table pointer) register from the MMU, (ii) the 32-bit `PC` (program counter) register from the instruction pipeline and (iii) the logic to implement the access control policy of the secure memory (see Figure 4). The logic for the two register fetch components is duplicated for each core although those of the secure core are not used in the access control. This result indicates that we can establish an on-chip secure communication channel between the SKM and the SecMan running on different cores (and also protect the latter's memory region) using less than 1% of additional hardware resources.

Table 3. FPGA Resource Utilization* of Leon3 Processor with and without the DragonBeam framework.

	Resource	Default	W/ DragonBeam	Δ
Dual Core	Registers	10258	10356	**0.96%**
	LUTs	19482	19511	**0.15%**
Quad Core	Registers	18932	19029	**0.51%**
	LUTs	37777	37835	**0.15%**

* Available resource: Registers (106400), LUTs (Look-Up Table, 53200)

6.4 Extension to Multiple Monitored Cores

So far we have considered the situation where the DragonBeam framework runs on a dual core processor. While we have shown how this works the issue remains that most modern systems have more than two cores. A typical configuration for such systems is that they run a single OS that manages all cores. In such situations, the SKM can run on any core at any given time. Hence, *every* monitored core needs to be hooked to the secure memory controller. Now, when a memory transaction comes in, the program counters and the context table pointer registers of all monitored cores are *multiplexed* with the transaction master ID. The rest of the logic remains unchanged as only one SKM exists and thus only one set of base, size, and `CTP` is registered.

We extended the original architecture (shown in Figures 6 and 4) to a quad-core configuration in which three cores are monitored by one secure core. The bottom half of Table 3 indicates that the extended architecture still imposes a very small hardware cost. This SMP configuration with a single SKM instance can reduce the performance overheads, especially due to in-SKM operations such as the hidden module detection, on the main cores because the SKM can run in parallel with the main applications running on different cores. To see this, we performed a similar experiment to the hidden module detection from Figure 7 in Section 6.2 with the quad-core configuration. The average overheads are between 0.8% and 2.1%, which are substantially lower than what we observed (about 4%) with a single monitored core on the dual-core setup.

For systems that have multiple operating systems executing on different cores (e.g., in the case of modern cloud computing systems with virtual machines that share a single under-

lying processor) the secure memory controller should be re-architected as there can exist multiple SKMs running on different monitored cores. The controller, at the very least, needs to have a separate register set for each individual SKM's information (i.e., the base, size, and CTP value) and these should be multiplexed with the transaction master ID. Also, the secure memory should be partitioned so that the SKMs do not interfere with each other. However we have not fully investigated if there needs to be additionally required HW changes.

6.5 Limitations

There exist some limitations of our approach. First of all, it is constrained to integrity checks. In other words, the DragonBeam framework does not cover information-leakage attacks (e.g., via side channels). In our model, access control mechanisms that are part of the DragonBeam framework are used to ensure the integrity and liveness of security components running in the untrusted OS but with no guarantees about the various side (or covert) channels created during the normal operation of the system.

Another important issue is that we trade off some performance for security – essentially one of the cores is reserved for security monitoring, which could otherwise be used as part of the main system. But this is something that system designers know about and can account for. This loss of performance comes with increased security guarantees – that might be fine for many systems where security is often very critical.

7. Related Work

The concept of hierarchically establishing integrity into a system appeared before in the Integrity Measurement Architecture (IMA) of the Linux kernel and part of the Trusted Computing approach [31]. IMA ensures that the system was in a secure state at some point in the past but we enable *dynamic monitoring at runtime*. Additionally, IMA does not perform integrity measurements for dynamically created data (e.g., transient runtime information) while our framework separates the integrity measurement *of* the security component from the integrity measurement done *by* the security component.

There exists work in which a multicore processor (or a co-processor) is employed as security measure in different aspects: (a) INDRA [33] where a monitoring core verifies functional behavior such as function return address using logs of application executions on monitored cores; (b) Chen et al. [8] delivers instruction-level traces (e.g., input/output operands, memory access address) to another core for inspection for the detection of memory leaks and access to unallocated memory; (c) Other work includes the use of hardware acceleration [9] and the attachment of reconfigurable logic to the main CPU checks [11]; (d) the Secure System Simplex Architecture (S3A) [23] and the SecureCore architecture [41] monitor the timing behavior of real-time embedded applications, the latter of which is extended to monitor memory behavior [42]. The common theme in these systems is that they dedicate hardware resources to specific security tasks. DragonBeam differs fundamentally in that we use the dedicated hardware of the secure core and the secure memory only to protect an in-memory security mechanism, i.e., the SKM. This allows us to support any desired type of security monitoring, irrespective of the events of interest, with no semantic gap in the way of accuracy.

Virtual Machine Introspection (VMI) [14] has been applied to process execution monitoring [20, 35], kernel control-flow integrity check [28], virtual memory and disk monitoring [25], dynamic information flow tracking [17, 40], system call tracing [30], etc. Although the 'out-of-box' approach can improve the security of intrusion detection system due to the separation, it misses out on a detailed view of the untrusted VM, that leads to *semantic gap* problems [12, 19]. Lares [26] closes the gap by placing hooks inside kernel APIs from which relevant information is passed to the security VM through the virtual machine monitor (VMM). They modified the VMM to protect the hooks by making their memory regions read-only. The switching between the VMs, however, cause a significant performance overhead. Sharif et al. [32] proposed Secure In-VM Monitoring (SIM) in which the hook handler runs in the same VM that it monitors (and thus reduces switching overhead) while being protected by the VMM through manipulation on the shadow page tables. This is also used by Wang et al. in [37] for in-VM hook memory protection.

One can implement our DragonBeam architecture with the use of virtualization in a similar manner. VMI has advantages in that no hardware modification is needed. However, VMI techniques rely on the security and correctness of the VMM, that in itself is susceptible to attacks [7, 27, 39]. DragonBeam significantly reduces the scope of such vulnerabilities. It also reduces overheads due to software interventions, at the cost of the additional hardware on chip, in this case, secure memory.

8. Conclusion

System designers must often contend with different modes and entry vectors of attacks, multiple security solutions and various monitoring and analysis techniques. A framework that can be used for integrating the different intrusion detection and analysis methods will provide significant value to such designers. In this paper, we presented such a framework that we call DragonBeam. The use of this framework allows designers to implement and carry out their own monitoring and analysis without affecting the execution of the main system, i.e., little to no effects on the critical paths of the system.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by grants from NSF CNS 13-02563, NSF CNS 12-19064, NSF CNS 10-35736, NSF CNS 14-23334, Navy N00014-16-1-2151, and Navy N00014-13-1-0707. Man-Ki Yoon was also supported by Qualcomm Innovation Fellowship. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

References

[1] AMBA Specifications. http://www.arm.com/products/system-ip/amba-specifications.php.

[2] LEON3 Processor. http://www.gaisler.com/index.php/products/processors/leon3.

[3] Suterusu Rootkit. http://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/.

[4] Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit. http://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html.

[5] modhide1 Rootkit. http://packetstormsecurity.com/files/favorite/24880/.

[6] Hijacking system calls with loadable kernel modules. http://r00tkit.me/?p=46.

[7] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith. Vm-based security overkill: A lament for applied systems security research. In *Proc. of the Workshop on New Security Paradigms*, 2010.

[8] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proc. of the workshop on Architectural and system support for improving software dependability*, 2006.

[9] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proc. of the International Symposium on Computer Architecture*, 2008.

[10] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[11] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture*, 2010.

[12] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proc. of the ACM Conference on Computer Communications Security*, 2013.

[13] D. Eastlake, 3rd and P. Jones. Us secure hash algorithm 1 (sha1), 2001.

[14] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, 2003.

[15] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[16] J. L. Henning. SPEC CPU2006 memory footprint. *SIGARCH Comput. Archit. News*, 35(1):84–89, Mar. 2007.

[17] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

[18] 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 *Proc. of the ACM Conference on Computer and Communications Security*, 2014.

[19] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proc. of the ACM Conference on Computer and Communications Security*, 2007.

[20] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proc. of the USENIX Annual Technical Conference*, 2006.

[21] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proc. of the European Conference on Computer Systems*, 2014.

[22] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[23] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *ACM Conference on High Confidence Networked Systems*, 2013.

[24] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proc. of the USENIX Conference on Security*, 2013.

[25] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proc. of Annual Computer Security Applications Conference*, 2007.

[26] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proc. of the IEEE Symposium on Security and Privacy*, 2008.

[27] G. Pék, A. Lanzi, A. Srivastava, D. Balzarotti, A. Francillon, and C. Neumann. On the feasibility of software attacks on commodity virtual machine monitors via direct device assignment. In *Proc. of the ACM Symposium on Information, Computer and Communications Security*, 2014.

[28] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proc. of the ACM Conference on Computer and Communications Security*, 2007.

[29] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Prof. of the Conference on USENIX Security Symposium*, 2004.

[30] J. Pfoh, C. Schneider, and C. Eckert. Nitro: hardware-based system call tracing for virtual machines. In *Proc. of the International conference on Advances in information and computer security*, 2011.

[31] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proc. of the USENIX Security Symposium*, 2004.

[32] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proc. of the ACM Conference on Computer and Communications Security*, 2009.

[33] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *Proc. of the International Symposium on Computer Architecture*, 2006.

[34] C. SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., 1992. ISBN 0-13-825001-4.

[35] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process out-grafting: An efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proc. of the ACM Conference on Computer and Communications Security*, 2011.

[36] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In S. Jajodia and J. Zhou, editors, *Security and Privacy in Communication Networks*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. 2010.

[37] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proc. of the ACM Conference on Computer and Communications Security*, 2009.

[38] J. Wei, B. Payne, J. Giffin, and C. Pu. Soft-timer driven transient kernel control flow attacks and defense. In *Proc. of the Annual Computer Security Applications Conference*, 2008.

[39] R. Wojtczuk. Subverting the xen hypervisor - xen 0wning trilogy part i. *Black Hat USA*, 2008.

[40] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proc. of the ACM Conference on Computer and Communications Security*, 2007.

[41] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Proc. of the IEEE Real-Time Embedded Technology and Applications Symposium*, 2013.

[42] M.-K. Yoon, S. Mohan, J. Choi, and L. Sha. Memory Heat Map: Anomaly detection in real-time embedded systems using memory behavior. In *Proc. of the ACM/EDAC/IEEE Design Automation Conference*, 2015.