Guarded Modules: Adaptively Extending the VMM's Privileges Into the Guest

Kyle C. Hale and Peter A. Dinda
{*k-hale, pdinda*}*@northwestern.edu*
Department of Electrical Engineering and Computer Science
Northwestern University

Abstract

Executing VMM-provided code with privileged access to specific hardware and VMM resources within an untrusted guest operating system can enable new mechanisms to enhance functionality, performance, and adaptability. We present a software technique, guarded execution of privileged code in the guest, that allows the VMM to provide this capability, as well as an implementation for Linux guests in the Palacios VMM. Our system, which combines compile-time, link-time, and runtime techniques, provides the module developer with the following guarantees: (1) A kernel module will remain unmodified and it will acquire privilege only when untrusted code invokes it through developer-chosen, valid entry points with a valid stack. (2) Any execution path leaving the module will trigger a revocation of privilege. (3) The module has access to private memory. The system also provides the administrator with a secure method to bind a specific module with particular privileges implemented by the VMM. This lays the basis for guaranteeing that *only* trusted code in the guest can utilize special privileges. We give two examples of guarded Linux kernel modules: a network interface driver with direct access to the physical NIC and an idle loop that uses instructions not usually permitted in a guest, but which can be adaptively selected when no other virtual core shares the physical core. In both cases *only* the guarded module has these privileges.

1 Introduction

By design, a virtual machine monitor (VMM) does not trust the guest operating system and thus does not allow it access to privileged hardware or VMM state. However, such access can allow new or better services for the guest, such as the following examples.

- Direct guest access to I/O devices would allow existing guest drivers to be used, avoid the need for virtual devices, and accelerate access when the device could be dedicated to the guest. In existing systems, the VMM limits the damage that a rogue guest could inflict by only using self-virtualizing devices [14, 19] or by operating in contexts such as HPC environments, where the guest is trusted and often runs alone [10].
- Direct guest access to the Model-Specific Registers (MSRs) that control dynamic voltage and frequency scaling (DVFS) would allow the guest's adaptive control of these features to be used instead of the VMM's whenever possible. Because applications running on the guest enjoy access to more rich information than the VMM does, there is reason to believe that guest-based control would perform better.
- Direct guest access to instructions that can halt the processor, such as `monitor` and `mwait`, would allow more efficient idle loops and spinlocks when the VMM determines that such halts can be permitted given the current configuration.

Since we cannot trust the guest operating system, to create such services we must be able to place a component *into* the guest operating system that is both tightly coupled with the guest and yet protected from it. In prior work [5], we presented GEARS, a framework for allowing the implementation of a service to span the guest and the VMM, even without guest cooperation. GEARS provides the ability to inject modules into the guest, but the injected code runs with the same privilege and the same hardware access as other, untrusted guest code. In this paper, we extend this functionality to allow for the injected code to be endowed with privileged access to hardware and the VMM that the VMM selects, but only under specific conditions that preclude the rest of the guest from taking advantage of the privilege. We refer to this privileged injected code as a *guarded module*, and it is effectively a piece of the VMM running in the guest con-

text.

Our technique leverages compile-time and link-time processing which identifies valid entry and exit points in the module code, including function pointers. These points are in turn "wrapped" with automatically generated stub functions that communicate with the VMM. Our current implementation of this technique applies to Linux kernel modules. The unmodified source code of the module is the input to the implementation, while the output is a kernel object file that includes the original functionality of the module and the wrappers. Conceptually, a guarded module has a *border*, and the wrapper stubs (and their locations) identify the valid *border crossings* between the guarded module, which is trusted, and the rest of the kernel, which is not.

A wrapped module can then be injected into the guest using the existing GEARS framework, or added to the guest voluntarily. The wrapper stubs and other events detected by the VMM drive the second component of our technique, a state machine that executes in the VMM. An initialization phase determines whether the wrapped module has been corrupted and where it has been loaded, and then protects it from further change. Attempted border crossings, either via the wrapper functions or due to interrupt/exception injection, are caught by the VMM and validated. Privilege is granted or revoked on a per-virtual core basis. Components of the VMM that implement privilege changes are called back through a standard interface, allowing the mechanism for privilege granting/revoking to be decoupled from the mechanism for determining when privilege should change. The privilege policy is under the ultimate control of the administrator, who can determine the binding of specific guarded modules with specific privilege mechanisms.

Our contributions are as follows:

- We describe the design of the joint compile-time and run-time guarded module mechanism.
- We describe the implementation of the design for supporting guarded Linux modules in the context of the Palacios VMM [11, 9]. Our implementation is publicly available within the Palacios codebase.
- We evaluate the performance of our implementation, independent of the service and the privilege mechanism.
- We extend Palacios with a privilege mechanism, a PCI device passthrough capability that can dynamically acquire and release privilege, and then demonstrate passthrough NIC access using a guarded module that drives this mechanism. Only the module has access to the NIC.
- We extend Palacios with a second privilege mechanism, selectively-enabled access to the `monitor` and `mwait` instructions, and then demonstrate adaptive use of these instructions in a guarded module. Only the module has access to the instructions and can halt the physical core using them.

2 Related work

Process Isolation Protecting trusted applications from an untrusted OS has recently become an active area of research. Overshadow [3] first showed that hardware virtualization techniques can be used to ensure control-flow, data, and address space integrity for a process running in the guest. TrustVisor [17] extended this idea with a much smaller trusted computing base (TCB). Flicker [18] uses nascent hardware support to effectively protect trusted applications. XOMOS [13] achieves the same goal, albeit with a new ABI and an ISA that has not yet been implemented in real hardware. InkTag [6] and Virtual Ghost [4] both aim to further defend these trusted applications from a small subset of potential Iago attacks [2], a new class of attacks in which a malicious kernel crafts return values from system services to trick a trusted application into following a code path intended by the attacker. However, these systems not only lack support for trusted *kernel components*, they also leverage existing protection domains and do not consider the protection of a trusted component from attacks originating in *the same address space*.

Kernel-space Isolation A large portion of previous work on kernel-space isolation is intended for isolating an entire kernel from *untrusted*, external components. LeVasseur's work on using virtual machines as vehicles for commodity driver reuse and fault isolation [12] shows promise, but these techniques involve using driver code residing in a completely separate virtual machine.

Swift showed, with *Nooks* [22], that code wrappers can isolate faulty code in Linux kernel extensions, improving the reliability of the core kernel. While Nooks provides an illustrative example of defining boundaries between driver and kernel code, it requires modifications to the kernel in which the drivers reside. Our system requires no such modifications. Further, Nooks does not consider the situation in which a trusted module/extension requires protection *from* an untrusted kernel—our primary area of concern.

Both LXFI [16] and SecVisor [20] explore isolation in terms of guaranteeing kernel integrity. LXFI mitigates the potential for privilege escalation attacks against kernels by requiring that programmers annotate their mod-

ules. SecVisor insulates kernels from untrusted code by only allowing VMM-authorized code to execute, preventing a broad class of code-injection attacks against the kernel. Protecting the kernel against both malicious attacks and faulty software components are important problems, but they are orthogonal to our concerns. Our system guarantees the integrity of kernel *modules* that enjoy both a higher level of trust and privilege than the rest of the OS.

VM Introspection There have been several examples of leveraging the guest-host relationship to improve VM monitoring and resource management, especially in the context of autonomic computing [24, 15, 23, 7]. However, as far as we are aware, the only existing use case for trusted, isolated components within a guest kernel is for security monitors in which the only protected state is the code and data of the monitor itself, not higher-privilege state such as that required to access the hardware such as we outlined in Section 1.

IntroVirt [8] allows a VMM to invoke code in the guest, but does not deal with enforcing separate levels of trust within the *same* guest.

SYRINGE [1] provides a mechanism by which secure monitoring code can leverage functions in an untrusted guest. This system employs a secure VM along with an untrusted VM. When the monitoring code in the secure VM needs to call a function in the untrusted VM, the hypervisor forwards the call, managing control-flow and data integrity such that the secure VM is not compromised. However, this system is more akin to a secure, cross-core RPC facility that does not address border crossings within the same address space—a major component of our work.

Secure in-VM monitoring, or SIM [21], addresses performance issues raised by previous VM introspection techniques by allowing monitoring code to run directly in the guest while ensuring the monitor's integrity. While SIM touches on the border crossings that are our focus, it largely sidesteps the issue by using a completely separate address space for the trusted monitor code. We do not have this option as we seek to guard modules that reside in the same address space as the untrusted kernel.

As far as we are aware, the guarded module system we present is the first of its kind that guarantees both control-flow and data integrity for modules that share the same address space as an untrusted OS kernel. Guarded modules require no specialized hardware and no modifications to the guest OS in which they execute.

3 Trust and threat models; invariants

We assume a completely untrusted guest kernel. A developer will add to the VMM selective privilege mechanisms that are endowed with the same level of trust as the rest of the core VMM codebase. A module developer will assume that the relevant mechanism exists. The determination of whether a particular module is allowed access to a particular selective privilege mechanism is made at run-time by an administrator. The central relationship we are concerned with is between the untrusted guest kernel and the module. A compilation process transforms the module into a guarded module. This then interacts with run-time components to maintain specific invariants in the face of threats from the guest kernel.

Control-flow integrity The key invariant we provide is that the privilege on a given virtual core will be enabled if and only if that virtual core is executing within the code of the guarded module and the guarded module was entered via one of a set of specific, agreed-upon entry points. The privilege will be disabled whenever control flow leaves the module, including for interrupts and exceptions.

The guarded module boasts the ability to interact freely with the rest of the guest kernel. In particular, it can call other functions and access other data within the guest. A given call stack might intertwine guarded module and kernel functions, but the system guards against attacks on the stack as part of maintaining the invariant.

A valid entry into the guarded module is not checked further. Our system does not guard against an attack based on function arguments or return values, namely Iago attacks. The module author needs to validate these himself. Note, however, that the potential damage of performing this validation incorrectly is limited to the specific privilege the module has.

Code integrity Disguising the module's code is not a goal of our system. The guest kernel can read and even write the code of the guarded module. However, any modifications of the code by any virtual core will be caught and the privilege will be disabled for the remainder of the module's lifetime in the kernel. The identity of the module is determined by its content, and module insertion is initiated external to the guest with a second identifying factor, guarding against the kernel attempting to spoof or replay a module insertion.

Data integrity Data integrity, beyond the registers and the stack, is managed explicitly by the module. The mod-

ule can request private memory as a privilege. On a valid entry, the memory is mapped and usable by the module, while on departing the module, the memory is unmapped and is thus invisible and inaccessible to the rest of the kernel.

4 Design and implementation

The specific implementation of guarded modules we describe in this paper applies to Linux kernel modules. Our implementation fits within the context of the Palacios VMM and takes advantage of code generation and linking features of the GCC and GNU binutils toolchains. The VMM-based elements leverage functionality commonplace in modern VMMs, and thus could be readily ported to other VMMs. The code generation and linking aspects of our implementation seem to us to be feasible in any C toolchain that supports ELF or a similar format. The technique could be applicable to other guest kernels, although we do assume that the guest kernel provides runtime extensibility via some form of load-time linking.

In our implementation, a guarded Linux kernel module can either be voluntarily inserted by the guest or involuntarily injected into the guest kernel using the GEARS framework. The developer of the module needs to target the specific kernel he wants to deploy on, exactly as in creating a Linux kernel module in general.

The guarded module is a kernel module within the guest Linux kernel that is allowed privileged access to the physical hardware or to the VMM itself. The nature of this privilege, which we will describe later, depends on the specifics of the module. We refer to the code boundary between the guarded module and the rest of the guest kernel as the *border*.

Border crossings consist of control flow paths that traverse the border. A *border-out* is a traversal from the module to the rest of the kernel, of which there are three kinds. The first, a *border-out call* occurs when a kernel function is called by the guarded module, while the second, a *border-out ret*, occurs when we return back to the rest of the kernel. The third, a *border-out interrupt* occurs when an interrupt or exception is dispatched. A *border-in* is a traversal from the rest of the kernel to the guarded module. There are similarly three forms here. The first, a *border-in call* consists of a function call from the kernel to a function within the guarded module, while the second, a *border-in ret* consists of a return from a *border-out call*, and the third, a *border-in rti* consists of a return from a border-out interrupt. Valid border-ins should raise privilege, while border-outs should lower privilege. Additionally, any attempt to modify the module should lower privilege.

The VMM contains a new component, the *border control state machine*, that determines whether the guest has privileged access at any point in time. The state machine also implements a registration process in which the injected guarded module identifies itself to the VMM and is matched against validation information and desired privileges. This allows the administrator to decide which modules, by content, are allowed which privileges. After registration, the border control state machine is driven by hypercalls from the guarded module, exceptions that occur during the execution of the module, and by interrupt or exception injections that the VMM is about to perform on the guest.

The VMM detects attempted border crossings jointly through its interrupt/exception mechanisms and through hypercalls in special code added to the guarded module as part of our compilation process. Figure 1 illustrates how the two interact.

4.1 Compile-time

Our compilation process, *Christoization*[1], automatically wraps an existing kernel module with new code needed to work with the rest of the system. Two kinds of wrappers are generated. *Exit wrappers* are functions that interpose on the calls from the guarded module to the rest of the kernel. An exit wrapper, added using link-time processing, signals the VMM by a hypercall to lower privilege just before the underlying function call is made. When the function returns, it signals the VMM to validate the stack and raise privilege. *Entry wrappers* are functions that interpose on calls from the kernel into the guarded module. Entry wrappers, which are introduced by source preprocessing, use hypercalls to signal the VMM to raise privilege when called, and then lower privilege when the call returns to the kernel. The precise positions of the hypercall instructions in the wrappers are used by the VMM to validate the requests.

We designed our compile-time tool chain so that module developer effort is minimized when generating a guarded module. The requisite knowledge and materials are the same as what would be required of a developer writing a Linux kernel module. The necessary inputs to our toolchain are the guest Linux Makefile and kernel headers, as well as the source and Makefile for the module to be Christoized. Additionally, the privilege names required by the module are passed as command-line parameters. Access to the guest Linux source tree may also

[1] Named after the famed conceptual artist, Christo, who was known for wrapping large objects such as buildings and islands in fabric.

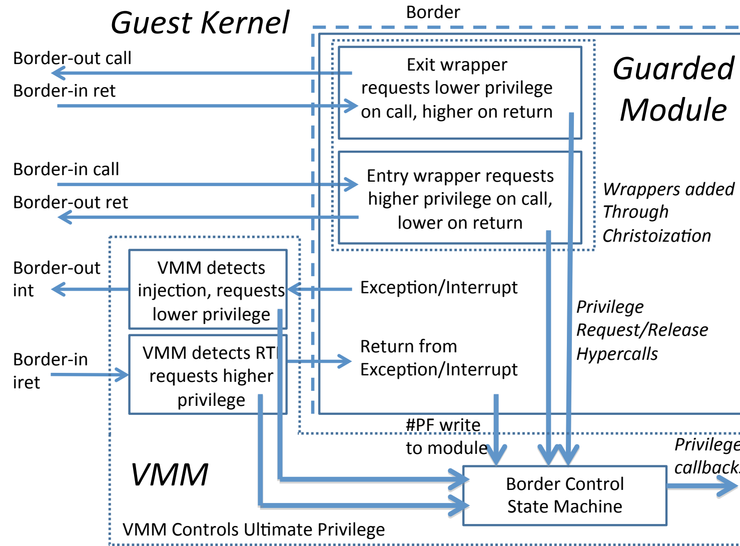

Figure 1: Guarded modules, showing operation of wrappers and interaction of state machine on border crossings.

be required if the developer wishes to use external functions that use non-standard calling conventions.

The first stage of the Christoization process is module source analysis. We scan the source files of the module, looking for functions that are assigned as callbacks. These functions represent entry points into the module, as the kernel will invoke them asynchronously. In order to effectively identify all of these functions, we must run a preprocessing pass over the module to make sure that external inlined functions and macros are accounted for. Once the entry callbacks are identified, we must search the source for the function that the module developer registers using Linux's `module_init` macro. This function will serve as the initial gateway into the module and must be intercepted by the VMM.

In the source annotation stage, each entry callback assignment in the source is changed to a macro that will expand to an entry wrapper function particular to that callback. These wrappers are added to the source file automatically and are depicted in Figure 2. The key idea here is that a hypercall is inserted both before and after the call to the original entry point. The remaining instructions are there to preserve the environment in such a way that the original function is not aware that it has been wrapped. The `module_init` routine is then similarly wrapped with a registration hypercall that notifies the VMM when it has been inserted into the guest kernel.

The linker wrapping stage takes the output of the annotation stage (a compiled object) and identifies undefined function references. These represent exits to the kernel. They are wrapped with exit wrappers, which are assembly stubs similar to entry wrappers. Exit wrappers lower privilege before the original call and raise it on return. They are added using `ld`'s function wrapping capability. The result of this linking step is that the module's original unresolved external references are resolved to the exit wrappers, while the exit wrappers contain references to the original unresolved symbols. As a result, any external call from the original module goes through an exit wrapper.

The final stage of the Christoization process is metadata generation. Here, information collected in the previous stages is aggregated into a formatted file with which the administrator can later register the guarded module. The essential metadata consists of the module's name, its required privileges, and the offsets in the compiled object of the identified valid entry points. This list can later be further restricted or expanded by the module developer. Additionally, to ensure module integrity at load-time, a cryptographic content hash of the code segment is performed and recorded. This metadata is later passed by the administrator to the VMM during the guarded module registration process, and it is used from then on by the border control state machine to validate the hypercalls and other events it receives.

```
entry_wrapped:
      popq  %r11
      pushq %rax
      movq  $border_in_call, %rax
(a)   vmmcall
      popq  %rax
      callq entry
      pushq %rax
      movq  $border_out_ret, %rax
(b)   vmmcall
      popq  %rax
      pushq %r11
      ret   (to rest of kernel)
```

Figure 2: An entry wrapper for a valid entry point. Exit wrappers are similar, except they invoke border out on on a call, and border in after returning.

4.2 Run-time

The run-time element of our system is based around the border control state machine. As Figure 1 illustrates, the state machine is driven by hypercalls originating from the guarded module, and by events that are raised elsewhere in the VMM. As a side-effect of the state machine's execution, it generates callbacks to other components of the VMM (*selective privilege-enabled VMM components*) notifying them when valid privilege changes occur. The state machine also handles the initialization of a guarded module and its binding with these other parts of the VMM. We now describe guarded module execution with respect to the state machine.

Module initialization The guarded module is injected into the guest, either voluntarily by the user, or involuntarily by the administrator using GEARS's code injection facility. The module's initialization code immediately calls the guarded module registration function that was generated by Christoization. This function makes an initialization hypercall, providing a claimed hash as its argument. In response, the state machine validates the module using the metadata associated with the claimed hash. First, the address of the initialization hypercall instruction, combined with the known offset of the instruction in the text segment stored in the metadata, allows us to determine the load address of the module's text segment. The metadata includes the length of the text section. With this information, the state machine then marks the text segment as unwritable in the shadow or nested page tables, making it impossible for the guest to change it. The next step is to compute the hash over the text segment memory and compare it to the hash stored in the metadata.[2] If the hashes match, the state machine notifies the selective privilege-enabled component that privilege should be raised, transitions to the privileged state, enables interception of exceptions, and returns to the guest. At this point, the guarded module can complete the remainder of its initialization. In effect, module initialization is treated as the first border-in call.

Border-in call to border-out ret A valid entry into the guarded module results in a hypercall from the entry wrapper (Figure 2(a)) that requests a privilege raise. The address of this hypercall instruction is then validated against the list of addresses where such instructions were placed, which is stored in the metadata. If it is in the list, the state machine invokes a privilege-raising callback, and transitions to the privileged state. Before returning, it also enables interception of exceptions. Before exiting from a valid entry, the entry wrapper similarly invokes another hypercall (Figure 2(b)), which requests a lowering of privilege. When privilege is lowered, exception interception is returned to its nominal state.

Border-out call to border-in ret A call from the guarded module to the rest of the kernel results in a hypercall from the exit wrapper that requests a lowering of privilege. As a side-effect of lowering privilege, exception interception is returned to its nominal state. When the call returns, a second hypercall requests a raising of privilege. After sanity checking the address against the metadata, privilege is raised, and exception and interrupt interception are again enabled.

Border-out int to border-in rti The purpose of intercepting exceptions that occur when executing with privilege is to assure that we can lower privilege when these events trigger an interrupt handler dispatch and raise it once execution resumes in the guarded module. More generally, we must trap *any* switch from the guarded module code to kernel context. When the guest is not executing in the guarded module, nominal exception handling is sufficient. Our handler for exception intercepts simply causes the VMM to re-inject the exceptions alongside its normal injection of interrupt events.

Because we need to be aware of every interrupt/exception *dispatch*, we have modified the Palacios VM entry code so that, just before such an entry, if the guest is executing with privilege, we determine if an interrupt or exception injection will occur on the entry. If so, we lower privilege, switch back to nominal interception of

[2] A direct comparison of the text segment content is also possible.

exceptions, and enable interception of the `rti` instruction, which will be executed when the interrupt or exception handler completes. We also note the current `%rip` and other information related to this interrupt dispatch.

At this point, we allow the VM entry to complete, and interrupt dispatch ensues. We emulate `rti` instructions when they occur, looking for any `rti` that will return control to the instruction at which the original interrupt/exception was injected. When we discover a match, we raise privilege, re-enable exception interception, disable `rti` interception, and resume execution with privilege in the guarded module.

We note that one privilege that could be granted to a module is the ability to disable interrupts while it executes. If this is the case, this code path could be entirely avoided.

Internal calls The entry wrapper shown in Figure 2 and the exit wrappers are linked such that they are only invoked on border crossings. Calls internal to the guarded module do not have any additional overhead. The same applies for calls internal to the kernel.

Nesting and stack checking Although it is convenient to think of (and generate code for) border-crossings in matched pairs, it is important to realize that an execution path may involve multiple border-crossings. For example, the kernel might invoke a callback function on the module, which requires privilege, but which in turn calls a kernel function, which *should not* have privilege, and that subsequently makes another callback into the module, which *should*. The sequence of events for that example would be: border-in call, border-out call(*), border-in call, border-out ret, border-in ret(**), border-out ret. While border-ins and border-outs must eventually all be matched, they can nest. This nesting of border crossings introduces an opportunity to subvert the guarded module through the stack. Our primary concern is the protection of the `ret` in the border-out wrapper. If the border-out call(*) had its return address modified on the stack, the border-in ret(**) would return to that address with privilege raised!

To address this, the border control state machine tracks the nesting level and the stack state, and validates the stack state on any border-in. When a border-in occurs with a nesting level of zero, the state machine captures the starting point of this "first border-in" stack frame (i.e., `%rsp` and `%rbp`). When a border-out occurs, the state machine captures the ending point of this "last border-out" stack frame, and computes and stores a hash of the stack content from the first entry to this last exit.

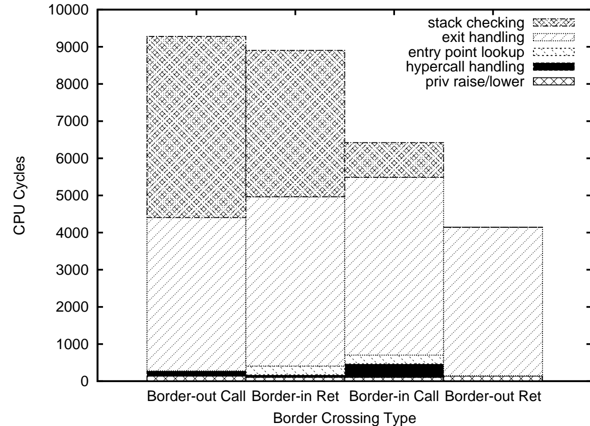

Figure 3: Privilege change cost with stack integrity checks.

On any border-in whose nesting level is greater than zero, the actual stack is again hashed and compared with the last border-out hash. If they do not match, privilege is not granted.

Deinitialization The Christoization processing inserts a deinitialization hypercall as the last thing the module executes. After validating the hypercall's location, the state machine lowers privilege, removes any special interception that is active, and remaps the module with guest-specified writability. Privilege will not change again unless the initialization hypercall is executed.

Suspicious activity The state machine detects suspicious activity by noting privilege changing hypercalls at invalid locations, shadow or nested page faults indicating attempts to write the module code, and stack hash mismatches. Our default behavior is simply to lower privilege when these occur, and continue execution. Other reactions are, of course, possible.

5 Evaluation

We now consider the costs of the guarded module system, independent of any specific guarded module that might drive it, and any selective privilege-enabled VMM component it might drive. We focus on the costs of border crossings and their breakdown. The most important contributors to the costs are VM exit/entry handling and the stack validation mechanism.

All measurements were conducted on a Dell PowerEdge R415. This is a dual-socket machine, each socket comprising a quad-core, 2.2 GHz AMD Opteron 4122, giving a total of 8 physical cores. The machine has

16 GB of memory. It runs Fedora 15 with a stock Fedora 2.6.38 kernel. Our guest environment uses a single virtual core that runs a BusyBox environment based on Linux kernel 2.6.38. The guest runs with nested paging using 2 MB page mappings and DVFS control is disabled.

Figure 3 illustrates the overheads in cycles incurred at runtime. All cycle counts were averaged over 1000 samples. There are five major components to the overhead. The first is the cost of initiating a callback to lower or raise privilege. This cost is very small at around 100 cycles. The second cost, labeled "hypercall handling", denotes the cycles spent inside the hypercall handler itself, not including entry validations, privilege changes, or other processing involved with a VM exit. This cost is also quite small, and also typically under 100 cycles. "entry point lookup" represents the cost of a hash table lookup, which is invoked on border-ins when the instruction pointer is checked against the valid entry points that have been registered during guarded module initialization. The cost for this lookup is roughly 240 cycles. "exit handling" is the time spent in the VMM handling the exit outside of guarded module runtime processing. Finally, "stack checking" denotes the time spent ensuring control-flow integrity by validating the stack. This component raises the cost of a border crossing by 5000 cycles, mostly due to stack address translations and hash computations. Border-in calls are less affected due to the initial translation and recording of the entry stack pointer, while border-out rets are unaffected. Reducing the cost of this validation is the subject of on-going work.

The guarded module codebase consists of the compile-time tools, which comprise 223 lines of Perl, 260 lines of Ruby and the run-time elements added to the VMM. The latter are generally concentrated in an optional extension of 1007 lines of C that could be ported to other VMMs. Some changes to the VMM core were made to facilitate interrupt and exception interception and dispatch to the GEARS guarded module system. These changes include 178 lines of C.

6 Examples

We now consider two examples of using the guarded module functionality, drawn from the list in the introduction. In the first example, selectively-privileged PCI passthrough, the guarded module, and only the guarded module, is given direct access to a specific PCI device. We illustrate the use of this capability via a guarded version of a NIC driver. In our second example, selectively-privileged `mwait`, the guarded module, and only the guarded module, is allowed to use the `mwait` instruction. We illustrate the use of this capability via guarded module that adaptively replaces the kernel idle loop with a more efficient `mwait` loop when it is safe to do so.

We conducted all measurements in this section with the configuration described in Section 5.

6.1 Selectively privileged PCI passthrough

Like most VMMs, Palacios has hardware passthrough capabilities. Here, we use its ability to make a hardware PCI device directly accessible to the guest. This consists of a generic PCI front-end virtual device ("host PCI device") , an interface it can use to acquire and release the underlying hardware PCI device on a given host OS ("host PCI interface"), and an implementation of that interface for a Linux host.

A Palacios guest's physical address space is contiguously allocated in the host physical address space. Because PCI device DMA operations use host physical addresses, and because the guest programs the DMA engine using guest physical addresses it believes start at zero, the DMA addresses the device will actually use must be offset appropriately. In the Linux implementation of our host PCI interface, this is accomplished using an IOMMU: acquiring the device creates an IOMMU page table that introduces the offset. As a consequence, any DMA transfer initiated on the device by the guest will be constrained to that guest's memory. A DMA can then only be initiated by programming the device, which is restricted to the guarded module. This restriction also prevents DMA attacks on the module that might originate from the guest kernel.

A PCI device is programmed via control/status registers that are mapped into the physical memory and I/O port address spaces through standardized registers called BARs. Each BAR contains a type, a base address, and a size. Palacios's host PCI device virtualizes the BARs (and other parts of the standardized PCI device configuration space). This lets the guest map the device as it pleases. For a group of registers mapped by a BAR into the physical memory address space, the mapping is implemented using the shadow or nested page tables to redirect memory reads and writes. For a group of registers mapped into the I/O port space, there is no equivalent to these page tables, and thus the mappings are implemented by I/O port read/write hooks. When the guest executes an IN or OUT instruction, an exit occurs, the hook is run, and the handler simply executes an IN or OUT to the corresponding physical I/O port. If the host and guest mappings are identical, the ports are not intercepted, allowing the guest to read/write them directly.

We extended our host PCI device to support selective privilege; in the terminology of Section 4.2, it is now a selective privilege-enabled VMM component. In this mode of operation, virtualization of the generic PCI configuration space of the device proceeds as normal. However, at startup, BAR virtualization ensures that the address space regions of memory and I/O BARs are initially hooked to stub handlers. The stub handlers simply ignore writes and supply zeros for reads. This is the *unprivileged mode*. In this mode, the guest sees the device on its PCI bus, and can even remap its BARs as desired, but any attempt to program it will simply fail because the registers are inaccessible. In selectively privileged operation, the host PCI device also responds to callbacks for raising and lowering privilege. Raising privilege switches the device to *privileged mode*, which is implemented by remapping the registers in the manner described earlier, resulting in successful accesses to the registers. Lowering privilege switches back to unprivileged mode, and remaps the registers back to the stubs. Raising and lowering privilege happens on a per-core basis.

While the above description is complex, it is important to note that only about 60 lines of code were needed to add selectively privileged operation to our existing PCI passthrough functionality. Combined with the rest of the guarded module system, the selectively privileged host PCI device lets us permit fully privileged access to the underlying device within a guarded module, but disallow it otherwise.

Making a NIC driver into a guarded module As an example, we used the guarded module system to generate a guarded version of an existing NIC device driver within the Linux tree, specifically the Broadcom BCM5716 Gigabit NIC. No source code modifications were done to the driver or the guest kernel. We Christoize this driver, creating a kernel module that we can later inject into the untrusted guest. The border control state machine in Palacios pairs this driver with the selectively privileged PCI passthrough capability. Recall that Christoization is almost entirely automated, so the result is an unmodified device driver, executing in the guest, having direct access to the NIC, while nothing else in the guest does.

The NIC uses exactly one BAR to define a 32 MB region of the memory address space. Raising and lowering privilege amounts to editing the shadow or nested page tables to remap these addresses. Assuming 2 MB superpages and suitable alignment, the system will adjust 16 page table entries when changing privilege.

Packet Sends	
Border-in	1.06
Border-out	1.06
Border Crossings / Packet Send	**2.12**
Packet Receives	
Border-in	4.64
Border-out	4.64
Border Crossings / Packet Receive	**9.28**

Figure 4: Border crossings per packet send and receive for the NIC example.

Overheads Compared to simply allowing privilege for the entire guest, a system that leverages guarded modules incurs additional overheads. Some of these overheads are system-independent, and were covered in Section 5. The most consequential component of these overheads is the cost of executing a border-in or border-out, each of which consists of a hypercall or exception interception (requiring a VM exit) or interrupt/exception injection detection (done in the context of an in-progress VM exit), a lookup of the hypercall's address, a stack check or record, conducting a lookup to find the relevant privilege callback function, and then the cost of invoking that callback.

We now consider the system-dependent overhead for the NIC. There are two elements to this overhead: the cost of changing privilege and the number of times we need to change privilege for each unit of work (packet sent or received) that the module finishes. The cost of raising privilege for the NIC is 4800 cycles (2.2 μs), while lowering it is 4307 cycles (2.0 μs).

Combining the system-independent and system-dependent costs, we expect that a typical border crossing overhead, assuming no stack checking will consist of about 3000 cycles for VM exit/entry, 4000 cycles to execute the border control state machine, and about 4500 cycles to enable/disable access to the NIC. These 11500 cycles comprise 5.2 μs on this machine. Stack checking would add an average of about 4500 cycles, leading to 16000 cycles (7.3 μs).

To determine the number of these border crossings per packet send or receive, we counted them while running the guarded module with a controlled traffic source (ttcp) that allows us to also count packet sends and/or receives. Dividing the counts gives us the average. There is variance because the NIC does interrupt coalescing.

Figure 4 shows the results of this analysis for the NIC. Sending requires on the order of 2 border crossings (privilege changes) per packet, while receiving requires on the order of 9 border crossings per packet. Note that many of the functions that constitute border crossings are actually leaf functions defined in the kernel. This indicates

that we could further reduce the overall number of border crossings per packet by pulling the implementations of these functions into the module itself.

6.2 Selectively privileged mwait

Recent x86 machines include a pair of instructions, `monitor` and `mwait`, that can be used for efficient synchronization among processor cores. The `monitor` instruction indicates an address range that should be watched. A subsequent `mwait` instruction then places the core into a suspended sleep state, similar to a `hlt`. The core resumes executing when an interrupt is delivered to it (like a `hlt`), or when another core writes into the watched address range (unlike a `hlt`). The latter allows a remote core to wake up the local core without the cost of an inter-processor interrupt (IPI). One example of such use is in the Linux kernel's idle loop.

In Palacios, and other VMMs, we cannot allow an untrusted guest to execute `hlt` or `mwait` because the guest runs with physical interrupts disabled. A physical interrupt is intended to cause a VM exit followed by subsequent dispatch of the interrupt in the VMM. If an `mwait` instruction were executed in the guest under uncontrolled conditions, it could halt the core indefinitely. This precludes the guest using the extremely fast intercore wakeup capability that `mwait` offers.

Under controlled conditions, however, letting the guest run `mwait` may be permissible. When no other virtual core is mapped to the physical core (so we can tolerate a long wait) and we have a watchdog that will eventually write the memory, the guest might safely run an `mwait`. To achieve these controlled conditions requires that we limit the execution of these instructions to code that the VMM can trust and that this code only execute `mwait` when the VMM deems it safe to do so. A malicious guest could use an unrestricted ability to execute `mwait` to launch a denial-of-service attack on other VMs and the VMM. We enforce this protection and adaptive execution by encapsulating the `mwait` functionality within the safety of a guarded module.

Adding selectively-privileged access to `mwait` to Palacios was straightforward, involving only a few lines of code. We then implemented a tiny kernel module that interposes on Linux's default idle loop, specifically modifying `pm_idle`, a pointer to the function that points to the idle implementation. Our module points this to a function internal to itself that dispatches either to an `mwait`-based idle implementation within the module or to the original idle implementation, based on a flag in protected memory that is shared with Palacios. Palacios sets this flag when it is safe for the module to use `mwait`. In these situations, the guest kernel enjoys much faster wake-ups of the idling core.

To assure that only our module can execute `mwait` we transform it into a guarded module using the techniques outlined earlier in the paper. A border-in to our module occurs when Linux calls its idle loop. If the border-in succeeds, Palacios stops intercepting the use of `mwait`. When control leaves the module, a border-out occurs, and Palacios resumes intercepting `mwait`. If code elsewhere in the guest attempts to execute these instructions, they will trap to the VMM and result in an undefined opcode exception being injected into the guest.

This proof-of-concept illustrates how the VMM can use guarded modules to safely adapt the execution environment of a VM to changing conditions.

7 Conclusions and future work

We presented the design, implementation, and evaluation of a system for guarded modules. The system allows the VMM to add modules to a guest kernel that have higher privileged access to physical hardware and the VMM while protecting these guarded modules and access to their privileges from the remainder of the guest kernel. Our system is based on joint compile-time and run-time techniques that bestow privilege only when control flow enters the guarded module at verified locations. We demonstrated two example uses of the guarded module system. The first is passthrough access to a PCI device, for example a NIC, that is limited to a designated guarded module (a device driver). The guest kernel can use this guarded module as it would any other device driver. We further demonstrated selectively privileged use of the `monitor` and `mwait` instructions in the guest, instructions that could wreak havoc if their use was not constrained to a guarded module that cooperates with the VMM.

Our ongoing and future work lies along two lines. First, we will explore methods that can further enhance the performance of this system. Building upon the analysis of Section 6, we plan to further study methods by which we can reduce the cost and number of border crossings needed for a specific module. As previously mentioned, we are investigating an expansive linking process in which kernel functions invoked by the guarded module are incrementally incorporated into the module itself. Our second line of investigation is in designing other virtualization services that could be simplified or enabled by employing guarded modules.

References

[1] CARBONE, M., CONOVER, M., MONTAGUE, B., AND LEE, W. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID 2012)* (September 2012).

[2] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)* (March 2013).

[3] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)* (March 2008).

[4] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)* (March 2014).

[5] HALE, K., XIA, L., AND DINDA, P. Shifting GEARS to enable guest-context virtual services. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC 2012)* (September 2012).

[6] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)* (March 2013).

[7] HU, L., SCHWAN, K., GULATI, A., ZHANG, J., AND WANG, C. Net-cohort: Detecting and managing vm ensembles in virtualized data centers. In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2012)* (September 2012).

[8] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP 2005)* (October 2005).

[9] LANGE, J., DINDA, P., HALE, K., AND XIA, L. An introduction to the palacios virtual machine monitor—release 1.3. Tech. Rep. NWU-EECS-11-10, Department of Electrical Engineering and Computer Science, Northwestern University, October 2011.

[10] LANGE, J., PEDRETTI, K., DINDA, P., BRIDGES, P., BAE, C., SOLTERO, P., AND MERRITT, A. Minimal overhead virtualization of a large scale supercomputer. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011)* (March 2011).

[11] LANGE, J., PEDRETTI, K., HUDSON, T., DINDA, P., CUI, Z., XIA, L., BRIDGES, P., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)* (April 2010).

[12] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)* (December 2004).

[13] LIE, D., THEKKATH, C. A., AND HOROWITZ, M. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)* (October 2003).

[14] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 2006)* (May 2006).

[15] LU, L., ZHANG, H., JIANG, G., CHEN, H., YOSHIHIRA, K., AND SMIRNI, E. Untangling mixed information to calibrate resource utilization in virtual machines. In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011)* (June 2011).

[16] MAO, Y., CHEN, H., ZHOU, D., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP 2011)* (October 2011).

[17] MCCUNE, J. M., LI, Y., NING, Q., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP 2010)* (May 2010).

[18] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM European Conference in Computer Systems (EuroSys 2008)* (April 2008).

[19] RAJ, H., AND SCHWAN, K. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC 2007)* (July 2007).

[20] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)* (October 2007).

[21] SHARIF, M. I., LEE, W., CUI, W., AND LANZI, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)* (November 2009).

[22] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)* (October 2003).

[23] WANG, L., XU, J., AND ZHAO, M. Application-aware cross-layer virtual machine resource management. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC 2012)* (September 2012).

[24] XU, J., ZHAO, M., FORTES, J., CARPENTER, R., AND YOUSIF, M. On the use of fuzzy modeling in virtualized data center management. In *Proceedings of the 4th International Conference on Autonomic Computing (ICAC 2007)* (June 2007).