NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NU-EECS-13-04

April 16, 2013

Guarded Execution of Privileged Code in the Guest

Kyle C. Hale and Peter A. Dinda

Abstract

Allowing a guest to have direct, privileged access to hardware can enhance its performance and functionality. Privileged access to hardware and the VMM also enables and improves the performance of virtualization services by allowing portions of their implementations to be hoisted into the guest, even uncooperatively. However, granting such privilege currently requires that the *entire* guest be trusted. We present a software technique, guarded execution of privileged code, that allows the VMM to inject code modules into the guest that enjoy unrestrained access to specific hardware and VMM resources. Our system, which combines compile-time, link-time, and run-time techniques, provides the module developer with the guarantee that the module remains unmodified, and that it acquires privilege only when untrusted code invokes it through developer-chosen, valid entry points with a valid stack. An execution path leaving the module will then trigger a revocation of privilege. The system also provides the administrator with a secure method for binding 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 a motivating example by guarding the execution of a privileged, network interface driver in the form of a Linux module, such that it, and only it, has uninhibited access to the NIC hardware.

This project is made possible by support from the United States National Science Foundation (NSF) via grant CNS-0709168, and the Department of Energy (DOE) via grant DE-SC0005343.

Keywords: guarded modules, virtual machines, services, code transformation

Guarded Execution of Privileged Code in the Guest

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

Abstract

Allowing a guest to have direct, privileged access to hardware can enhance its performance and functionality. Privileged access to hardware and the VMM also enables and improves the performance of virtualization services by allowing portions of their implementations to be hoisted into the guest, even uncooperatively. However, granting such privilege currently requires that the *entire* guest be trusted. We present a software technique, guarded execution of privileged code, that allows the VMM to inject code modules into the guest that enjoy unrestrained access to specific hardware and VMM resources. Our system, which combines compile-time, link-time, and run-time techniques, provides the module developer with the guarantee that the module remains unmodified, and that it acquires privilege only when untrusted code invokes it through developer-chosen, valid entry points with a valid stack. An execution path leaving the module will then trigger a revocation of privilege. The system also provides the administrator with a secure method for binding 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 a motivating example by guarding the execution of a privileged, network interface driver in the form of a Linux module, such that it, and only it, has uninhibited access to the NIC hardware.

1 Introduction

A virtual machine monitor (VMM) does not trust the guest operating system or application running on top of it to execute privileged code or directly access devices. This is by design, and exceptions are rare. Modifications of privileged physical state such as the interrupt vector table pointer would allow a guest to subvert the VMM. However, it is also the case that allowing the guest access to privileged host or VMM state can simplify and/or speed up different services that the VMM may provide. The most obvious example is passthrough access to I/O devices, which allow existing guest drivers to be used and may permit very high performance. Here, the VMM needs to limit the damage that a rogue guest could inflict, for example by using hardware features that allow the self-virtualization of the device [6, 7], or by operating in contexts such as HPC environments, where the guest is trusted and often runs alone [4].

In recent work [2] we argued that allowing the implementation of virtualization services to extend into the guest can often simplify their design, improve their performance, or enable otherwise unfeasible services. We presented GEARS, a framework for allowing the implementation of a virtualization service to span the guest and the VMM, even without guest cooperation. The element of GEARS we focus on in this paper is code injection: GEARS allows a service developer to inject and execute code within the guest OS and application without its cooperation. For example, it can force the injection of a Linux kernel module. Currently, any injected code runs with the same privilege and the same hardware access as other guest code.

In this paper, we extend this functionality to allow for the injection of code into the guest kernel that runs at whatever privilege and with whatever hardware access the VMM selects. We refer to this injected code as a *guarded module*. When a guest thread of execution (including an interrupt) enters the guarded module at a valid entry point, the VMM will raise the privilege as appropriate. When execution leaves the guarded module, the VMM will lower the privilege to ordinary levels. This functionality allows a *part* of the guest to execute with privilege, specifically the part the VMM has supplied.

This guarded module functionality can then be used to implement virtualization services or for other purposes.

Our technique leverages compile-time and link-time processing which identifies valid entry and exit points in the module code, including via function pointers. These points are in turn "wrapped" with automatically generated stub functions that communicate with the VMM. The implementation of our 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 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 of privilege granting/revoking to be decoupled from the mechanism of determining when privilege should change. Ultimately, 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. This implementation will be made publicly available within the Palacios codebase when this paper is published.

- We evaluate the performance of our implementation, independent of virtualization service and privilege mechanism.

- We extend Palacios with a privilege mechanism, a PCI device passthrough capability that can dynamically acquire and release privilege.

- We study the performance of passthrough NIC access using guarded modules and the selective passthrough mechanism.

2 Related work

Swift et al. showed, with *Nooks* [9], that code wrappers could be employed to isolate faulty code in Linux kernel extensions, improving the reliability of the core kernel. While the overall goals of Nooks are quite different (isolation, recovery, and compatibility), the system provides an illustrative example of defining and protecting the boundaries between driver and kernel code.

Secure in-VM monitoring, or SIM [8], also executes VMM code in guest context, but fits within the theme of introspection and security rather than providing virtualization services. SIM seeks to allow the VMM efficient and secure access to guest internals, but not to equip the guest with privileged access to hardware or VMM-resident resources. The entry and exit gates used in SIM bear many similarities to our generated wrappers, so it is instructive to point out their differences. While these gates exist solely to protect a well-known piece of sensitive monitoring code, our wrapping toolchain provides a general purpose mechanism for protecting the system during states of privilege that have been dynamically provisioned. The system maintains no notion of the particular implementation of the modules to which the privileged functionality is to be provisioned. Further, SIM is designed for a virtualization-based security monitor built by the hypervisor developer. Our system enables a kernel module developer with no knowledge of VMM internals to securely and automatically extend his or her module with privilege in a virtualized environment.

SYRINGE [1] is another in-VM monitoring system, but it provides a mechanism by which the monitoring code can leverage functions in the guest. It does so using a code injection facility that shares some similarities with our GEARS framework, but which is more comparable to a cross-core remote procedure call. Secure code can invoke a guest-resident utility function by signaling the VMM to inject a function call, whose results are then passed to the secure code. However, here the monitor code resides in a secure VM, and the function call is injected into an entirely separate guest VM, so border crossings between secure and insecure code—a major component of our work—do not arise.

3 Context of work

Our work lies within the context of the Palacios VMM and the GEARS framework, which we now discuss in further detail. Although we leverage functionality in this software, it is important to note that the required functionality is readily available in other VMMs.

3.1 Palacios VMM

Our system is implemented in the context of our Palacios VMM. Palacios is an OS-independent, open source, BSD-licensed, publicly available, embeddable VMM designed as part of the V3VEE project (`http://v3vee.org`). The V3VEE project is a collaborative community resource development project involving Northwestern University, the University of New Mexico, Sandia National Labs, and Oak Ridge National Lab. Detailed information about Palacios can be found elsewhere [5, 3]. Palacios is capable of virtualizing large scale systems (4096+ nodes) with $< 5\%$ overheads [4]. Palacios's OS-agnostic design allows it to embed into a wide range of OS architectures.

The Palacios implementation is built on the virtualization extensions deployed in current generation x86/x64 processors—specifically, AMD's SVM and Intel's VT. Palacios supports both 32 and 64 bit host and guest environments, both shadow and nested paging models, and a significant set of devices that constitute the PC platform. Due to the ubiquity of the x86/x64 architecture, Palacios is capable of operating across many classes of machines. Palacios has successfully virtualized commodity desktops and servers, high-end Infiniband clusters, and Cray XT and XK supercomputers.

3.2 GEARS framework

The Guest Examination and Revision Services (GEARS) framework, implemented within Palacios, enables *guest-context virtual services*, VMM services whose implementations reside partly, or even entirely, within the guest itself. Guest-context virtual services have several advantages over those implemented within the core of a hypervisor. They are not restrained by the limited software interface that the VMM sees, and can implement functionality that would simply not be possible within the VMM itself.

We showed that GEARS, a minimal set of extensions to the hypervisor, can enable a broad range of guest-context services. The central components that comprise GEARS include system call interception, process environment modification, and code injection. In this work we focus on the code injection facilities of GEARS, but more details can be found in previous work [2].

GEARS can inject code into the guest kernel or a user-space process, and the injected code can execute at a time chosen by the hypervisor. Code injected into user-space can even link with guest-resident dynamically-linked libraries. However, while the code is injected without the guest's knowledge or cooperation, it is certainly not protected from any malicious intent, and so must execute in the same limited privilege level as that of the guest itself. However, to provide extended functionality, VMM code must operate at a higher privilege level, and thus must provide protection mechanisms commensurate with this privilege. In this paper, we seek to allow guest-context VMM code to operate with extended privilege, while maintaining isolation from potentially malicious, unprivileged guest code.

4 Guarded modules

A guarded module is a component of the guest kernel for which the VMM provides a specialized execution environment. Typically, this implies an environment with higher privileges than normal. The combination of the VMM and the guarded module maintains the following invariant: Privilege is raised (the specialized execution environment is active) for a given virtual core 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 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. The "guarding" here refers to a garrison established between the module and the rest of the kernel by protecting the code of the guarded module and by securing the control flow into and out of it. However, the system must validate the stack to maintain this garrison, thereby insulating any stack variables used internal to the guarded module.

The guest kernel may read and write the code of the guarded module, although modifications of the code by any virtual core will eliminate any subsequent possibility of privileged execution in the guarded module. By default, the code of the guarded module that the guest kernel sees is its actual code, although it could be disguised by the VMM using well-known techniques. Also by default, the guest kernel can read and write the guarded module's data. The implementor of the guarded module bears the responsibility of insuring data integrity. The VMM can of course optionally provide access to data hidden from the guest kernel as part of the privileged execution environment.

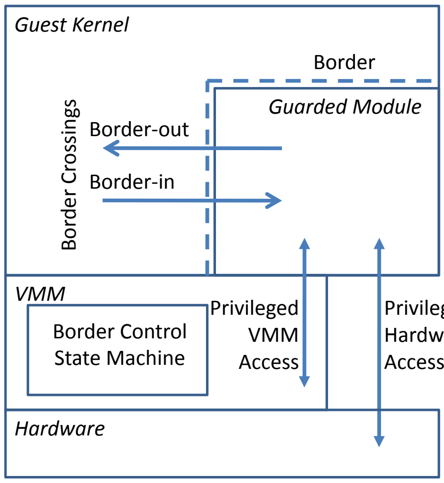

Figure 1: Big picture of guarded modules.

4.1 Guarded Linux kernel modules

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 described in Section 3.2. 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.

Figure 1 illustrates the run-time structure of our guarded module implementation, and documents some of the terminology we use. 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 2 illustrates how the two interact.

4.2 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 in 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

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

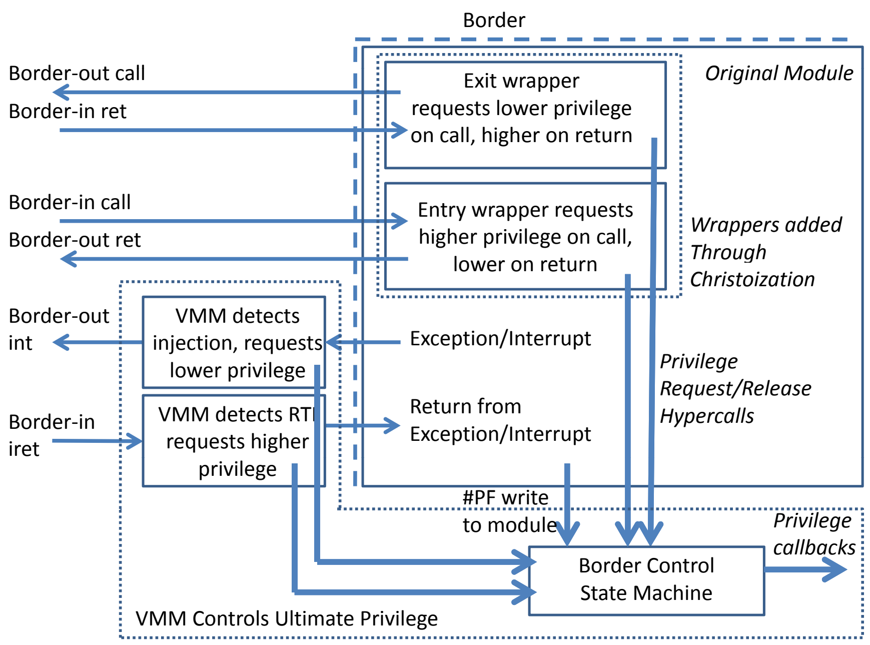
Border
Border-out call
Border-in ret
Exit wrapper requests lower privilege on call, higher on return
Original Module
Border-in call
Border-out ret
Entry wrapper requests higher privilege on call, lower on return
Wrappers added Through Christoization
Border-out int
VMM detects injection, requests lower privilege
Exception/Interrupt
Privilege Request/Release Hypercalls
Border-in iret
VMM detects RTI requests higher privilege
Return from Exception/Interrupt
#PF write to module
Privilege callbacks
Border Control State Machine
VMM Controls Ultimate Privilege

Figure 2: Guarded modules, showing operation of wrappers and interaction of state machine on border crossings.

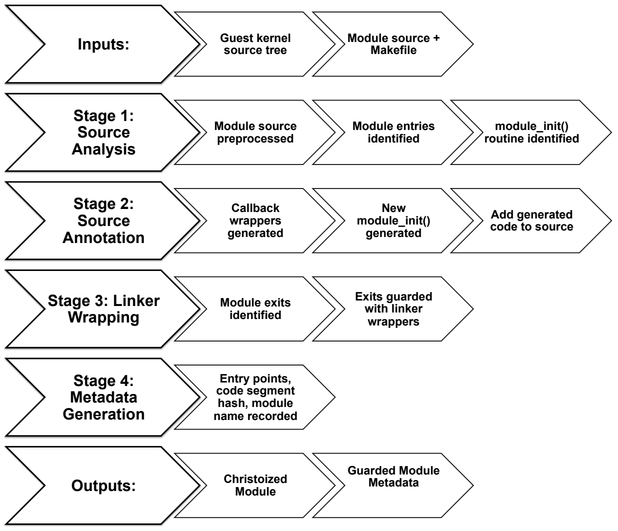
Inputs:
Guest kernel source tree
Module source + Makefile
Stage 1: Source Analysis
Module source preprocessed
Module entries identified
module_init() routine identified
Stage 2: Source Annotation
Callback wrappers generated
New module_init() generated
Add generated code to source
Stage 3: Linker Wrapping
Module exits identified
Exits guarded with linker wrappers
Stage 4: Metadata Generation
Entry points, code segment hash, module name recorded
Outputs:
Christoized Module
Guarded Module Metadata

Figure 3: Christoization process

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. Figure 3 depicts the Christoization process at a high level. 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 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 4. 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, and which are shown in Figure 5. 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 reproduce 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.

It is again important to note that the Christoization process is fully automated. The module developer need only intervene when the recognized entry points into the module do not meet his or her needs.

4.3 Run-time

The run-time element of our system is based around the border control state machine. As Figure 2 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.

```
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 4: An entry wrapper for a valid entry point.

```
exit_wrapped:
        popq  %r11
        pushq %rax
        movq  $border_out_call, %rax
(a)     vmmcall
        popq  %rax
        callq exit
        pushq %rax
        movq  $border_in_ret, %rax
(b)     vmmcall
        popq  %rax
        pushq %r11
        ret   (back into guarded module)
```

Figure 5: An exit wrapper for the valid exit point.

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 4(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 4(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 (Figure 5(a)) 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 (Figure 5(b)) 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 all 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 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 wrappers shown in Figure 4 and Figure 5 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` that is the last line of Figure 5. 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

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

border-out" stack frame, and computes and stores a hash of the stock content from the first entry to this last exit. 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.

4.4 Interfaces

We have previously described how the system works from the perspective of the guarded module developer (Section 4.2). We now consider it from the perspective of the developer of the selective privilege-enabled VMM component that the system drives, and from the perspective of the administrator, who controls which VMs get special selective privileges, and which guarded modules these privileges should be associated with.

The purpose of the selective privilege-enabled VMM component is to implement the actual raising and lowering of privilege. The developer of this component uses a simple registration function during the initialization of the component. The registration function associates a *privilege name* with a state pointer and a group of callback functions for the context of a specific VM. There are four callbacks, one each for initialization of selective privilege behavior, deinitialization of the same, and for raising and lowering privilege. The raise/lower callbacks occur per-virtual core, and are required to complete their operation before returning.

The administrator includes the guarded module extension in the guest configuration file, and gives the extension a name for the specific privilege it will export. This name matches the privilege name noted before, and is ultimately resolved upon guest creation. The administrator also uses a user-level tool to associate the guarded

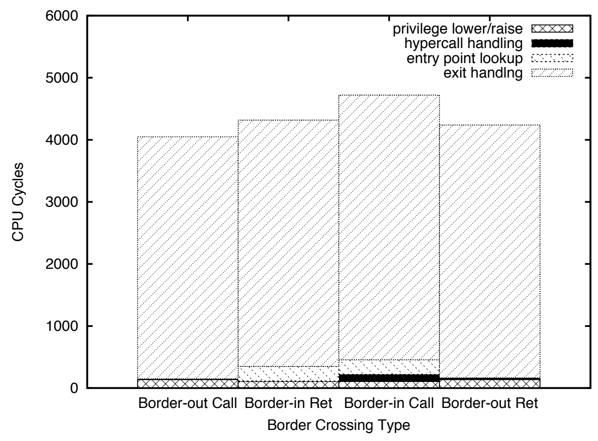

Figure 6: Privilege change cost without stack integrity checks.

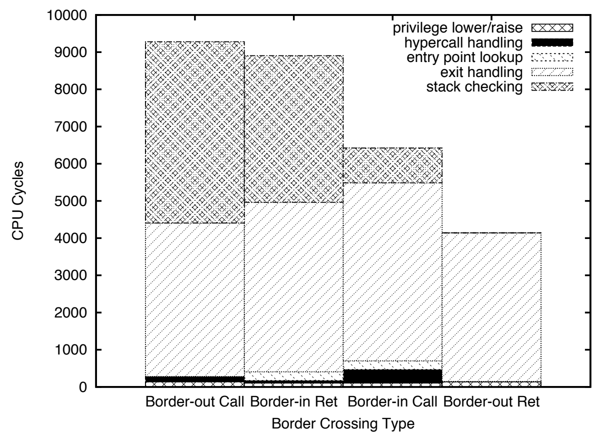

Figure 7: Privilege change cost with stack integrity checks.

module's metadata file with the privilege name within the context of a specific VM. Hence, the administrator has ultimate control over privilege.

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, for some crossings, the stack validation mechanism.

All measurements were conducted on a Dell PowerEdge R415. This is a dual-socket machine, each socket with a quad-core, 2.2 GHz AMD Opteron 4122 installed, 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. DVFS control is disabled.

Figure 6 illustrates the overheads in cycles incurred at runtime without stack validation enabled. All cycle counts were averaged over 1000 samples. There are four major components to the overhead. The first is the cost of initiating a lower or raise callback. This cost is very small at around 100 cycles. The second cost, labeled "hypercall handling", denotes the cycles spent inside the hypercall itself, not including entry validations and privilege changes. 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. Finally, "exit handling" is the time spent in the VMM handling the exit outside of guarded module runtime processing. At about 4000 cycles, this component dominates the border-crossing cost in our implementation in Palacios. The typical total border-crossing cost is only a few hundred cycles more than this. Reducing this base overhead relies on enhancements in the remainder of the VMM and in the hardware virtualization extensions.

Figure 7 gives a similar breakdown, but with stack integrity checking functionality enabled, and operation proceeding as described in Section 4.3. Here, the typical costs of border-out calls and border-in returns have grown to about 9000 cycles. The 5000 cycle expansion is due to to stack address translations and hash computations. Border-in calls also grow slightly slower due to the initial translation and recording of the entry stack pointer. Border-out rets are unaffected. The overhead of the stack integrity checking mechanism could be reduced by checking the minimal set of sensitive components on the guest stack, e.g. the return addresses and frame pointers.

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 Example

Having described the general design, implementation, and evaluation of our guarded module system within the Palacios VMM, we now consider a complete example of functionality built upon it. The goal of the example is to illustrate how a complete system operates and shake loose issues that are independent of the concept of guarded modules themselves.

In our example, we seek to provide a guest OS with direct access to a network interface card (NIC), but without trusting the guest. The NIC's device driver is available in the Linux kernel source tree, and can be compiled as a module. More specifically, we focus on the Broadcom BCM5716 Gigabit NIC. We Christoize the 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 Palacios's PCI passthrough functionality, which we have extended with selectively privileged operation. 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.

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.

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 4.3, 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 selective privilege 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.

The original host PCI device, in which the guest has fully privileged access to the underlying device at all times, comprised 500 lines of C. Adding selective privilege operation expanded the implementation to approximately 553 lines of C. Combined with the rest of the 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.

6.2 Selective passthrough for the NIC

The NIC uses exactly one BAR to define a 32 MB region of the memory address space. When it is mapped by the host PCI device in selective privilege operation, a privilege raise request causes our system to map the device by changing the shadow or nested page table entries corresponding to the guest physical address region the guest has chosen. On a privilege lower request, these entries are removed, so any subsequent access to those addresses will cause a page fault which eventually calls our hooked stub.

Privilege Change	Cycles	μs
Lower	4307	2.0
Raise	4800	2.2

Figure 8: System-dependent overhead for the NIC.

6.3 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.

Changing privilege In Palacios, memory region mappings, including those for regions that map to callback functions ("memory hooks") reside in a red-black tree mapped to the underlying nested or shadow page tables being used. Here, we employ nested paging and 64 bit operation. In this combination, the host PCI device, when fronting for the NIC, responds to a privilege raise request by: (1) removing the existing 32 MB region that maps the BAR to a memory hook, (2) modifying the corresponding nested page table entries, and (3) adding a new 32 MB region mapping the BAR to the actual device's memory region. Lowering privilege repeats steps 1 and 2, but for 3, the privilege implementation maps a memory hook region and marks the page table entries as unavailable so that nested page faults occur and redirect to callback functions. Assuming 2 MB superpages and suitable alignment, the system will adjust 16 nested page table entries in each case.

Function Name	Frequency
border-in call to	
bnx2_msi_1shot	0.23
bnx2_start_xmit	0.16
border-out ret from	
bnx2_msi_1shot	0.23
bnx2_start_xmit	0.16
border-out call to	
consume_skb	0.66
__phys_addr	0.01
border-in ret from	
consume_skb	0.66
__phys_addr	0.01
Border-in	**1.06**
Border-out	**1.06**
Total / Packet Send	**2.12**

Figure 9: NIC border crossing per packet send.

Figure 8 shows the measured system-dependent overhead for raising and lowering privilege for the NIC. We conducted all measurements in this section with the configuration described in Section 5.

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).

Border crossings per packet We instrumented the border-crossing handlers in our system to record the addresses at which the crossings happened, and then used those addresses to determine the functions within the guarded module and the guest kernel that were involved with these crossings. To generate traffic, we used ttcp in the guest to communicate with ttcp on a separate physical machine on the same Ethernet switch.

The NIC does interrupt coalescing, so determining precisely where individual packet transmissions/receptions begin and end presents a challenge. Instead, we also tracked the number of packets communicated over the interval of time that the ttcp traffic was active. Dividing the counts for the individual functions and the packet counts allowed us to determine the average number of border-crossings of each kind, and for each function.

Figures 9–10 show 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 reduce the overall number of border crossings per packet by pulling the implementations of these functions into the module itself. We leave further details and exploration of these leaf functions for future work.

Function Name	Frequency
border-in call to	
bnx2_msi_1shot	0.04
border-out ret from	
bnx2_msi_1shot	0.04
border-out call to	
skb_put	1.0
napi_gro_receive	1.0
eth_type_trans	1.0
__netdev_alloc_skb	1.0
__phys_addr	0.56
__napi_schedule	0.04
border-in ret from	
skb_put	1.0
napi_gro_receive	1.0
eth_type_trans	1.0
__netdev_alloc_skb	1.0
__phys_addr	0.56
__napi_schedule	0.04
Border-in	**4.64**
Border-out	**4.64**
Total / Packet Receive	**9.28**

Figure 10: NIC border crossings per packet receive.

6.4 Performance

Consider a machine sending packets on a TCP connection operating in congestion-avoidance mode. Each ACK received will clock a packet send. Given this pairing and the previous analysis, we would expect to see 11 border crossings for each packet sent on this TCP connection. At 5.2 μs per border crossing, we would expect to be able to transmit $\frac{1}{11\times5.2} = 17.4$ thousand TCP packets per second. With a 1500 byte MTU, this would be 26.1 MB/s (208 Mb/s). This the upper bound on the performance we should expect given the current overheads. With stack checking enabled, we would expect 18.9 MB/s (151 Mb/s).

Figure 11 shows the measured performance, using ping and ttcp, of the selectively privileged passthrough

with stack-check	
ping latency	0.186ms
ttcp throughput	129.54Mb/s
without stack-check	
ping latency	0.164ms
ttcp throughput	170.212Mb/s

Figure 11: NIC performance.

NIC in our system. The Christoized kernel module executes in the guest, and it interacts with the run-time elements of the system, which grant and revoke privileged access to the NIC as control flow enters and leaves the module. This combination constitutes a proof-of-concept of the entire guarded module system. The performance agrees with the bounds analysis given earlier.

7 Conclusions and future work

We presented the design, implementation, and evaluation of a system for allowing modules in the guest OS to obtain higher privileged access to the physical machine and the VMM. Our system is founded on joint compile-time and run-time techniques that bestow privilege only when control flow enters the guarded module at verified locations. We demonstrated an example use of the system by creating a passthrough NIC that only a designated guarded module we inject into the guest can program. The guest kernel can use this guarded module as it would any other NIC driver.

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), pp. 22–41.

[2] 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).

[3] 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.

[4] 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).

[5] 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).

[6] 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* (May 2006).

[7] 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)* (July 2007).

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

[9] 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), pp. 207–222.