Hardware-Software Collaboration for Secure Coexistence with Kernel Extensions

Daniela Oliveira
University of Florida
Dept. of Electrical and Computer Engineering
daniela@ece.ufl.edu

Nicholas Wetzel
Bowdoin College
Dept. of Computer Science
nwetzel@bowdoin.edu

Max Bucci
Bowdoin College
Dept. of Computer Science
mbucci@bowdoin.edu

Jesus Navarro
NVIDIA
jnavarro@nvidia.com

Dean Sullivan
University of Central Florida
Dept. of Electrical Engineering and Computer Science
dean.sullivan@ucf.edu

Yier Jin
University of Central Florida
Dept. of Electrical Engineering and Computer Science
yier.jin@eecs.ucf.edu

ABSTRACT

Our society is dependent upon computer systems that are the target of a never-ending siege against their resources. One powerful avenue for exploitation is the operating system kernel, which has complete control of a computer system's resources. The current methodology for kernel design, which involves loadable extensions from third parties, facilitates compromises. Most of these extensions are benign, but in general they pose a threat to system trustworthiness: they run as part of the kernel and some of them can be vulnerable or malicious. This situation is paradoxical from a security point of view: modern OSes depend, and must co-exist, with untrustworthy but *needed* extensions. Similarly, the immune system is continuously at war against various types of invaders and, through evolution, has developed highly successful defense mechanisms. Collaboration is one of these mechanisms, where many players throughout the body effectively communicate to share attack intelligence. Another mechanism is foreign body co-existence with its microbiota. Remarkably, these properties are not leveraged in kernel defense approaches. Security approaches at the OS and virtual machine layers do not cooperate with each other or with the hardware. This paper advocates a new paradigm for OS defense based on close collaboration between an OS and the hardware infrastructure, and describes a hardware-software architecture realizing this vision. It also discusses the architecture design at the OS and hardware levels, including experimental results from an emulator-based prototype, and aspects of an ongoing hardware implementation. The emulator-based proof-of-concept prototype, Ianus, uses Linux as the OS and the Bochs x86 emulator as the architecture layer. It successfully minimized kernel extensions interactions with the original kernel. Its security was evaluated with real rootkits and benign extensions. Ianus' performance was analyzed with system and CPU benchmarks and it caused a small overhead to the system (approximately 12%).[1]

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Operating system security and protection

General Terms

Security

Keywords

HW-SW collaboration, OS defense, kernel extensions, immune system

1. INTRODUCTION

Our society has become dependent on increasingly complex networked computer systems which are the target of a never-ending siege against their resources, infrastructure and operability for economic and political reasons. Attacks on computer systems can have devastating consequences to our society, such as a nuclear power facility going rogue, an electrical grid shutting down an entire city, or the financial sector going down after a hit in a bank [1, 2]. One particularly powerful avenue for system exploitation is the compromise of the operating system kernel, which has complete control of a computer system's resources and mediates access of user applications to the hardware. The current methodology for kernel design, which involves loadable extensions from third parties, facilitates system compromises. In this methodology the kernel has an original set of components that can be extended during boot time or while the system is running. This paradigm is adopted by modern OSes and represents a convenient approach for extending the kernel functionality.

Kernel extensions, especially device drivers, currently make up a large fraction of modern kernel code bases (approximately 70% in Linux and a larger percentage in Windows) [3]. Most of these extensions are benign and allow the system to communicate with an increasing number of diverse I/O devices without the need of OS reboot or recompilation. However, they pose a threat to system trustworthiness because they run with the highest level of privileges and can be vulnerable or malicious. This situation is paradoxical from a security point of view: modern OSes depend and must co-live with untrustworthy but *needed* extensions.

Similar to the current scenario of computer security, the mammalian immune system faces the exact same challenges every day, continuously battling against various types of invaders. It is hitherto the most successful defense system and has been perfected by Nature through millions of years of evolution. Leveraging immune system defense mechanisms built-up over eons is the key to im-

prove computer systems security, particularly OS security. Among its various defense mechanisms, two are most relevant. The first is *cooperation*. Team work is vital to a properly functioning immune system, where many types of white blood cells collaborate across the entirety of the human body to defend it against invaders. The second is *foreign body co-existence*. The human body has ten times more microbes than human cells. Most of these microbes are benign and carry out critical functions for our physiology, such as aiding digestion and preventing allergic reactions. In spite of that, a fraction of these microbes pose a threat to our bodies as they can cause pathologies. The immune system has evolved so that it can maintain an homeostatic relationship with its microbiota, and this involves controlling microbial interactions with host tissues, lessening the potential for pathological outcomes [4].

Remarkably, these two highly successful immunity mechanisms have not been applied to OS security. The two key players that make up a computer system, OS and hardware, interact precariously with each other and do not cooperate. Security approaches employed at the OS and virtual machine (VM) layers do not cooperate with the hardware, nor do they communicate information to dynamically adapt to future incursions. Current virtualization-based security solutions do not rely on collaboration with the guest OS because they are based on the traditional paradigm for OS protection, which advocates placing security mechanisms in a VM layer, thereby leaving the OS with no active role in its own defense. This is because the current threat model only defines the VM and the hardware as trustworthy so that the guest OS is considered untrustworthy and easily compromised by malware [5, 6, 7, 8, 9, 10]. This traditional model suffers from two main weaknesses.

The first weakness is the semantic gap: there is significant difference between the high level abstractions observed by the guest OS and the low level abstractions at the VM. The semantic gap hinders the development and widespread deployment of virtualization-based security solutions because these approaches need to inspect and manipulate abstractions at the OS and architecture level to function correctly. To address the semantic gap challenge, traditional VM-based security solutions use a technique called *introspection* to extract meaningful information from the system they monitor [5]. With introspection, the physical memory of the guest OS is mapped at the VM address space for inspection. High level information is obtained by using detailed knowledge of the OS layout, algorithms and data structures [11].

The second weakness of traditional VM-based OS defense mechanisms is the introspection mechanism itself, which is a manual, error prone, and time consuming task that, despite being perceived as secure until recently, *does rely on the integrity of the guest OS to function correctly*. Traditional introspection solutions assume that even if the guest OS is compromised, their mechanisms and tools, residing at a lower-level (VM) will continue to report accurate results. However, Baram *et al* [12] argued that this security assumption does not hold because an adversary, after gaining control of an OS (*e.g.*, through kernel-level rootkits), can tamper with kernel data structures so as a bogus view of the system is provided for introspection tools.

This paper advocates a new paradigm for OS defense: OSes should evolve to closely interact with the hardware playing an active role in maintaining safe interactions with their extensions. Similar to the immune system, computer systems should control the interactions between extensions and the original kernel lessening the potential for security breaches. In this paper, a hardware-software (HW-SW) collaborative architecture for OS defense is proposed. The main idea is that the OS will provide the hardware with intelligence needed for enforcement of security policies that allow for safe co-existence of the kernel and its extensions. Specifically, the following security policies are considered:

- Kernel extensions should never directly write into kernel code and data segments, including limited portions of the stack segment, except into their own address spaces.
- Kernel extensions should only interact with the kernel and other extensions through exported functions.[2]

Enforcing these policies requires architectural support so that extensions' execution are monitored and stopped in case of a violation. Controlling the execution of kernel extensions lies at the heart of the proposed approach. HW-SW collaboration is necessary because enforcing these policies requires system-level intelligence that only the OS can provide and architectural capabilities that only the hardware can support. For example, only the OS knows the boundaries of extensions in main memory and addresses of kernel functions. However, only the hardware can interpose on low level abstractions such as writes into main memory, invocation of `CALL` instructions and origin of the instructions for enforcing the policies. The cooperation benefits are clear when one considers that the OS and the hardware have access to a distinct set of functionalities and information, which in combination allows for enforcement of security policies that control the execution of kernel extensions.

This paper discusses the challenges of designing a hardware architecture that allows for cooperation and communication with the OS. An emulator-based proof-of-concept prototype called Ianus [3] was developed to validate the hardware implementation and the paradigm. It uses the Bochs Intel x86 emulator [13] as the architecture layer and Linux Ubuntu 10.04 (kernel version 2.6.32) as guest[4] OS. Ianus' experimental evaluation showed it successfully confined extensions into their own address spaces and contained their interactions with other parts of kernel code and data. Ianus' security was assessed with a set of real kernel rootkits which were all stopped before any malicious actions were performed and with benign extensions that could run normally. The overhead to the system was analyzed with a set of system and CPU benchmarks and was found to be low, approximately 12%.

This paper is organized as follows. Section 2 discusses the challenges of HW-SW collaboration for security and the main requirements for such architecture. Section 3 describes in details the design and implementation of an emulator-based proof-of-concept prototype validating this vision. In section 4 the paper shows the experimental analysis of the prototype in terms of security and performance. Section 5 brings a discussion of other aspects of the hardware implementation and future work. Section 6 summarizes relevant related work in kernel protection and hardware support for system security. Section 7 concludes the paper.

2. CHALLENGES FOR HARDWARE-SOFTWARE COOPERATION

The main challenge to HW-SW cooperation is that current security approaches are single-layered and separate the field into distinct realms, either hardware or software. These approaches are

[2]Exported functions are those required by extensions to perform their tasks and can be viewed as the kernel interface to extensions. Kernel extensions, when loaded into the system can also export a subset of their functions to other extensions.

[3]Ianus is an ancient Roman God who has two faces, each looking in the opposite direction. His image is associated with the ability of imagining two opposites or contradictory ideas existing simultaneously.

[4]The term "guest" is used here because the operating system is running on top of an emulator, which runs on a host OS.

isolated and work independently. Security solutions at the OS and VM layers do not cooperate with the hardware, nor do they communicate information about attacks. Hardware security approaches, on the other hand, mainly focus on Trojan prevention and detection and rarely leverage system level context. Further, OS defense approaches generally make the flawed assumption that the underlying hardware is trustworthy, while hardware malware has been found in many embedded systems from chips used by the military, to off-the-shelf consumer electronics [14, 15, 16].

The main step for allowing HW-SW cooperation is implementing the communication interface between OS and hardware following a pre-defined communication protocol. The goal is to provide a physical mechanism for hardware and software to communicate, apply security policies, exchange intelligence, and physically monitor system operations. This hardware component, named *hardware anchor*, is designed and implemented in the processor with the goals of minimizing the performance overhead and better utilizing existing hardware resources.

The hardware anchor is made of two main components: a cross-boundary interface and a software operation checker. The cross-boundary interface enables communication and information exchange between the OS kernel and the processor. The anchor receives and interprets security instructions from the OS kernel, collects system-level intelligence and enforces the security policies. The security instruction is a new instruction added to the processor and behaves like a software interrupt with parameters passed in general purpose registers. The intelligence information includes boundaries of extensions, kernel function addresses and types (exported or non-exported), user-defined sensitive information, and protected memory spaces. Since the security instruction fetching and decoding functionalities share the on-board processor resources with normal instructions, the cross-boundary interface will be seamlessly embedded with the processor. The information collected through the interface will be stored inside secure hash tables within the processor, and cannot be accessed by any other hardware module.

The second anchor component, the system operation checker, is also located in the processor and monitors OS operations based on information collected through the cross-boundary interface. For example, the OS will downcall the anchor to provide memory boundaries of extensions whenever they are installed in the system. The hardware anchor will then record this information and block all of the extension's operations at the architectural level that violate the security policies.

The system operation checker performs security validations to make sure OS extensions operate within the restricted boundaries defined by the security policies. The checker operates in a preventive mode so that any operations issued by kernel extensions will be checked for trustworthiness before they are performed. Several types of security checks can be performed. For example, the checker can monitor reads and writes in kernel space and calls to kernel functions.

OS kernel security instructions are the software counterpart of the hardware anchor. The OS kernel is modified to include calls to the hardware at specific points in its execution. For example, immediately after boot, the OS will perform calls to the hardware to pass kernel function addresses. The kernel also calls the hardware every time a new extension is installed/uninstalled to pass/remove its boundaries in main memory. It also calls the hardware to pass addresses of functions added by the extension. Whenever an extension allocates memory, the kernel calls the hardware to update the extension boundary. A diagram of the proposed hardware anchor enhanced architecture is shown in Figure 1.

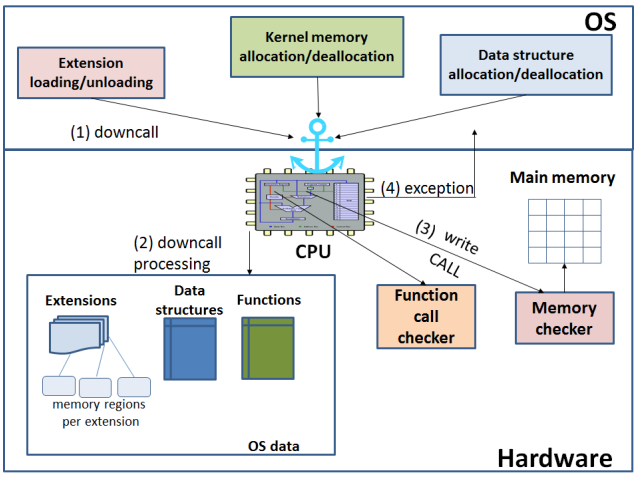


Figure 1. Architecture - High level view.

3. EMULATOR-BASED PROTOTYPE

The immune system-inspired kernel defense approach involves an OS that directly downcalls the hardware to pass information about loading, unloading and memory boundaries of extensions. Upon being processed by the CPU these downcalls are forwarded to handlers at the architecture layer, which are also responsible for maintaining the information passed. Figure 1 shows Ianus' high level view, which has the following key features: OS downcalls, downcall processing handlers, extensions' information kept in the architecture layer, and a checker for memory writes and function calls.

Downcalls are direct calls from the OS to the CPU (*Step 1* in Figure 1) and can have a variable number and type of parameters. Ianus has downcalls for extension loading, unloading, and dynamic allocation and deallocation of kernel memory. Every time an extension is loaded, the OS downcalls the CPU to pass the extension's name and the address and size of its object code. The extension's name uniquely identifies it in the system. When an extension allocates memory, the OS downcalls the CPU passing information about the address and size of the area. Memory dynamically allocated by extensions, despite being part of the kernel, do not receive the same level of protection given to original kernel data areas. The security policy adopted is to not allow kernel code and data being overwritten (bypassing kernel exported functions) by extension's instructions, which are considered low integrity. However, extensions cannot be prevented from writing into their own allocated memory areas, which requires tracking at the architecture layer. When an extension frees memory, the OS downcalls the CPU to provide it with the memory region address. Memory deallocated by an extension is considered again a high-integrity area of kernel space.

Upon receiving a downcall the CPU delegates its processing to specific handlers (*Step 2* in Figure 1), which create objects representing extensions and their attributes in the architecture layer. Extensions' memory boundaries (a range of linear addresses) are kept in an internal hash table per extension at the architecture layer. When an extension is unloaded the handler destroys the corresponding extension's object and removes the extension's corresponding linear addresses from the extension's hash table. Whenever kernel memory is allocated the handler checks if the instruction performing the allocation belongs to any of the extensions and if it does, the handler inserts this area into the extension's hash table. Finally, when kernel memory is deallocated, the handler

checks if the area belongs to any of the extensions active in the system and if it does, this memory region is removed from the extension's hash table of linear addresses.

3.1 Assumptions and Threat Model

This paradigm assumes an active OS which, like the immune system, is in charge of its own protection against its community of extensions with the support of the architecture layer. It is assumed that most kernel extensions are benign, but a small fraction of them will attempt to compromise the kernel and execute malicious actions.

It is also assumed an establishment time immediately after boot and all code and data present in the system before it are considered trustworthy. All extensions installed in the system are monitored, but they do not suffer any restriction on their execution, as long as they do not attempt to bypass kernel exported functions and write into the kernel code and data segments.

3.2 Implementation

An emulator-based proof-of-concept prototype, Ianus, was implemented to evaluate this architecture. Ianus used the Bochs x86 32-bit emulator as the architecture layer and Linux Ubuntu 10.04 kernel version 2.6.32 as the guest OS. Bochs was used due to its flexibility for performing architectural changes. The modifications in the guest OS consisted of a total of seven downcalls added to the kernel as assembly instructions. Bochs was extended with downcall processing handlers, an anchor instruction (downcall), data structures for keeping extensions' attributes and range of memory areas, and a checker invoked in all functions performing writes in main memory.

3.2.1 OS Downcalls and Handlers

The downcalls are implemented as an unused software interrupt in the Intel 32-bit x86 architecture (unused vector 15). The `INT` n instruction generates a call to the interrupt or exception handler specified by the destination operand. This operand (the interrupt vector) is an 8-bit number from 0 to 255, which uniquely identifies the interrupt. Vector 15 is reserved by Intel and is not in use. The `INT` n instruction was modified to handle vector 15 and this new software interrupt is handled similarly to how system calls are processed with parameters passed in general purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, and EBP).

The extensions' downcalls were placed in the system calls `sys_init_module` and `sys_delete_module`. It was necessary to insert two different downcalls in `sys_init_module` because during extension loading, after the initialization function is invoked, the memory area corresponding to the extension's init part is freed. The first downcall is placed after the extension's initialization function is invoked and passes the extension's name, and the address and size in bytes of the extension's init and core parts. The corresponding handler adds the memory range of the init and core parts (as linear addresses) into the extensions' hash table at the architecture layer.

After `sys_init_module` invokes the initialization function, a second downcall signals the architecture layer that the extension's init part will be freed. The corresponding handler removes the init memory region from the extensions' hash table. A downcall is also inserted in the system call `sys_delete_module` (invoked when an extension is unloaded) to remove the extension's information (attributes and memory regions) from the architecture layer.

Downcalls were also placed into the kernel functions `kmalloc()` and `vmalloc()` to handle memory dynamically allocated by extensions. The addresses of the callers of these allocation functions were obtained using the `__builtin_return_address` gcc hack to the Linux kernel [17], and allowed the architecture layer handlers to discover whether the caller function belonged to any of the active extensions. This strategy allows the distinction between memory allocations made by an extension and by the original kernel. If the address of the caller belongs to any of the extensions, the handler adds this newly allocated memory region to the extensions' hash table. Downcalls were also inserted in the corresponding deallocation functions `kfree()` and `vfree()`. The corresponding downcall handlers check whether the caller's address belongs to any of the active extensions tracked and if it does, remove the freed memory range from the extensions' hash table.

The downcall handlers at the architecture layer must translate virtual addresses from the OS into linear addresses. Each virtual address is represented by a segment and an offset inside this segment. The virtual addresses are included in the machine language instructions to specify the address of an operand or instruction. For example, in the assembly instruction `MOVE EDI,[EBX]`, the content of memory location given by register EBX is stored into register EDI. In this case, register EBX contains the offset of a virtual address in a particular segment. However, the security mechanisms at the architecture layer deal with linear addresses. In the Intel x86 architecture (used in this work) a linear address is a 32-bit number used to address a memory range up to 4 GB (addresses 0 to $2^{32}-1$).

Linux employs a limited form of segmentation by using three registers (`cs`, `ds`, and `ss`) to address code (CS), data (DS) and the stack (SS) segments. Processes running at user-level mode use these registers to address respectively the user code, data and stack segments. Code executing at kernel-level use these registers to address the kernel data, code and stack.

Each handler, upon receiving a virtual address from the OS in one of the general purpose registers must translate it into a linear address. The virtual address (segment and offset) is forwarded to the segmentation unit in the architecture layer and translated into a 32-bit linear address that can be used to index the extensions' hash table. Downcalls passing memory addresses can refer to data structures stored in the kernel DS or code in the kernel CS. For instance, the name of an extension or the address of a dynamically allocated memory region are located in the kernel DS. An extension's core and init parts reside in the kernel CS.

3.2.2 OS and Downcall Integrity

A key requirement of the proposed architecture is to guarantee the integrity of the OS downcalls. A kernel extension does not have privileges to issue downcalls and should not tamper with downcalls issued by the original kernel. This policy prevents a malicious extension from issuing a bogus downcall, or tampering with information passed by the original kernel to the architecture layer. These goals are addressed through the verification of all writes into kernel code and data segments and the origin of a downcall instruction.

The first security policy adopted is that kernel extensions are not allowed to perform write operations into the kernel code and data segments. The architecture layer contains a module for checking the validity of all write operations performed in the kernel code and data segments using Algorithm 1. This check is performed immediately before an instruction attempts to write a value into a memory location ($mAddr$). The architectural functions that perform writes into memory were instrumented to invoke the checker before any write is performed in main memory.

Whenever a write operation is about to be performed, it is checked whether the write is being attempted at kernel mode. This is done by checking the CPL value, which is represented by a 2-bit field in the `cs` register. Then it is checked whether the linear ad-

Algorithm 1 Checker for the validity of write operations in main memory

Input: $mAddr$, the address to be written in memory and $iAddr$ the address of the instruction performing the store operation.
Output: An exception is raised if the write is invalid.

```
if (CPL == 0) && (ISADRESSFROMEXTENSION(iAddr)) &&
(!ISADDRESSFROMEXTENSION(mAddr) then
    if (segment ≠ SS) then
        EXCEPTION
    else
        if (lAddr > EBP) then
            EXCEPTION
        end if
    end if
end if

function ISADDRESSFROMEXTENSION(addr)
    for i = 0 to Extension.length do
        if Extension_i[addr] then
            return true
        end if
    end for
end function
```

dress of the instruction storing data ($iAddr$) into memory belongs to any of the extensions' memory region monitored at the architecture layer. Next, it is checked whether the memory address being written ($mAddr$) belongs to an extension itself, which is not considered a security violation. Following, the segment being written is checked. If the write is attempted at the data or code segments, an exception is raised because it is considered a security violation (*Step 4* in Figure 1). If the segment is SS (stack) it is checked whether the target address is higher than the current value of register EBP. If it is higher, this is an indication of a stack corruption attempt and an exception is raised. The kernel has the discretion to treat this policy violation the way it finds most appropriate. One possible action is to unload the offending extension from the system.

The integrity of downcall instructions is checked with architectural support. Upon execution of the `INT $15` instruction it is checked whether the instruction bytes come from an extension. This is done by hashing the current instruction linear address to the hash tables that maintain the memory regions for extensions. If the instruction belongs to any of the extensions, an exception is raised.

3.2.3 Monitoring Non-Exported Function Calls

In the proposed architecture extensions interactions with kernel functions are monitored. Extensions are supposed to only invoke kernel exported functions. A function call checker intercepts all `CALL` instruction invoked by an extension and verifies whether its target address belongs to an exported function. If the target address of the `CALL` instruction corresponds to a non-exported function (from the kernel or other extensions) or even to an address that does not correspond to a function (an indication of a return-oriented attack [18]), the CPU raises an exception.

The addresses of all kernel functions (exported and non-exported) are obtained from the System.map file created during kernel compilation and made available to the architecture layer. Functions are distinguished from static data structures by the symbol type or by where the symbol is located. For example, if the symbol belongs to the text section, it corresponds to a function. Exported symbols are identified by their prefix: in System.map all exported symbols start with the prefix *ksymtab*.

Extensions also contain their own symbols (static data structures and functions) and can export some of them. Whenever an extension is installed, the OS extracts information about its symbols (names, types, virtual addresses and whether they are exported) and executes a downcall to make this information available to the architecture layer. Extensions' symbols information in Linux can be obtained through the `sym` field in the `module struct` (exported symbols), and through the extension's symbol tables contained in the `__ksymtab`, `__ksymtab_gpl` and `__kstrtab` sections of the extension code segment (all symbols) [19]. When an extension is unloaded, its symbols are removed from the kernel and from the architecture layer. The complete information about kernel and extension's symbols is kept at the architecture layer in a hash table indexed by the symbol's address (linear address).

Whenever the CPU is about to execute a `CALL` instruction, the following checks are performed. First the function call checker determines if the function invocation is being performed at kernel level. Following, it determines whether the `CALL` function comes from a extension's code. Next, it is determined whether the `CALL` target can be found in the symbols table at the architecture layer and whether the symbol is exported. If the symbol is not exported it is verified whether the extension invoking the function actually owns it, which is allowed. If those checks do not pass, an exception is raised. Another possibility is the target of the `CALL` instruction not belonging to any symbol in the kernel or in its extensions. In this case, the symbol is not a function, which is an indication of a return-oriented attack [18].

3.2.4 Monitoring Extensions' Access to Data Structures

Algorithm 1 can be extended for fine-grained monitoring of how extensions access kernel data structures. Data structures allocated by the kernel should only be modified by extensions indirectly through kernel exported functions. A direct access to a kernel data structure is considered a security vulnerability. Knowledge about the boundaries of all dynamic and static data structures in main memory is required for the monitoring of read/write access to kernel data structures by extensions. The boundaries of kernel static data structures can be found in the System.map file as explained in section 3.2.3.

Dynamically allocated data structures, however, cannot be found in a static file such as System.map as they are unknown at compilation time. In Linux dynamic data structures are created via the slab allocator [20], which divides groups of objects into *caches* that store a different type of object. For example, the `task_struct` cache is a group of data structures of this type. In the slab allocator, three functions control the dynamic allocation/deallocation of kernel data structures: `kmem_cache_create`, `kmem_cache_alloc`, and `kmem_cache_free`. The first function, `kmem_cache_create` is invoked only once when the system is booted and creates a new cache for a particular type of data structure, *e.g.*, `task_struct`. The second function `kmem_cache_alloc` is used to allocate memory for a new data structure for the cache type passed as argument (the cache type determines the type of the data structure). For example, the cache `mm_struct` is used to allocate data structures of that corresponding type. The third function `kmem_cache_free` is used to return the memory area allocated to a data structure to the corresponding cache when it needs to be deallocated.

In Ianus the slab allocator's functions from the OS are instru-

mented to inform the architecture layer whenever a data structure of a particular type is created and freed. Two hash tables at the architecture layer are used to keep this information. The first, the *Data Structure Type* is indexed by the data structure's type (the cache name) and also contains its size in bytes. Whenever a new cache is created, a new entry is inserted in this table. The second hash table, called *Data Structure*, is indexed by the data structure's linear address and keeps up-to-date information about the active data structures in the kernel. Whenever a kernel data structure is created, a new entry is inserted in this table. When a data structure is deallocated its entry is removed from the table.

In this extension, Algorithm 1 works as follows. After verifying that the segment being written is not the stack ($segment \neq SS$), it checks whether the segment being written is the data segment (DS). If it is, the memory checker records in a file, which is available to the system administrator, the type of the data structure being accessed.

3.3 Aspects of the Hardware Implementation

In the current hardware implementation, the anchor is embedded in the open-source SPARC V8 processor, as per the diagram shown in Figure 2. It includes the cross-boundary interface, the software security checker, and hash tables to store system level intelligence, such as extensions boundaries and addresses of kernel functions and static data structures. The hardware anchor monitors all traffic through the processor and takes control when a recognized downcall is issued through the interface. When a recognized downcall is fetched, the anchor halts the Integer Unit (IU), transfers the control of incrementing the program counter to the Anchor Control Unit (ACU), and ports the instruction-cache (I-cache) and data-cache (D-cache) data wires to the ACU. In this way, the SPARC architecture specific interrupt handling is suspended and subsequent fetch, decode, and execution stages are controlled by the ACU only.

Many OS downcalls will pass virtual addresses through the anchor, for instance the initial virtual address of a recently loaded extension. Therefore, the ACU also acts to control the Memory Management Unit (MMU) by severing the MMU input/output lines from the processor and porting them to the anchor. Subsequent MMU virtual-to-physical translations will operate on virtual addresses passed by the OS through the anchor. The physical addresses are returned to the ACU to be stored as system level intelligence needed for enforcement of security policies. Once the ACU finishes operating on the passed OS intelligence, control is given back to the IU and the input/output lines are returned to normal functionality. The ACU acts as the processor control unit by directing data flow when an OS downcall is issued. The hash tables that store system-level intelligence are part of the hardware anchor and are not accessible to any other component of the processor.

4. EXPERIMENTAL EVALUATION

This section presents the results of the experiments validating Ianus. All experiments were executed in an Intel quad core 3.8 GHz with 16 GB RAM and running Linux Ubuntu 12.04 as host OS. Each performance experiment was executed ten times and the results were averaged. The evaluation assessed Ianus security and performance. The security analysis investigated whether Ianus guaranteed the integrity of the downcalls and the information passed through them, the protection level against kernel-level malware (rootkits), and whether or not it caused disruption in the normal operation of benign modules (false positives). The performance analysis investigated Ianus' overhead to the system (OS, architecture layer and the two combined).

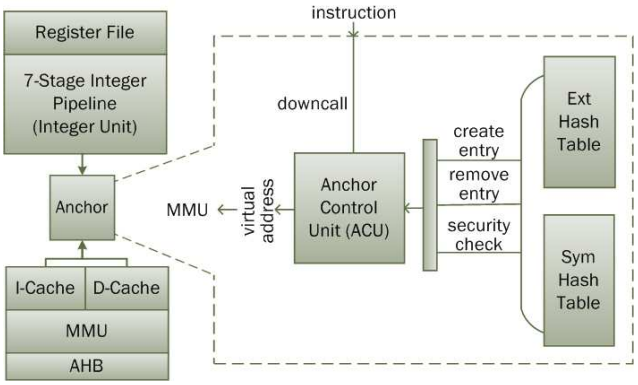


Figure 2. Hardware Anchor - Detailed view.

4.1 Security Analysis

Ianus' security was analyzed against real rootkits that exercised the following security concerns: (i) tampering with kernel code and data, (ii) tampering with downcall parameters, and (iii) issuing bogus downcalls. False positives were also evaluated with benign kernel modules and drivers. Table 1 shows the rootkits tested in this evaluation. The last two rootkits in the table were implemented by the authors and this section details their interactions with Ianus' security mechanisms.

4.1.1 Tampering with Kernel

The authors implemented a kernel rootkit (*General Keylogger*) as a loadable kernel module (LKM) that attempts, like most rootkits, to corrupt the OS's system call table. In its initialization function the rootkit replaces a pointer to a legitimate system call function with a pointer to its malicious version of the system call. This is a very common type of kernel attack in spite of recent Linux versions attempting to make the system call table inaccessible to kernel extensions. The rootkit has keylogging capabilities and was based on the system call hijacking approach described in [21]. The rootkit hijacks the system call table by first locating its address in the kernel through brute force and writes into the system call table by first setting the CR0 register's first bit to 0, which changes the CPU from protected to real mode. After tampering with the system call table, the rootkit puts the CPU back to protected mode. These actions were done with the kernel functions `read_cr0` and `write_cr0`.

When this malicious extension is loaded, the architecture layer has complete information about its boundaries in memory. When the initialization function is invoked, one of its instructions attempts to perform a write operation in an area which is part of the kernel data segment. The goal is to overwrite this area with a malicious address into one of the slots of the system call table. The write operation is checked at the architecture level and it is detected that (i) it is performed in kernel mode, (ii) the target memory location is in the kernel data segment, (iii) the instruction bytes come from the text of the extension's init part, and (iv) the memory area being written is not part of any extension's dynamically allocated memory region. The write operation is aborted (thus preventing any compromise) with the CPU raising an exception handled by the OS. All other rootkits that operate by tampering with the system call table were stopped similarly.

4.1.2 Issuing a Bogus Downcall

Here the goal was to evaluate whether or not kernel-level malware could issue bogus downcalls to the CPU. The authors im-

Table 1. Rootkits used in the security evaluation.

Name	Attack approach	Functionalities
KBeast	system call hooking	network activity, process and file hiding, keylogging, anti-delete, anti-kill and anti-remove
bROOTus	system call hooking	covert channel, module, process and file hiding, keylogging and privilege escalation
LVTES	system call hooking	keylogging
rkit	system call hooking	privilege escalation
kbd_notifier	registration of malicious function with notifier chain	keylogging
Bogus Downcall	direct invocation of INT $15	issuing of a bogus downcall to the hypervisor
General Keylogger	system call hooking	keylogging

Table 2. Extensions' access to functions in kernel space - Benign drivers and modules.

Module	Number of Exported Symbols	Security issues
Drivers loaded during boot		
i2c_piix4	0	none
serio_raw	0	none
floppy	0	none
parport_pc	2	invocation of add_dev (from parport)
parport	32	invocation of parport_pc_data_forward (from parport_pc)
psmouse	0	none
ppdev	0	none
lp	0	none
8390	11	invocation of __ticket_spin_unlock (from kernel)
ne2k_pci	0	none
Benign extensions from SourceForge		
frandom	1	none
tier	0	none
rxdsk	N/A	none
usb_vhci_iocifc	N/A	none
usb_vhci_hcd	N/A	none
tty0tty	0	none

plemented a rootkit that attempted to perform a downcall passing fabricated parameters to the CPU. The downcall was issued in the rootkit's initialization function. As in the previous examples, immediately before the initialization function is invoked the architecture layer is keeping track of all memory areas in use by the extension. The extension's initialization function is invoked and issues a downcall causing the CPU to execute instruction `INT $15`. Upon executing the interrupt instruction the origin of its bytes is verified at the architecture layer by hashing the instruction linear address to the hash tables that maintain the extensions' memory regions. The hash is a hit, which shows that the downcall is being issued by an extension, and an exception is raised.

The only rootkit Ianus was not able to contain was the `kbd_notifier` keylogger [22], which operates without the need to tamper with kernel code and data. It is a stealthy rootkit that works by registering a malicious function with the kernel keyboard notifier chain, which is invoked whenever a keyboard event occurs and allows the malware to record the keys pressed by a user at the kernel level.

4.1.3 Extensions' Access to Kernel Functions

Another important aspect of the evaluation was to analyze Ianus' behavior when executing kernel extensions. The common assumption is that benign extensions will only access kernel exported functions to perform their tasks. Table 2 illustrates the evaluation done with benign drivers installed during boot and benign extensions from SourceForge [23]. From the set of extensions analyzed, three benign drivers invoke non-exported functions from other extensions and the kernel. These issues caused the CPU to raise an exception to the OS.

Table 3 shows how real rootkits access kernel functions. In general, rootkits need to invoke a great number of kernel non-exported functions to operate. The only exception was the `kbd_notifier` keylogger.

4.1.4 Extensions' Access to Kernel Data Structures

Table 4 shows that, in general, benign extensions and drivers do not directly access kernel data structures. The only exceptions were the `parport` and `floppy` drivers, which access the `task_struct` of a process. Rootkits, on the other hand, need to tamper with some kernel data structure to succeed and the vast majority of them tamper with the system call table. The only exception is the `kbd_notifier` keylogger [22], which operates without the need to tamper with kernel code and data.

4.2 Performance Analysis

This section analyzes Ianus' performance impact in the whole system using system microbenchmarks from Unixbench [24] and a subset of the SPEC CPUINT2006 benchmark suite [25]. The execution times were normalized to the execution time of the system without any modifications to the OS and the Bochs x86 emulator. Using the unmodified Bochs as a basis for normalization allowed

Table 3. Extensions' access to functions in kernel space - Rootkits.

Module	Number of exported symbols	Non-exported functions invoked
rkit	0	native_read_cr0(kernel)
bROOTus	0	__ticket_spin_unlock(kernel) sys_read(kernel) sys_getdents64(kernel) sys_readdir(kernel) udp4_seq_show(kernel) tcp4_seq_show(kernel)
KBeast	0	native_read_cr0(kernel) sys_read(kernel) sys_write(kernel) sys_getdents64(kernel) sys_open(kernel) sys_unlink(kernel) sys_unlinkat(kernel) sys_rename(kernel) sys_rmdir(kernel) sys_delete_module(kernel) sys_kill(kernel)
kbd_notifier	0	none
LVTES	0	native_read_cr0(kernel) sys_read(kernel) sys_getdents64(kernel) sys_write(kernel) sys_open(kernel) sys_close(kernel)

Table 4. Extensions' access to kernel data structures.

Module	Access type	Data Structure
Drivers loaded during boot		
i2c_piix4	N/A	none
serio_raw	N/A	none
floppy	Read	task_struct
parport_pc	N/A	none
parport	Read	task_struct
psmouse	N/A	none
ppdev	N/A	none
lp	N/A	none
Benign modules from sourceforge		
frandom	N/A	none
tier	N/A	none
rxdsk	N/A	none
usb_vhci_iocifc	N/A	none
usb_vhci_hcd	N/A	none
tty0tty	N/A	none
Rootkits		
rkit	Write/Read	sys_call_table
brootus	Write/Read	sys_call_table
ipsecs_kbeast_v1	Write/Read	sys_call_table
new_read	Write/Read	sys_call_table
kbd_notifier	N/A	none
hijack_syscall	Write/Read	sys_call_table
lvtes	Write/Read	sys_call_table

the evaluation to be focused on the actual overhead of the security mechanisms and not on the Bochs overhead as an Intel x86 emulator.

Figure 3(a) shows the performance overhead of the OS downcalls during normal OS operation for Unixbench. These benchmarks exercised `execl` calls, file reads and writes (`fsbuffer`, `fsdisk` and `fstime`), pipe throughput (`pipe`) and pipe-based context switch (`context1`), process creation (`spawn`) and system call overhead (`syscall`). For these experiments Unixbench ran with the modified and the unmodified version of the guest OS. The goal here was to isolate the performance overhead of downcall issuing at the OS for intense system-level activity. Figure 3(a) shows that the overhead of downcall issuing at the OS is negligible (average 2%) for most of the system benchmarks.

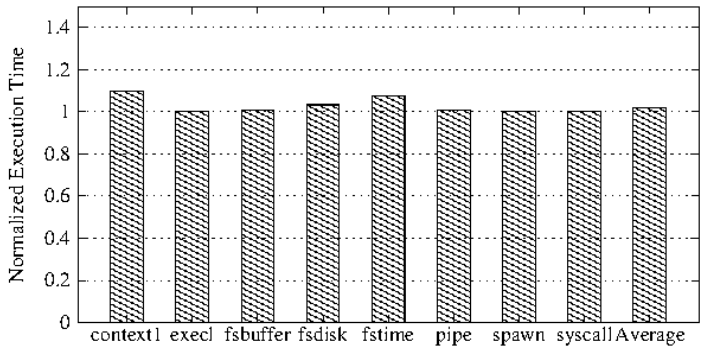
Figure 3(b) shows the performance analysis for five benchmarks from SPEC CPUINT2006. The overhead was measured for two different situations. In the first (*OS Downcalls*), the system was running the modified version of the OS containing all downcall issuing. Here the goal was to evaluate the overhead to the OS for a CPU intensive benchmark. The second setting (*Downcall handling*) had the same configuration as the first, but now the downcalls were being processed by the handlers at the architecture layer. Figure 3(b) corroborates Figure 3(a) results in the sense that the downcall issuing overhead at the OS is very low. Downcall processing caused an increase of 12% on average to the execution time of the SPEC CPUINT benchmarks at the architecture layer. The overhead of 12% is low when we consider that certain types of applications that require a high level of security, (*e.g.*, a power grid or a server at a national security agency), can trade performance for security. The hash tables at the architecture layer required less than 5 MB of main memory.

5. DISCUSSION

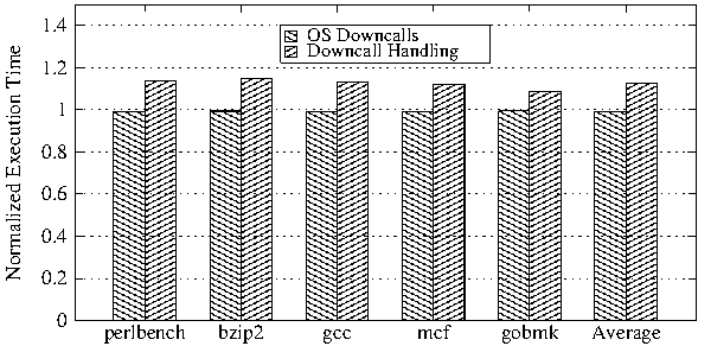
Similar to OS kernels, modern hardware design has increasingly relied on third party extensions. Under the current hardware design methodology, only the core processor/microprocessor designed in-house goes over full functionality and security testing and is considered trustworthy. Peripheral IP (Intellectual Property) modules and firmware extensions are often not fully tested to save cost, to shorten time-to-market, or to increase the reusability of the designed systems. However, third-parties IP cores can contain malicious logic and hardware also needs to co-exist with untrustworthy but needed extensions [26, 15, 27].

The proposed architecture can be broadened to include the hardware itself in the protection mechanisms. Hardware policies can be added to the architecture so that the hardware can be protected holistically with the aid of OS intelligence. OS intelligence can help define the trusted boundaries for legitimate operations of each IP module as well as the nominal time lengths of bus occupation in order to prevent false positives when detecting DoS (denial-of-service) style Trojans. For example, if data encryption is performed, then the key, the plaintext, and the intermediate results are sensitive information that only the cryptographic co-processor can access. The addresses of the sensitive information in memory are then passed to the hardware anchor as part of the OS intelligence. All other IP modules are prohibited from visiting the memory space where the sensitive data is stored. Including the hardware in the security policies also addresses concerns of HW-SW integration vulnerabilities: modern computer systems are not prepared to counter attacks that rely on the interaction between HW-SW pairs.

Another key innovation of the proposed HW-SW collaborative architecture is the ability to build a system with any combination



(a) Unixbench



(b) SPEC CPU INT

Figure 3. Performance overhead.

of OS and hardware security policies chosen by a system designer or integrator at the time the system is built. Each policy implementation contains an OS and a hardware component. At the hardware side, the security policies are all part of the anchor functionality, but are only effective if explicitly applied by the OS through the anchor. Under this architecture a security policy can be explicitly disabled by the OS due to performance or false positive concerns.

Benign kernel extesions coming from the network that modify kernel data structures without using an exported kernel function will have their actions reported as a kernel integrity violation. This situation can be overcome if such extensions are obtained from a network interface that the system considers high integrity or if they are installed in the system before the establishment time.

Attacks that do not need to write into kernel space to succeed [28] or that compromise a hypervisor [29, 30] or that write into kernel memory through trusted code paths in the kernel are beyond of the scope of Ianus. Further, the target of `JMP` instructions, which can be leveraged by a rootkit to bypass Ianus' checks, are not currently checked in the prototype.

6. RELATED WORK

A great amount of research has been done regarding hypervisor-based kernel protection and security solutions leveraging traditional introspection. This section discusses relevant work in VM-based introspection, kernel integrity defenses, and hardware support for system security.

6.1 Virtual Machine Introspection

VM-based intrusion detection systems leverage introspection in two ways: passive [5, 31] and active [9, 32, 33]. Passive introspection accesses a guest OS memory to reconstruct its state and abstractions. The OS state is recreated from low level data such as memory page frames.

Active introspection addresses better the semantic gap problem by allowing a more up-to-date view of a guest OS state. Xenprobes [32] and Lares [9] place hooks inside the guest OS to intercept some key events, and invoke a security tool residing at VM level to treat the event. HIMA [33] is a VM-based agent to measure the integrity of a guest OS by intercepting system calls, interrupts and exceptions. All of these approaches (passive and active) consider the guest OS untrustworthy and do not *actively* interact or leverage it in the introspection process. This limits the amount and variety of system-level information that can be collected. L4 microkernels ([34]) functioning as a hypervisor also requires a L4-aware OS which downcalls the hypervisor and resembles the idea of downcalls of this work. The OS modifications turn system calls, memory and hardware accesses into calls to the hypervisor. Differently from L4 microkernel, the purpose of the OS downcalls in Ianus is exclusively for aiding security.

Recently, researchers have been working on better ways to perform traditional introspection, which is an error-prone and manual process. Chiueh *et al* [35] propose to inject stealthy kernel agents to a guest OS to enable virtual appliance architectures to perform guest-OS specific operations. Virtuoso [11] creates introspection tools for security applications with reduced human effort. SIM [36] enables security monitoring applications to be placed back in the untrusted guest OS for efficiency. It still suffers from the same semantic gap challenges as traditional introspection approaches because it was not designed to rely on data from the guest OS. Fu and Lin [37] apply system-wide instruction monitoring to automatically identify introspection data and redirect it to the in-guest kernel memory. A limitation is that certain types of data cannot be redirected, limiting the amount of guest OS information that can be obtained. Other line of work [38, 39], based on process migration, proposes to relocate a suspect process from inside the guest OS to run side by side with an out-of-VM security tool. The challenge is

that some processes are not suitable for migration.

6.2 Kernel Integrity Defense

Many authors have previously addressed kernel protection. The works focusing on prevention use some form of code attestation or policy to decide whether or not a piece of code can be executed in kernel mode. SecVisor [7] employs a hypervisor to ensure that only user-approved code executes in kernel mode: users supply a policy, which is checked against all code loaded into the kernel. NICKLE [8] uses a memory shadowing scheme to prevent unauthorized code from executing in kernel mode. A trusted VM maintains a shadow copy of the main memory for the guest OS and performs kernel code authentication so that only trusted code is copied to the shadow memory. During execution, instructions are fetched only from the shadow memory. Code attestation techniques [6] verify a piece of code before it gets loaded into the system

Some approaches can offer some protection against non-control data attacks [40] that tamper with kernel data structures by directly injecting values into kernel memory. Livewire [5] is a VM architecture with policies for protecting certain parts of the kernel code section and the system call table. KernelGuard [41] prevents some dynamic data rootkit attacks by monitoring writes to selected data structures. Oliveira and Wu [42] used a performance expensive dynamic information flow tracking system (DIFT) and a set of shadow memories to prevent untrusted bytes to reach kernel space.

There are also many works addressing detection. Copilot [43] uses a PCI add-in card to access memory instead of relying on the kernel. Lycosid [44] and VMWatcher [45] perform detection based on a cross-view approach: hiding behavior is captured by comparing two snapshots of the same state at the same time but from two different points of view (one from the malware and the other not). OSck [46] protects the kernel by detecting violation in its invariants and monitoring its control flow.

The difference between this work and previous research in VM introspection and OS protection is that here the OS, like our immune system, has an active role in its protection against compromise from kernel extensions. The architectural layer acts only as a collaborative peer leveraging the key information about extensions passed by the OS to monitor the interactions of extensions and the original kernel. Having the OS in charge of monitoring itself streamlines kernel defense when compared to related work based on manual and error-prone introspection.

6.3 Extension Isolation

A great body of work in the literature focus on isolating or confining the effects and execution of device drivers and modules. Nooks [47] introduced the concept of shadow drivers and isolate them in a separate address space so that they can be recovered after a fault. HUKO [48] and Gateway [10] built on this idea by leveraging hypervisor support to protect the approach that confine modules in a separate address space from a compromised OS. Ianus's goals are similar to Nooks, HUKO, Gateway in the sense of protecting the kernel from malicious or misbehaving extensions. However, Ianus does not attempt to physically isolate the kernel from its extensions, but provide a way for them to co-exist. This provided much more flexibility to the system. For example, Section 4 showed that many drivers do invoke kernel (and other module's) non-exported functions and their execution would be disrupted in HUKO or Gateway. Ianus can be fine tuned to allow more privileges to certain modules known to be benign. Also, the proposed architecture can be extended to include the hardware itself in the protection mechanisms.

Some lines of work advocate running drivers partially or entirely in user space. Ganapathy *et al* [49] introduced the idea of micro-drivers in which drivers execute partly in user space and partly in the kernel. Nexus [50] and SUD [51] confine buggy or malicious device drivers by running them in user-level processes. Some works attempt to achieve driver isolation in software, such as SFI [52], where the object code of untrusted modules are rewritten to prevent their code from jumping to an address outside of their address space. The proposed architecture allows extensions to be executed without any modifications.

6.4 Hardware Infrastructure for System Security

Besides the software approaches, researchers have also proposed to rely on enhanced hardware infrastructure to protect system security. Hardware architectures to prevent memory corruption bugs and to prevent information leakage were developed for information flow integrity within computer systems. However, vulnerabilities were detected in these methods through which attackers can bypass the protection schemes. Chen *et al* proposed a HW-SW architecture supporting flexible and high-level software policies for sensitive data protection. Similar approaches were also developed in the area of embedded systems, where limited on-chip resources are available [53, 54, 55, 56, 57, 58, 59, 60].

The main difference of these works and the proposed architecture is that these approaches work in isolation with the hardware and the OS. As discussed in Section 5, this collaborative architecture can be extended to include security policies to protect the hardware itself and offer a flexible set of security policies that can be customized during system build time. The proposed architecture offers a system builder great flexibility for balancing security, performance and false positives.

7. CONCLUSIONS

Current OS defense approaches are single-layered and operate in isolation with the hardware, failing to recognize its key capabilities for enforcing security policies. HW-SW cooperation is a promising approach to improve system trustworthiness because each layer has access to a distinct set of intelligence and functionality, which in combination can offer higher security guarantees than when operating independently. This paper discussed a HW-SW architecture for allowing an OS to safely co-exist with its extensions. This architecture is inspired by the immune system collaborative defense mechanisms that maintain a symbiotic relationship with its needed but untrustworthy microbiota.

A proof-of-concept prototype for this architecture, named Ianus, was implemented with Linux and the Bochs x86 emulator as the collaborative architecture layer. Ianus's was studied with several real rootkits and benign extensions. In the experiments all malicious rootkits were stopped and Ianus caused no false positives for benign modules. Ianus' performance was analyzed with system and CPU benchmarks and the system overhead was low (12% on average).

Acknowledgments

This research is funded by the National Science Foundation under grant CNS-1149730.

8. REFERENCES

[1] "Symantec Internet Security Threat Report 2013 (`http://www.symantec.com/security_response/publications/threatreport.jsp`)."

[2] J. Carr, *Cyber Warfare*. O'Reily, 2011.

[3] A. Kadav and M. M. Swift, "Understanding Modern Device Drivers," *ASPLOS*, 2012.

[4] L. Sompayrac, *How the Immnune System Works*. Wiley-Blackwell, 4th ed., 2012.

[5] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," *Network and Distributed System Security Symposium*, 2003.

[6] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," *ACM SOSP*, 2003.

[7] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," *ACM SOSP*, 2007.

[8] R. Riley, X. Jiang, and D. Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing," *RAID*, 2008.

[9] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring using Virtualization," *IEEE Symposium on Security and Privacy*, May 2008.

[10] A. Srivastava and J. Giffin, "Efficient Monitoring of Untrusted Kernel-mode Execution," *NDSS*, 2011.

[11] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," *IEEE Security and Privacy*, 2011.

[12] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "DKSM: Subverting Virtual Machine Introspection for Fun and Profit," *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp. 82–91, 2010.

[13] "bochs: the Open Source IA-32 Emulation Project (http://bochs.sourceforge.net)."

[14] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pp. 697–708, 2013.

[15] S. Skorobogatov and C. Woods, "Breakthrough Silicon Scanning Discovers Backdoor in Military Chip," in *Cryptographic Hardware and Embedded Systems (CHES 2012)*, vol. 7428 of *Lecture Notes in Computer Science*, pp. 23–40, Springer Berlin Heidelberg, 2012.

[16] A. Cui, J. Kataria, and S. Stolfo, "Killing the Myth of Cisco IOS Diversity: Recent Advances in Reliable Shellcode Design," in *USENIX Worshop on Offensive Technologies (WOOT)*, 2011.

[17] "GCC Hacks in the Linux Kernel (http://www.ibm.com/developerworks/linux/library/l-gcc-hacks/)."

[18] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," *ACM CCS*, pp. 552–561, 2007.

[19] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel, Third Edition*. O'Reilly, 2005.

[20] R. Love, *Linux Kernel Development*. Novell Press. 2005.

[21] "Syscall Hijacking (http://memset.wordpress.com/2011/03/18/syscall-hijacking-dynamically-obtain-syscall-table-address-kernel-2-6-x-2/))."

[22] "How to: Building Your Own Kernel Space Keylogger. (http://www.gadgetweb.de/programming/39-how-to-building-your-own-kernel-space-keylogger.html)."

[23] "SourceForge.net: Open Source Software (http://sourceforge.net)."

[24] "UnixBench (http://www.tux.org/pub/tux/benchmarks/)."

[25] "SPEC - Standard Performance Evaluation Corporation (http://www.spec.org/cpu2006/)."

[26] E. Greenbaum, "Open Source Semiconductor Core Licensing," *Harvard Journal of Law & Technology*, vol. 25, no. 1, pp. 131–157, 2011.

[27] A. Cui, M. Costello, and S. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," in *20th Annual Network & Distributed System Security Symposium (NDSS)*, 2013.

[28] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware Supported Rootkit Concealment," *IEEE Security and Privacy*, pp. 296–310, 2008.

[29] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, "SubVirt: Implementing Malware with Virtual Machines," *IEEE Security and Privacy*, May 2006.

[30] J. Rutkowska, "Subverting VistaTM Kernel For Fun And Profit," *Black Hat Briefings*, 2006.

[31] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection Through VMM-based "Out-of-the-Box" Semantic View Reconstruction," *ACM CCS*, 2007.

[32] N. A. Quynh and K. Suzaki, "Xenprobes, A Lightweight User-Space Probing Framework for Xen Virtual Machine," *USENIX Annual Technical Conference*, 2007.

[33] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "HIMA: A Hypervisor-Based Integrity Measurement Agent," *ACSAC*, 2009.

[34] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, (New York, NY, USA), pp. 207–220, ACM, 2009.

[35] T. cker Chiueh, M. Conover, M. Lu, and B. Montague, "Stealthy Deployment and Execution of In-Guest Kernel Agents," *BlackHat*, 2009.

[36] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure In-VM Monitoring Using Hardware Virtualization," *ACM CCS*, 2009.

[37] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," *IEEE Symposium on Security and Privacy*, May 2012.

[38] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, ""Process Out-Grafting: An Efficient 'Out-of-VM' Approach for Fine-Grained Process Execution Monitoring," *ACM CCS*, March 2011.

[39] Z. Gu, Z. Deng, D. Xu, and X. Jiang, "Process Implanting: A New Active Introspection Framework for Virtualization," *IEEE SRDS*, 2011.

[40] S. Chen, J. Xu, and E. Sezer, "Non-Control-Hijacking Attacks are Realistic Threats," in *USENIX Security*, 2005.

[41] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring," *International Conference on Availability, Reliability and Security*, 2009.

[42] D. Oliveira and S. F. Wu, "Protecting Kernel Code and Data with a Virtualization-Aware Collaborative Operating

System," *ACSAC*, December 2009.

[43] N. Petroni, T. Fraser, and W. A. Arbaugh, "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor," *USENIX*, 2004.

[44] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based Hidden Process Detection and Identification using Lycosid," *ACM VEE*, 2008.

[45] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," *ACM CCS*, pp. 128–138, November 2007.

[46] O. S. Hofmann, A. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring Operating System Kernel Integrity with OSck," *ASPLOS*, 2011.

[47] M. Swift, B. N. Bershad, and H. M. Levy, "Improving the Reliability of Commodity Operating Systems," *ACM SOSP*, pp. 207–222, 2003.

[48] X. Xiong, D. Tian, and P. Liu, "Practical Protection of Kernel Integrity," *NDSS*, 2011.

[49] V. Ganapathy, M. J. Renzelmann, M. M. S. Arini Balakrishnan, and S. Jha, "The Design and Implementation of Microdrivers," *ASPLOS*, pp. 168–178, Mar. 2008.

[50] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider, "Device Driver Safety through a Reference Validation Mechanism," *OSDI*, pp. 241–254, 2008.

[51] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in Linux," *USENIX*, 2010.

[52] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-based Fault Isolation," *ACM SOSP*, pp. 203–216, 1993.

[53] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pp. 85–96, 2004.

[54] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pp. 378–387, June 2005.

[55] M. Dalton, H. Kannan, and C. Kozyrakis, "Deconstructing Hardware Architectures for Security," in *5th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) at ISCA*, 2006.

[56] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee, "A Software-hardware Architecture for Self-protecting Data," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pp. 14–27, 2012.

[57] S. Mao and T. Wolf, "Hardware Support for Secure Processing in Embedded Systems," *Computers, IEEE Transactions on*, vol. 59, no. 6, pp. 847–854, 2010.

[58] M. Rahmatian, H. Kooti, I. Harris, and E. Bozorgzadeh, "Hardware-Assisted Detection of Malicious Software in Embedded Systems," *Embedded Systems Letters, IEEE*, vol. 4, no. 4, pp. 94–97, 2012.

[59] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August, "RIFLE: An architectural framework for user-centric information-flow security," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pp. 243–254, 2004.

[60] W. Shi, J. Fryman, G. Gu, H.-H. Lee, Y. Zhang, and J. Yang, "InfoShield: A Security Architecture for Protecting Information Usage in Memory," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pp. 222–231, 2006.

ABOUT THE AUTHORS:

Daniela Oliveira received her B.S. and M.S. degrees in Computer Science from the Federal University of Minas Gerais, Brazil, in 1999 and 2001, respectively, and the Ph.D. degree in Computer Science in 2010 from the University of California at Davis. She is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Florida. Her main research interest is interdisciplinary computer security, where she employs successful ideas from other fields to make computer systems more secure. Her current research interests include employing biology and warfare strategies to protect operating systems in cooperation with the architecture layer. She is also interested in understanding the nature of software vulnerabilities and social engineering attacks. She is the recipient of the 2012 NSF CAREER Award and the 2012 United States Presidential Early Career Award for Scientists and Engineers (PECASE).

Nicholas Wetzel received his B.S. degree in Computer Science from Bowdoin College in 2014. He worked as an undergraduate research assistant at Daniela Oliveira's cyber security lab at Bowdoin for one year before she moved to the University of Florida.

Max Bucci is currently a Computer Science major at Bowdoin College and expects to receive his B.Sc. degree in Computer Science in 2015. He worked as an undergraduate research assistant at Daniela Oliveira's cyber security lab at Bowdoin for one year before she moved to the University of Florida.

Jesus Navarro received his B.S. degree in Computer Science from Bowdoin College in 2013. He worked as an undergraduate research assistant at Daniela Oliveira's cyber security lab at Bowdoin for two years before she moved to the University of Florida. He works now as a software engineer at NVIDIA in Santa Clara, California.

Dean Sullivan received the B.S. degree in Electrical Engineering in 2013 and is currently pursuing the Ph.D. degree in Computer Engineering from the University of Central Florida. His current research focuses on designing security-enhanced computer architecture that protects both software and hardware. His research interests include computer architecture, secure HW/SW co-design, hardware security and trust, and VLSI design.

Yier Jin received his B.S. and M.S. degrees in Electrical Engineering from Zhejiang University, China, in 2005 and 2007, respectively, and the Ph.D. degree in Electrical Engineering in 2012 from Yale University. Since 2012, he has been an assistant professor in the EECS Department at the University of Central Florida. His research focuses on the areas of trusted embedded systems, trusted hardware intellectual property (IP) cores and hardware-software co-protection on computer systems. He is also interested in the security analysis on Cyber-Physical Systems (CPS) and Internet of Things (IoT) with particular emphasis on information integrity and privacy protection.