Practical Taint-Based Protection using Demand Emulation

Alex Ho†, Michael Fetterman†‡, Christopher Clark†,
Andrew Warfield†, and Steven Hand†

†University of Cambridge Computer Laboratory
15 JJ Thomson Avenue
Cambridge CB3 0FD
{firstname.lastname}@cl.cam.ac.uk

‡Intel Research Cambridge
15 JJ Thomson Avenue
Cambridge CB3 0FD
{firstname.lastname}@intel.com

ABSTRACT

Many software attacks are based on injecting malicious code into a target host. This paper demonstrates the use of a well-known technique, data tainting, to track data received from the network as it propagates through a system and to prevent its execution. Unlike past approaches to taint tracking, which track tainted data by running the system completely in an emulator or simulator, resulting in considerable execution overhead, our work demonstrates the ability to dynamically switch a running system between virtualized and emulated execution. Using this technique, we are able to explore hardware support for taint-based protection that is deployable in real-world situations, as emulation is only used when tainted data is being processed by the CPU. By modifying the CPU, memory, and I/O devices to support taint tracking and protection, we guarantee that data received from the network may not be executed, even if it is written to, and later read from disk. We demonstrate near native speeds for workloads where little taint data is present.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; C.1.3 [**Processor Architectures**]: Other; D.4.5 [**Operating Systems**]: Reliability; C.4 [**Processor Architectures**]: Performance of Systems; B.8 [**Hardware**]: Performance and Reliability—*Miscellaneous*

General Terms

Security, Performance, Reliability, Design

Keywords

Demand emulation, emulation, false tainting, QEMU, tainting, virtual machine, virtualization, Xen

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'06, April 18–21, 2006, Leuven, Belgium.
Copyright 2006 ACM 1-59593-322-0/06/0004...$5.00.

1. INTRODUCTION

Modern computer systems live under a constant threat of exploitation. The Internet is well-known to host a multitude of viruses and worms, which attempt to compromise the security of system software. The problems caused by such attacks are also well understood: systems are modified to present advertisements, capture and log user activity, or assimilated into botnets where they may be used as drones to attack or infect other systems.

Many well-known exploits take advantage of the generality of modern "general-purpose" computers. Data is treated as an amorphous collection of bits, which may be used in arbitrary roles: In a stack-smashing attack, the data composing a web request is made to include modifications to a server's call stack that compromise the system. Viruses and trojans may be attached to applications or included with email, and the execution of these applications may perform malicious modifications to an unsuspecting host. Many of the vulnerabilities associated with these attacks stem from the ability to execute arbitrary data received from the network. However, in a hostile Internet environment of connected hosts under constant attack, it seems sensible that the ability to execute data received from the network should be the exception rather than the norm.

Several researchers have observed this problem in the past and have explored system extensions to the way that data is managed in order to prevent such exploits. The common approach used is that of *tainting*: memory containing data received over the network is specially marked and its contents are prevented from being executed. While past systems have demonstrated the effectiveness of tainting in avoiding compromises, they generally use full-system hardware emulation or simulation. The unfortunate consequence of these techniques is a severe performance penalty, rendering the resulting system impractical for deployment outside of a laboratory environment. Other approaches improve performance but at the cost of limiting their scope to user-space code within an individual process.

This paper presents an implementation of taint-based system protection that is practical for use on modern workstations. Our approach embodies two properties that have not simultaneously been demonstrated by previous systems: it is *comprehensive* and *high-performance*. First, by tracking data throughout the entire operating system, and not just within a single application, and by preserving taint annotations even on data that is written to disk, the system achieves comprehensiveness,

maintaining the invariant that tainted data may never be executed. Second, by limiting the use of hardware emulation to just those regions of code that interact with tainted data, we drastically reduce the overhead of tracking tainted data and protecting the system while retaining system-wide safety.

Our approach uses the Xen virtual machine monitor to run a protected OS within a virtual machine, and *dynamically switches execution* on to and off of a hardware emulator as taint-tracking is required. The coarse-grained combination of virtualization and full-system emulation, which to our knowledge is the first such implementation, allows the efficient incorporation of what is essentially a new hardware feature in an OS-agnostic manner on commodity systems.

While our approach is likely to be too expensive in its current form to deploy on servers, we believe it is sufficient to demonstrate the viability of the technique. We validate this claim by exercising our prototype implementation with a suite of microbenchmarks as well as a set of network-intensive tests. We demonstrate near native speeds for workloads where little taint data is present.

2. RELATED WORK

There has been significant previous work in the use of tainting to protect user applications. We focus on those techniques that target running applications and do not require *a priori* source or binary modifications to existing applications under the belief that solutions requiring either re-compilation or re-linking of existing applications are very unlikely to be deployable in practical settings.

Tracking tainted data from untrusted source into a processor and through a taint-enhanced memory model has been used with great success in a number of projects. The target application either can be executed in an emulator [1] or dynamically re-written at runtime [2, 3]. These techniques prevent the application from transferring control to untrusted code but, as mentioned above, result in systems that are too slow to deploy.

There also have been a number of proposals for architectural techniques to incorporate tainting into a processor [4, 5, 6]. These schemes maintain taint information in a bit associated with each memory byte and modify the processor to propagate this taint information. The processor is also prohibited from loading tainted data into the program counter. All of these proposed architectures have been prototyped using hardware simulators and not used on real platforms.

Despite the performance issues, however, this prior work has validated the usefulness of data tainting from a security perspective and demonstrated its applicability against a wide range of attacks. In this paper, we extend the case for the security benefits of data tainting by attempting to make it sufficiently low-overhead to deploy in real systems.

The notion of marking potentially dirty data beyond the processor has been proposed by Madsen [7]. A multi-level attribute associating a level of trustworthiness can be assigned to each file, and these trust levels can be combined to formulate a security policy. Unfortunately, associating trust at the file level requires changes to the operating system and is dependent on both the particular file system and operating system. Our use of virtualization allows us to assign taint at the underlying block level, independent of any file system or OS that may read and process the block.

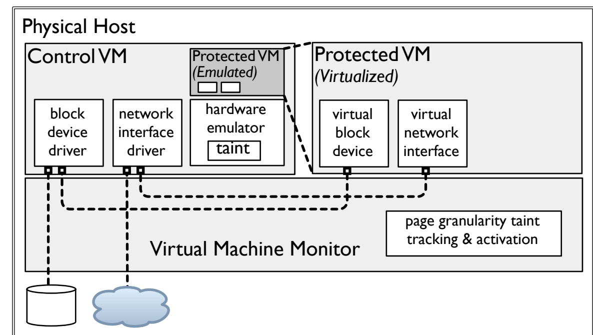

Figure 1: Taint-based protection architecture.

A number of environments for interpreted languages now offer automatic tainting for language variables. These include the taint mode to Perl [8], safe levels in Ruby [9], JavaScript (in Netscape Navigator 3.0) [10], and PHP [11].

Tainting can also be used to track the propagation of "dirty" data in a system. The TaintBochs project [12] analyzes both the propagation and liveness of tainted data in applications using a modified version of the Bochs x86 simulator. No attempt is made to prohibit the execution of tainted data; instead they maintain extensive logs of the propagation of taints and perform analysis at the end of the run.

Both virtualization and emulation have been used to detect and respond to network-based security exploits in very recent work. The Potemkin Honeyfarm [13] uses a large VM-based cluster to isolate exploits, with plans to quickly generate and disseminate signatures for emerging attacks. Somewhat similarly, Vigilante [14] uses dedicated "detection engines", some of which using data-flow analysis based on binary rewriting, to generate attack signatures. Argos [15] extends Vigilante by utilizing emulation to allow system-wide tracking of tainted data independent of the operating system.

These systems depend critically on the generation, dissemination, and incorporation of signatures being faster than actual worm outbreaks, which is incredibly challenging in the face of hitlist-based worms. Our approach attempts to make data-flow-based detection of sufficiently low overhead that it may be used to directly protect individual end systems.

Limited forms of execution protection have become available for modern hardware in recent years. Many CPUs now provide support for disabling execution permissions (NX) for regions of memory. Using OS modifications it has also been possible to provide stack and heap execution protection with reasonable efficiency [16].

Our implementation uses a modified hardware emulator to effectively add a new, data-flow-based protection mechanism to the system hardware. Simulation and emulation have often been used in this manner to prototype new hardware features including large SMP systems [17], symmetric maltreating [18], and fine-grained memory protection [19]. Our work improves on this general approach for some forms of extension by only emulating when necessary: when tainted data is not in processor registers, the system resumes full-speed, virtualized execution.

Type	Goal	Mode	Modification
Memory	Annotate tainted data at a byte granularity.	**V** **E**	Page-size tracking to trigger fault-based entry to emulation. Byte-granularity tracking of all VM memory.
CPU	Track tainted data through memory and raise an exception if an attempt is made to unsafely use tainted data.	**V** **E**	Fault on access to tainted data and enter emulation. Update tainted memory annotations as data is manipulated; trigger return to virtualized execution when registers are clean. Signal errors on use of tainted data.
Network	Mark received data as tainted before it is presented to the OS.	**V**	Virtual network device annotates memory as tainted on receive DMA path.
Disk	Track tainted data across access to persistent storage.	**V**	Block device maintains taint data on stored blocks, updating this on writes and annotating memory on reads.

Table 1: Virtual hardware modifications to support taint tracking. The mode field distinguishes between Virtualized and Emulated execution where appropriate.

3. SYSTEM OVERVIEW

This section provides a broad overview of the design of our taint-based protection prototype. We begin with a brief description of our system and then move on to describe the virtual hardware modifications we have made to support taint tracking and protection. Next, we discuss the lifespan of tainted data as it travels through the system. Finally, we discuss how we explicitly untaint data we wish to execute.

Applications run in a protected virtual machine running on a virtualized CPU at native speeds. If the processor accesses tainted data, untrusted bytes originating from outside the system, then the VMM switches the virtual machine from the virtual CPU to an emulated processor running as a user-space application in a control VM. The system then proceeds on the emulator, which tracks the propagation of the tainted data throughout the system. Once the emulated processor ceases to manipulate tainted data, the VMM can revert the system back to virtualized execution.

3.1 Virtual Hardware Support

We have designed taint tracking and protection as a broad set of hardware modifications to a commodity computer system. Since we want to ensure that all external data received from the network be tainted and that the tainting of data be preserved as it propagates through memory and persistent storage, these modifications involve changing the behavior of the CPU, memory, and I/O devices. While we use the combination of virtual and emulated execution to achieve performance, our modifications are made completely outside the protected virtual machine and require no changes to the protected OS or applications.

Figure 1 shows the high-level architecture of the system. As illustrated, the protected VM runs above a virtual machine monitor, and is assisted by a control VM, where the majority of the architectural modifications are implemented. Table 1 describes the individual modifications to virtual hardware that are required to track tainted data. The table additionally contrasts that modifications that are in place in each mode of execution; these will be discussed in more detail in Section 4.

3.2 Taint Model

Our taint model is based around a conventional von Neumann architecture with a CPU, memory, and I/O devices. Like program shepherding [2], we do not prohibit the transfer of tainted data through the system. Instead, we monitor the movement of data and ensure that tainted data is not used inappropriately.

We divide the CPU registers into two classes and track the current taint state of each member of the first class:

- **Data Registers**: These are the general purpose data registers for the architecture. Data is loaded into the registers, manipulated, and then stored back into memory. They also include floating point registers and any SIMD vector registers.

- **System Registers**: These additional registers are used to manage the processor's control plane and system resources, such as the program counter, stack, memory management, and processor operation modes.

Tainted data can flow freely between memory and data registers. System registers, however, should never be tainted; any attempt by the processor to move tainted data into a system register results in a taint violation. This is described more fully in the next section.

The taint model prevents processor operation from being affected by untrusted data from the outside world. Similarly to previous taint work [1], we consider the program counter a system register and prevent the processor from either loading or executing tainted data. Furthermore, we also deem the stack pointer a system register and prohibit loading tainted data. This prevents a stack frame overrun from constructing a frame pointer that causes activation records to be skipped on return from a subroutine.

Our model prevents the processor state from being directly affected by tainted data. Indirect attacks are still possible as, like other taint-based protection schemes, data-flow is not tracked across comparisons or arithmetic operations. However, because we trust the initial state of the machine, we assume that the system does not *a priori* include malicious code that an attacker can exploit. This point is discussed more thoroughly in Section 3.4 below.

The taint model also does not prevent attacks where application semantics are changed based on tainted data. Such attacks do not try to corrupt the processor state directly and so remain undetected. For example, a bug in ssh [20] overflowed the stack and placed a zero into a particular stack variable. This adulterated stack variable was later used as the userid of the ssh shell, thus allowing an intruder to gain privileged access to the host.

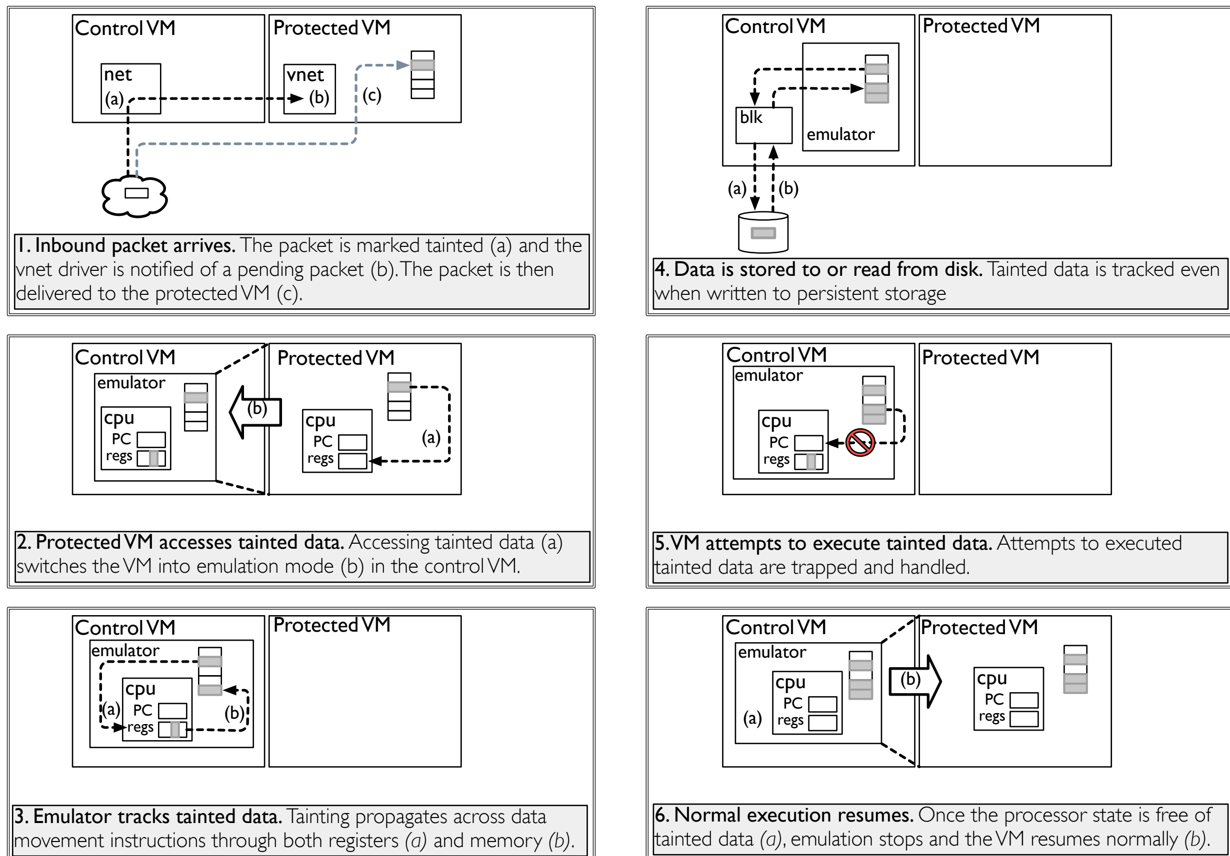

Figure 2: Tracking tainted data throughout the system.

3.3 Lifespan of Tainted Data

Figure 2 illustrates how tainted data is managed as it moves through the system. In Figure 2-1, a packet arrives at the host from the physical network interface. The packet is processed in the control VM, which is hosting the network device and the pages into which the data has been received are marked as tainted as they are transfered into the protected VM's pseudo-physical address space.

Shortly later, the protected VM handles an interrupt indicating that a new packet has been delivered. As received data is processed, the CPU attempts to load data from the tainted memory pages into registers. This results in a fault, and causes the VM to be switched into emulation (Figure 2-2). The current (virtualized) CPU state is preserved and execution is transferred into a hardware emulator running in user-space in the control VM. The emulator has direct mapped access to the VM's memory and so continues to execute the machine in place.

As this execution progresses, as shown in Figure 2-3, emulated instructions operate on tainted data. The emulated processor microcode is modified to track tainted data across memory and register stores. Stores of untainted data result in the cleaning of tainted memory. In this manner, well-behaved OSes will clean tainted memory as they zero freed pages.

Figure 2-4 shows that tainted data is also tracked across storage to disk. The virtual disk is modified to store an on-disk data structure mapping the location of tainted data. If data is re-read then the memory it is placed in is marked appropriately.

Figure 2-5 illustrates what happens if a VM attempts to execute a piece of tainted memory or load tainted data into a system register: an instruction exception is thrown, enabling the operating system to react as appropriate. Such events can occur when the VM loads the address of tainted data into the EIP, or when it simply advances execution into a tainted region of memory. The exception thrown could be an existing processor fault, such as invalid opcode, or we could extend the processor with a specific "tainted" exception. To minimize operating system software changes we opt for the former of these two options in our work.

Finally, after processing a region of tainted data, the processor finds that its registers have become clean (Figure 2-6). At this point, it executes to an appropriate opportunity and then transfers the processor state back to virtualized execution.

3.4 Managing and Executing Tainted Data

We demonstrate that the addition of a sweeping hardware feature, such as taint-tracking, can be efficiently achieved on commodity hardware. This work provides all the necessary mechanism for end-to-end taint protection on modern hardware, with sufficiently reasonable performance as to be deployed on desktop systems. Our current taint model is intentionally simple, for purposes of exposition and validation—it is the same as that described in systems such as Vigilante [14]. This model trusts the software on the host not to maliciously remove tainting. We believe that there is a clear opportunity to explore alternative taint models.

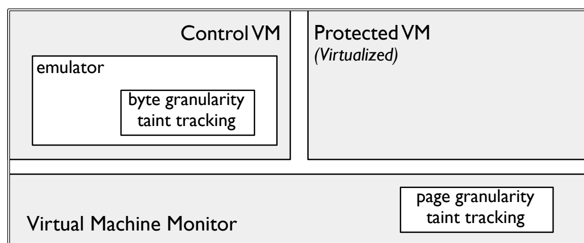

Figure 3: Tracking tainted memory. Tainted memory is tracked by two synchronized data structures: A byte-granularity hash table resides in the emulator, and a coarser page-granularity bitmap in the VMM.

3.4.1 *Untainting Data for Execution*

There are some situations where it is desirable to execute data received from the network. To allow users to install and run new applications or system upgrades, the system must provide a mechanism to untaint data. Furthermore, untainting should be as simple as possible to avoid users becoming discouraged with the additional security provided by taint-based protection. In order to achieve this, we require that data to be untainted be written to disk. We then provide a utility, called `bless`, which removes taint annotations from the blocks accordingly.

4. IMPLEMENTATION

Our implementation is based on the Xen VMM [21] with hardware emulation provided by a modified version of QEMU [22]. This section details how we have modified and combined these packages to provide taint-based protection.

4.1 Tracking Tainted Memory

Our design requires the addition of fine-grained taint tracking to the host architecture. We chose to track tainted data at a byte granularity, for fear that coarser (e.g. page granularity) tracking would lead to excessive propagation of taint annotations and, as a consequence, incur a higher performance overhead. Tracking of tainted data must strike a balance between two aspects of the system: As shown in Figure 3, the emulator must have fast access to a byte-granularity index of tainted memory, while the VMM must be able to map page-granularity taint information onto MMU hardware to activate emulation when necessary.

4.1.1 *Tracking Tainted Bytes (Emulation)*

The taint tracking data structure is a byte-granularity mask describing the location of all tainted data in the system. As the vast majority of accesses to this data structure are made in emulated execution, it is allocated inside the emulator's address space. This has the additional benefit of avoiding memory overhead and churn within the VMM itself. This data is store in a bit-per-byte bitmap representing all of the physical memory of the protected VM.

Table 2 contains a list of calls which are used to track and check tainting for the processor and memory; most are self-explanatory. Note that since control registers can never be tainted, there are no calls to set or query them.

The interface to this data structure is the only mechanism by which memory is marked or unmarked as being tainted. After the byte-granularity tracking structure is updated, these markings are generalized to a page granularity bitmap that is shared with the VMM.

void taint_memory_page(physical_addr, taint)
: Taint or untaint a page of physical memory. Used by device drivers when loading external data.

void taint_memory(virtual_addr, length, taint)
: Set the taint of a region of virtual memory.

boolean check_memory(virtual_addr, length)
: Check if any byte in region of virtual memory is tainted.

void taint_register(register_id, taint)
: Set the taint for a data register.

boolean check_register(register_id)
: Check if a data register is tainted.

Table 2: Taint API

4.1.2 *Tracking Tainted Pages (Virtualization)*

Unfortunately, the x86 hardware does not provide a general mechanism to generate memory faults at byte-granularity or based on physical addresses. To overcome these limitations, we ensure that any active page table entries pointing to tainted data are marked as not present, resulting in an fault into the VMM on their access. As the hardware protection is at page granularity, the shared tainted-physical-page bitmap is consulted on page faults. If a faulting address maps to a tainted page, a switch to emulated execution is triggered.

A second challenge with the x86 hardware is in mapping a list of tainted physical pages on to the virtual address space of all applications that may attempt to access it. As data is marked as tainted, it is important that all references to it reflect this fact and result in a page fault on access.

Our approach takes advantage of shadow page tables [23, 24, 25] to address this issue. This technique involves maintaining a protected version of a VM's page tables in hypervisor memory, beyond the reach of that VM. These shadow tables are used by the hardware MMU, and so contain authoritative information on a VM's virtual memory—the page tables held within the VM may be freely modified by the virtualized host, and are validated and translated into the live shadow tables as necessary. In the case of our implementation, new page table entries are validated against the bitmap of tainted pages and marked accordingly.

Using shadow page tables provides a level of indirection between the VM and physical memory, and allows new MMU functionality to be implemented in software. Other recent examples that use shadow page tables to extend the functionality of memory include live VM migration [26], which tracks page dirtying as a VM is iteratively copied over the network, and "flash cloning" [13] in which a running VM may be duplicated while memory is shared between the two VMs in a copy-on-write fashion.

The shadow fault handler is called whenever a page fault is received from the protected VM. In our implementation the fault handler is responsible for two important tasks. First, it tests to see if a faulting page is marked as tainted. If so, it flags that the VM should be restarted in emulation mode when it returns from the fault. Second, as an optimization, the shadow code maintains a list of code pages that have been translated and cached by the emulator. It ensures that these pages are always mapped read-only, and records them as requiring invalidation on update faults.

```
void op_movl_A0_im (u32_t imm)
{
  A0 = (u32_t) imm;
  taint_register(S_A0, 0);
}

void op_ldl_T0_A0 (void)
{
  T0 = (u32_t) ldl_p((u8_t *)A0);
  taint_register(S_T0,
                 check_memory(A0, 4));
}

void op_movl_EAX_T0(void)
{
  EAX = T0;
  taint_register(R_EAX,
                 check_register(S_T0));
}
```

Figure 4: QEMU micro-operations to load the accumulator with the contents of a memory location.

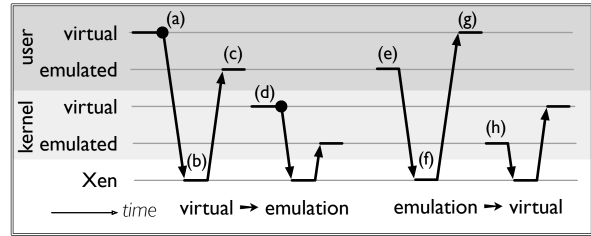

Figure 5: Transitioning between virtual and emulated execution. A virtualized host takes a page fault in either user (a) or kernel (d) and enters Xen; the shadow fault handler identifies that the page is tainted (b); and then returns to the same execution context in emulation (c) where the processor state is copied into the emulator. When a decision is made to end emulation (e,h), the processor state is copied back to Xen and a hypercall is made for Xen to unpause the virtual machine (f); finally, virtual execution continues (g).

4.1.3 Instruction-level Taint Tracking

QEMU splits each x86 instruction into a number of micro-operations that we have annotated with tainting logic. For example, the instruction to load the accumulator with the contents of a memory location (`movl $c0379da8, %eax`) is decomposed into a sequence of three micro-operations: one to load the memory address into a temporary register, one to load a second temporary register with the contents of memory, and a third to move the contents of the second temporary register into the accumulator. This decomposition is illustrated in Figure 4.

These RISC-style micro-operations provide a convenient place at which to add taint tracking. Referring once again to Figure 4 we see that the load of an immediate value into the (temporary) address register `A0` clears any possible previous taint; the load of a memory word into `T0` propagates the taint information associated with that 4-byte region of memory; and the assignment of `EAX` from `T0` simply propagates `T0`'s taint status.

4.2 Demand Emulation

Our implementation hopes to achieve high performance for what is effectively a "hardware extension" requiring emulation, but where emulated execution is not the common case. Assuming that only a limited portion of execution time is spent dealing with tainted data, we may run mostly in virtualized mode, with a very small execution overhead, and then switch into emulation only "on demand" while tainted data is being processed. We call these dynamic transitions between virtual and emulated execution V2E and E2V.

The addition of this live emulation support to Xen has required only minimal modifications to the hypervisor. The decision to transition into emulation is made immediately before control is passed from VMM to VM context. At this point, a flag is tested to determine if emulation is required. To switch to emulation, the VM is paused, and an interrupt is sent to the control VM instead of the activation of the virtualized instance. As such, QEMU acts very much as a drop-in software replacement for the physical CPU that may be turned on and off at a per-instruction granularity.

4.2.1 Entering emulated execution (V2E)

The left-hand side of Figure 5 shows entry to emulation from a taint fault in native execution. As was mentioned in Section 4.1.2, we transition from virtual to emulated execution whenever a fault is taken on a page of memory that has been marked as containing tainted data. The shadow fault handler in Xen marks the VM as requiring redirection on hypervisor return. While the emulator has all of the protected VM's memory directly mapped, the processor context must still be transferred before execution can proceed. Table 3 shows the IA-32 processor state that is transferred from Xen to QEMU.

Emulated instructions must still handle both faults, traps, and interrupts. The left-hand side of Figure 6 demonstrates how these are handled in our current implementation. In the case of faults, we chose to preserve the regular fault entry semantics provided by Xen, which authenticates hypercalls based on the calling VM. An alternative approach would be to add a "remote fault" entry, where the emulator could directly issue faults on behalf of another VM. When a faulting instruction is recognized by the emulator, it makes a hypercall to Xen, asking that the faulting instruction be reissued by the virtualized VM. On entry to Xen, a re-entry count is provided to indicated the number of VM entries that are to be allowed prior to returning to emulation. On faults, this value is set to one, allowing a single instruction to be evaluated.

Although this technique for faulting maintains the existing fault interfaces within Xen, the "double-bounced" routing through the hypervisor does represent a performance overhead. Our approach to date has been to design for stability, and the current technique avoids significant changes to the VMM. Additionally, emulation is sufficiently expensive that this may not be a key overhead to address. That said, the addition of a "remote fault" interface would eliminate protection VM crossings and will be interesting to investigate as future work.

Interrupts are handled similarly to faults, but require less routing across protection domains. However, as with virtualized execution there is no notion of a direct hardware interrupt to the virtual machine. Interrupts are handled directly by Xen and then translated to *event notifications* in the VM. Event notifications are a configurable and maskable bitmap of "ports"—effectively a virtual APIC—which are mapped as virtual interrupts to the VM. The bit-mask is stored on a page of memory shared between the hypervisor and each VM, and a summary "event pending" bit is tested whenever a VM is scheduled.

	#bytes
7 debug registers	28
8 general purpose registers	32
program counter (EIP) & status (EFLAGS)	8
4 control registers[1]	16
GDT and LDT base and limit	16
6 segment selectors	24
x87 floating point state and registers	86
XMM state and registers	132
Total	342

Table 3: IA-32 processor state transferred between Xen and QEMU.

The emulator must deal with interrupts similarly. On VM startup, the emulator locates the shared memory page containing the event bitmap. As long as a VM in running in emulation, this bitmap is tested regularly in between the execution of basic blocks. If an interrupt is detected:

- the emulator traps to Xen (with a re-entry count of zero);
- Xen prepares the interrupt frame; and
- execution resumes in the appropriate context, where the interrupt handler is run.

This is illustrated on the right-hand side of Figure 6.

It is not worth polling shared memory for interrupts at a finer granularity than emulated basic blocks since the CPU is not necessarily in a consistent state to be interrupted at such points. However, polling between basic blocks is complicated by the optimizations employed by QEMU which adaptively link consecutive basic blocks together. Tight loops, for instance calculating the CPU speed at boot, may end up linked together and eliminate opportunities to poll altogether. To address this, we introduce an emulated timer interrupt, which tests for pending interrupts every 50ms.

4.2.2 Returning to virtualized execution (E2V)

Returning from emulation to virtualized execution presents an interesting problem. A naïve approach might define a function *taint_check()* that checks if all data registers are clear of tainted data, and then invoke this after every instruction. This would be unwise for several reasons. First, testing the register set for tainted data is far too expensive to do every operation; secondly, the emulated CPU may temporarily be in an inconsistent state and require expensive canonicalization; and finally, since the x86 CPU has only eight registers it is entirely possible for all of these to become transiently clean in the middle of an operation on tainted data. Aggressively exiting emulation in situations such as these could result in thrashing in and out of emulated execution.

Instead our initial implementation allowed the emulator to run as long as possible and deferred the call to *taint_check* to the next event which forced QEMU to re-enter Xen. This turned out to be too long and we found ourselves emulating code unnecessarily.

[1]Tracking control registers is simplified by merit of the fact that Xen paravirtualizes the hardware interface. Many registers, such as machine specific registers and the task register, do not need to be tracked in V2E/E2V transitions.

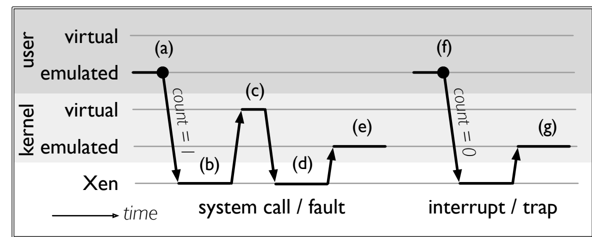

Figure 6: Handling system entry in emulation. When the emulator encounters a fault (a), it traps to Xen, which enables single stepping (b) and allows the faulting instruction to reissue from the VM (c). This faults into Xen normally, where the fault is handled (d), and control is passed to the emulated fault handler (e). Traps and interrupts enter Xen (f), and are redirected directly to the OS handler in emulation (g).

In an attempt to provide a low-overhead exit from emulation and avoid thrashing between the two execution modes, we have added a degree of hysteresis to when *taint_check* is called. During basic block execution, we count the number of consecutive memory accesses to untainted pages, an inexpensive operation given our bit per memory page bitmap. Upon completion of a chain of basic blocks, we check to see if the previous x memory references were untainted, and if so call *taint_check*.

We currently set $x = 50$ empirically, based on our development and evaluation workloads. Further evaluation is required to determine whether another value or another algorithm, perhaps with a higher degree of hysteresis, would be more beneficial. See Section 5.3 for additional details.

4.3 Device Extensions

Tainted data is tracked across network and block device interfaces by implementing *soft devices* [27]—in essence, extended virtual hardware—which may interact with I/O-related data outside the scope of the protected virtual machine. Since it is unsafe to share direct hardware access between multiple virtual machines, Xen uses "split drivers": a *driver VM* has direct access to the hardware, and presents an idealized view of the device to other VMs. A driver for this interface is installed in the client OS, and the driver VM multiplexes access to the physical device. In our implementation, the emulator, physical drivers, and device extensions all share a common VM, although they could potentially be separated out for additional robustness.

4.3.1 Tainting Network Data

Xen's current network interface delivers packets to individual VMs using page remapping. Inbound packets are received into a system-wide pool of free pages, which have been "donated" by VMs that use the network. A packet is written into an empty page, routed to the destination VM, and delivered to that VM by mapping the page into its physical address space, in place of a previously donated page.

We extend the virtual network interface to mark received packets as tainted before they are delivered to the protected VM. As a packet is received, the network interface passes the physical address and length of the packet to the taint-tracking module in the emulator, where it is incorporated into the system taint list. This marking is done asynchronously with batches of packets, but is always completed before the data is made available to the protected VM.

Processing packet headers results in considerable time spent in emulation, and involves code that is generally well tested and trusted to be safe. As a performance optimization, we allow the virtual network interface to be configured to deliver untainted packet headers. In addition to mapping the packet's page in the receiving VM, the network interface does some basic sanity checking, and then places an untainted copy of the packet headers in shared memory, which the protected VM may process without incurring taint faults. Similarly, for various network control packets (e.g. ARP packets, ICMP pings), the device driver can verify the integrity of the packets and bless them. The operating system does not need to enter emulation each time these packets arrive, and internal kernel data structures such as routing tables can thus be kept clean. In this sense, we effectively provide a "smart NIC" which blesses data packet headers and network control packets as being safe.

4.3.2 Tainting Storage

The virtual storage interface in Xen is structured similarly to that of the network, but does not involve page exchange since the target memory page for reads and writes always belongs to the protected VM and is known at the time of the request. The virtual disk device, running in the control VM, maintains a persistent data structure—a sparse tree—identifying all disk blocks that have been marked as tainted. On writes, the taint properties of memory are preserved on disk. An additional benefit to this data structure is that it is not part of the virtual disk and so cannot be addressed by the protected VM.

On reads from disk, the target pages are rendered invalid while the request is serviced by the control VM. Before the data is made available to the protected VM, the taint markings are updated as appropriate.

4.3.3 Selective Tainting

In some situations, it may be desirable to strike a further compromise between the performance overhead of taint tracking and the thoroughness of the system. As an extension of the header exemptions described for the virtual network interface in Section 4.3.1, we have modified a commonly used intrusion detection system (IDS), called `snort` to monitor all traffic that will be delivered to the protected VM.

This approach allows an interesting extension to the functionality of the IDS, in which it may divide inbound traffic into three classes:

- *Attack traffic* for which IDS rules already exist is simply dropped as normal;
- *White-list traffic* received from trusted hosts or destined for particular trusted services is not be marked as tainted;
- *Other traffic* which is treated as untrusted and passed into the system with taint-based protection.

One benefit to this technique is that it allows administrators to focus on the set of traffic that they do trust, rather than that which they do not, but without requiring unanticipated traffic to be dropped. Instead it is simply treated with an enhanced degree of protection.

Our current system includes a complete implementation of this functionality but since it represents a mechanism to strictly *reduce* the overhead of taint-tracking by tainting less data, we do not enable it in our performance evaluation.

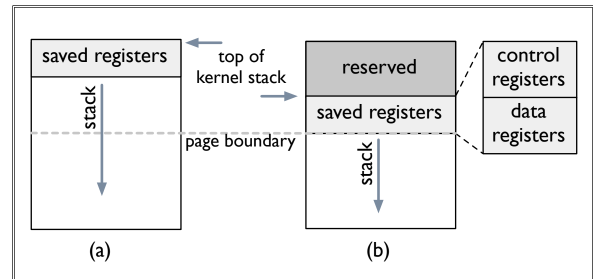

Figure 7: Default Linux kernel stack layout (a) for interrupt and exception processing, and modified stack layout (b) with potentially tainted registers on a separate page.

4.4 Operating System Enhancements

The implementation described thus far has been software agnostic. By modifying the virtual CPU, physical memory and devices offered to a virtual machine by a virtual machine monitor, our technique is independent of both the operating system and any user applications and does not require any changes.

Unfortunately, existing software has not been written to interact efficiently with page granularity tainting—the placement of tainted and untainted data together on the same memory page can force unnecessary transitions into emulation. These time consuming switches and the slowdown inherent in emulation can both be avoided simply by altering the placement of data structures in memory. We call this problem *false tainting* by analogy with the false sharing problem observed in multiprocessor cache coherence protocols and distributed shared virtual memory systems.

As a particular example, consider when a user application is suspended, due to either an exception such as a page fault or an external interrupt. The kernel saves the application's registers on the kernel's stack and then proceeds to handle the exception. However, if the application was processing tainted data at the time, one or more registers containing tainted data would be pushed onto the kernel stack as shown in Figure 7(a). The mere presence of tainted data on the kernel stack would most likely force the operating system to process the exception under emulation, even though no tainted data is accessed.

We minimize this by preallocating some space on each kernel stack. By default, Linux allocates two pages, or 8 Kbytes, for each kernel stack. By preallocating exactly the right amount of space on each stack, the user's possibly tainted registers are stored on the higher page, while all the call frames used while processing inside the kernel sit on the lower page (Figure 7(b)). We also zero the user's registers immediately after saving them, to keep any possibly tainted values from escaping into the kernel. While emulation may still be required for placing the user's tainted registers onto the higher page, and for retrieving them again just before exiting the kernel, this allows the majority of the kernel code to run in native mode. However, it does reduce the amount of space available to Linux on each kernel from 8 Kbytes to just over 4 Kbytes.

Fortunately, there is a Linux kernel configuration option that allows Linux to survive with only 4 Kbytes per stack (`CONFIG_4KSTACKS`). It does this by requiring all interrupts (and in particular, interrupts which occur while already executing inside the kernel) to switch to another, dedicated 4 Kbyte stack,

Configuration	stat (μsecs)	signal handling (μsecs)	fork (μsecs)	fork shell (μsecs)	file create (μsecs)	mmap latency (μsecs)
Linux	1.19	1.59	70	5,641	25.1	268
Xen Linux	1.46 (1.2x)	1.78 (1.1x)	197 (2.8x)	6,244 (1.1x)	27.1 (1.1x)	593 (2.2x)
Modified VM	2.02 (1.7x)	2.59 (1.6x)	208 (3.0x)	6,421 (1.1x)	29.6 (1.2x)	600 (2.2x)
Protected VM	2.08 (1.7x)	2.47 (1.6x)	259 (3.7x)	6,724 (1.2x)	29.6 (1.2x)	660 (2.5x)
100% emulated, & no tainting	49.60 (41.7x)	57.78 (36.4x)	7,420 (106.5x)	139,668 (24.8x)	467.5 (18.6x)	16,480 (61.5x)
100% emulated, & tainting	111.00 (93.3x)	176.14 (110.7x)	11,905 (170.8x)	415,979 (73.7x)	1,838.2 (73.2x)	23,699 (88.4x)

Table 4: Representative LMbench test results.

which are statically allocated per-CPU. By enabling this option and then allocating 8 Kbytes for each stack anyway, we keep the user's tainted data from slowing down system calls, interrupt handling, and exception processing in the kernel.

It is still possible for an interrupt to occur while processing tainted data in the kernel. If one or more of the registers are tainted, then they will taint the kernel stack as they are saved. Since the Linux `CONFIG_4KSTACKS` option will cause this interrupt to quickly switch to a separate stack, the interrupt handler itself does not end up using a tainted stack, but upon returning from the interrupt, the kernel's stack is now tainted. To combat this, we also zero the register save area on the kernel stack immediately after restoring the user's registers, and just before returning from the interrupt.

These modifications allow the kernel to process interrupts and exceptions almost entirely in native mode.

Another Linux-specific example involves buffer management in the TCP stack where multiple partially consumed buffers can be coalesced to save memory. This process can have the unfortunate side-effect of migrating tainted packet data back onto the same page as the packet header and negating the performance optimization described in Section 4.3.1. The workaround is to disable TCP packet queue collapsing for tainted data.

4.5 Validation and VMM Stability

Throughout our implementation, we have been struck by how stable the resulting system has been. As the emulator and device extensions all run in user-space, we rarely needed to reboot the physical host during development. Moreover, the changes to the VMM itself have been very small, amounting to only 528 lines of code. We also change 755 lines of kernel code in the control VM, mostly to provide the driver extensions. All the remaining implementation described here resides in user-space.

One concern in our final implementation was to ensure that we were in fact emulating in all the appropriate situations. To validate this, we enabled debug single-stepping whenever an emulated fault was reissued by the virtualized protected VM. The use of single-stepping ensured that *only* the faulting instruction is executed by the VM—if the emulator has misjudged an instruction, single-stepping ensures that the virtualized execution returns immediately to Xen, where execution may be redirected to the emulator. This validation technique did in fact identify a fault-handling bug, which has now been resolved.

5. EVALUATION

In this section, we evaluate the performance of our taint-based protection system. We use a snapshot of the public Xen source code taken on December 16, 2005, post 3.0 release. The test machine is a Dell 2650, with a single 2.8GHz P4 hyperthreaded CPU, 2GB of RAM and Broadcom TG3 gigabit Ethernet interface. 1GB of RAM was allocated to the control VM and 128MB to the protected VM, running an Ubuntu Linux distribution.

The machine configurations of interest are:

- **(a)** Native Linux running on bare hardware (Linux)
- **(b)** Linux running in a virtual machine (Xen Linux)
- **(c)** Linux in a VM, modified to be *taint-friendly*, by reducing data copying, cleaning the registers and stack, and tuning the network stack (Modified VM)
- **(d)** Modified Linux running as a protected virtual machine, switching to emulation to track tainted data (Protected VM)
- **(e)** Modified Linux, always running in the emulator, without tainting (100% emulated, no taint)
- **(f)** Modified Linux, always running in the emulator, propagating taints (100% emulated, tainting)

5.1 Untainted Performance

LMbench is a series of micro-benchmarks designed to evaluate the performance of operating system primitives. We specifically select the measurements of the processor, processes, file system, and virtual memory system. Table 4 shows the performance of: the stat system call, catching signals, the fork system call, forking and then executing /bin/sh in the child, creating a zero size file, and mapping and unmapping a file into/from memory.

The first three rows show the times and relative slowdown for native Linux, our taint-friendly kernel, and a protected VM. The stat, signal handling, and file create tests show a negligible increase in time and hence little penalty for tainting as no network data is involved.

The extra time required for the fork system call and mmap latency can be attributed to shadow page table maintenance. It is interesting to note that in the fork shell benchmark, where the time to instantiate the shell process dominates the time to fork the process, the modified VM and protected VM both approximate the performance of Xen Linux.

To understand the overhead of our implementation, we have also run LMbench on two modified systems. In the first, the system stays in the QEMU emulator all the time but with taint tracking disabled. However, the overhead of switching be-

Configuration	max	min	average
baseline $\left(\frac{c}{a}\right)$	3.0x	1.1x	1.9x
taint overhead vs Linux $\left(\frac{d}{a}\right)$	5.0x	1.1x	2.1x
taint overhead vs emulator $\left(\frac{d}{c}\right)$	1.7x	1.0x	1.1x
emulation overhead $\left(\frac{e}{c}\right)$	35.7x	15.8x	26.6x
taint overhead $\left(\frac{f}{e}\right)$	3.9x	1.0x	2.2x

Table 5: Combined processor, processes, file, and VM system latency LMbench test results.

Configuration	netcat (MB/sec) transmit ($x = 2,000$)		netcat (MB/sec) receive ($x = 20,000$)	
Linux	107.19		107.04	
Xen Linux	87.41	(1.2x)	80.43	(1.3x)
Modified VM	86.92	(1.2x)	47.87	(2.2x)
Protected VM	72.04	(1.5x)	5.94	(18.0x)

Configuration	dd|ssh (MB/sec) transmit ($x = 200$)		dd|ssh (MB/sec) receive ($x = 100$)	
Linux	38.38		34.21	
Xen Linux	28.52	(1.3x)	27.44	(1.2x)
Modified VM	25.62	(1.5x)	17.11	(2.0x)
Protected VM	0.29	(132x)	0.22	(155x)

Table 6: netcat and dd|ssh test results

tween the emulator and Xen to handle interrupts and exceptions remains. The final row shows results for a similar system but with taint tracking enabled.

We next analyze each individual LMbench processor, process, file, and VM system test. For each micro-benchmark, we compute the ratio of the scores from two configurations from Table 4. We list in Table 5 the best, worst, and average ratios across all micro-benchmarks for the configuration pair. This allows us to estimate the performance impact of individual components.

The first row shows the overhead that our taint-friendly kernel, running on a virtual machine, incurs, compared to native Linux. The next row shows the overhead that our implementation incurs for these LMbench tests, as compared to the same native Linux. These numbers are generally very low, as we would not expect LMbench to introduce tainted data into the system.

The third row shows the overhead of our implementation versus a VM that is running our taint-friendly kernel and network driver, but without any emulator; there is no taint tracking, and no switching into emulation. In other words, we add about 10 percent total overhead to LMbench for running it on a virtual machine that is doing full system taint tracking.

The fourth row shows the impact of running under emulation. With the V→E→V overhead, we see that the emulator is about 26.6 times slower than the raw processor.

Finally, we compare running with and without taint tracking in the emulator. This shows that our taint tracking additions to QEMU incur an approximately 2.2x slowdown *when we are in the emulator*. However, the goal is to avoid entering the emulator as much as possible.

5.2 Network Performance

Network traffic is the source of taint introduction into the machine, and the next experiments determine the typical network transmit and receive rates of a protected virtual machine. The **netcat** command sends data through a TCP connection between machines, with the amount of data transmitted chosen to exercise the machine for approximately 30 seconds.

```
recvhost% nc -l -p 2000 > /dev/null
sendhost% (sleep 5; dd if=/dev/zero bs=100k
     count=x) | nc recvhost 2000 -q 0
```

In the results in Table 6, native Linux sees symmetric, near wire-limited transmission and reception speeds for netcat on our 1 gigabit Ethernet. Virtualization then introduces a 20 percent performance cost on network speeds due to the interposition of our control VM between the unprotected virtual machine and its network.

Receive speeds are reduced by adding our taint-friendly virtual NIC (described in Section 4.3.1) that is pre-screening and filtering packets, and then separating their headers into untainted memory so that the protected VM can process the headers without touching the tainted payloads. To limit the excess propagation of taints while executing the kernel we have also added register and memory scrubbing on every interrupt, system call, and exception; and we have disabled the TCP stack's buffer collapsing to avoid copying of tainted data.

The taint-protected virtual machine achieves transmission speeds close to that of the unprotected virtual machine, which is what we would hope for—since the majority of the incoming network packets will simply be packet ACKs which carry no payload, and thus introduce no new tainted data into our machine. On the receive path, however, we see that a taint-protected virtual machine experiences an 18x performance decrease, as the incoming tainted packet data must be copied into user-space, with all such copying necessarily performed inside the emulator.

To explore the expected worst-case performance of our system, we drive a CPU-bound computation on tainted data whilst also receiving interrupts by running **dd through ssh**.

```
recvhost% ssh sendhost dd if=/dev/zero
     bs=100k count=x > /dev/null
```

On native Linux, we see that the machine is CPU bound—it is unable to use all of the gigabit Ethernet bandwidth. The unprotected virtual machine again sees a performance decrease due to the additional layer between the VM and the network.

As predicted, this is an exceptionally bad workload for our taint-protected virtual machine. The initial key exchange between ssh and sshd deposits tainted data in user-space that is accessed to decrypt every arriving packet, so triggering emulation. In addition, the interrupt processing caused by packet

Configuration	V2E (cycles)	E2V (cycles)
1 CPU	39.6k	31.5k
1 CPU (with hyperthreading)	36.8k	24.0k
2 CPUs (without HT)	37.2k	34.0k

Table 7: Transition times from virtualization to emulation and vice versa.

arrival and system call handling for transferring data between the kernel and user-space causes the protected virtual machine to be continually switched between virtual and emulated environments, incurring expensive transition overheads and decreasing performance to a 150th of native execution.

5.3 Transition Costs

We measure the transition time taken to switch between running virtualized and running under the emulator by sampling the processor cycle counter either side of carefully constructed instruction sequences, deliberately touching tainted data (in the case of entry to emulation) or terminating a carefully placed basic block with untainted registers (to cause a return to virtualization). The results are shown in Table 7. These show the number of cycles on a 2.4GHz CPU occurring between two `rdtsc` instructions with three other instructions between them that trigger mode transition[2]. None of the lasting effects on the virtual machine (cache effects, TLB effects, etc) are accounted for here.

The transition results indicate that it is desirable to run the control VM and the protected VM on two different threads on a hyperthreaded processor. With our workload, hyperthreading enables both virtual machines to make progress in parallel where possible, successfully beating the uniprocessor alternative, and yet allows them to exchange the VM state on a transition in their shared caches, which reduces the time of a transition. In contrast, the SMP configuration has to move this processor configuration (and the VM's working set, which is not directly measured by these numbers) between CPU caches, slowing it down and outweighing the benefit of avoiding inter-VM cache pollution.

Determining when to return from emulation to virtualized execution is a subtle question deserving of further research. It would satisfy the correctness constraints of our model to simply exit emulation whenever all of the CPU registers are untainted. An artifact of our implementation is that it restricts switching to occur only at the end of basic blocks. The important question is whether it is *desirable* to switch back to virtualized execution at the soonest possible opportunity: while all the registers may be clean at a given instant, if they are about to be tainted again then we would rather avoid the rather large transition times from Table 7.

Towards this end, we added a basic hysteresis function: transitions from emulation would only be considered after N sequential memory accesses have all accessed untainted pages. (Notice that even if we are not touching tainted data, we remain in emulation if we are touching pages which contain tainted data, i.e. false taint hits.) Table 8 shows the effects of hysteresis settings of 5, 50, 500, and 5000 on our netcat receive workload.

[2]The observed difference in cycle counter values executing without performing a transition was typically 84 cycles, nearly all of which is just the cost of the `rdtsc` instructions themselves

hysteresis	MB/sec
5	5.93
50	5.79
500	4.01
5000	3.24

Table 8: Effect of hysteresis on netcat receive

In this experiment, tainted data is being received in bulk by netcat and immediately discarded into `/dev/null`. All the copy_to_user calls in the kernel are touching tainted data, and thus must occur in emulation. In this particular benchmark, we see that increasing hysteresis does not provide a benefit; however we have observed counterexamples during development and believe that future research ought consider a dynamic control system for the hysteresis value.

6. CONCLUSION

This paper presents a novel architecture for taint-based data protection using a combination of machine virtualization and emulation, safely achieving high-performance for computation by dynamically switching to emulated execution only when tainted data is being handled by the processor.

6.1 Summary

When executing in a taint-free environment, we are able to drastically improve upon the overhead of continuous emulation and, in effect, introduce a new hardware feature with reasonable overhead on commodity hardware. Although our work extends previous research on taint-based data tracking, our approach is both more efficient and more comprehensive: to our knowledge we are the first to propagate taint markings to and from disk, ensuring that tainted data may never be executed.

We have identified the importance of introducing taint-friendly modifications to an operating system kernel, to reduce the window of tainted execution and limit the propagation of taints. Adding register and stack washing to interrupt handler code prevents the execution of tainted user processes from unnecessarily forcing the kernel into the emulator.

Exporting the address space of a running virtual machine into a user-space process in a second virtual machine to perform system emulation proved to be a convenient and powerful technique, although ultimately at the expense of some performance due to the high transition costs incurred. We believe that implementing emulation within the hypervisor would provide a significant system performance improvement.

6.2 Ongoing and Future Work

The current taint-tracking implementation within our modified QEMU required the explicit addition of taint logic to the execution of each micro-operation. Instead of adding taint code manually to the almost 1,000 micro-operations used by QEMU, we are working on using the CIL [28] C-to-C transformation tool to add taint annotations automatically. We anticipate that this will greatly simplify experimentation with alternate taint models, such as byte-level tainting within each register.

A second piece of ongoing work targets additional performance enhancements which go beyond the simple per-micro-operation taint tracking. Instead we are investigating the application of run-time data-flow analysis to "collapse" a sequence of tainting operations. Ideally this will be integrated closely with the basic-block optimization and chaining techniques used by QEMU, allowing near-native execution of emulated code while maintaining accurate taint semantics.

We decided to use an emulator running in user-space in the control VM to simplify the system architecture. In retrospect, transition costs between the virtualization and emulation and the need to "double bounce" to service a system call or fault during emulation was greater than what we originally anticipated. Placing the emulator within the VMM and aligning the data structures used by the emulator with the VMM's would drastically minimize these costs.

The basic hardware support assume in our work so far has been page-granularity protection. However some researchers [29, 30] have explored finer granularity hardware protection techniques in which the ECC bits are manipulated in order to reliably protect individual cache lines. We anticipate that using such techniques would greatly improve performance by reducing the amount of false tainting.

We started this work, hoping to avoid any changes to the kernel or application code, but found that the changes in Section 4.4 were required to combat the accidental leaking of tainted data into many kernel operations. We would like to explore explicitly exposing the operating system and the application writer to the notion of running on a tainted system, and see if various forms of taint segregation can be improved cooperatively. For example, one can imagine a taint-aware garbage collector which keeps tainted objects on tainted pages, thus trying to minimize false taints.

The approach of dynamic emulation presented in this paper has allowed us to implement a sweeping new architectural feature—in this case taint-based protection—in a novel fashion. We feel that this is in some sense just one application of demand emulation: the approach is more generally applicable for any "sometimes on" architectural features, where performance can be regained by reverting to virtualized execution whenever a feature is not in use.

In addition to the requirement that new features be "sometimes on" for this approach to be useful, we require that there exist an appropriate mechanism (such as page protection) to activate the switch from virtualized to emulated execution. Given these constraints, we are confident that there is a considerable opportunity to explore the development of other new hardware features using this technique, and look forward to examining this area in the near future.

Acknowledgments

The work presented in this paper benefited greatly from discussions with Ian Pratt, Keir Fraser, Jon Crowcroft, Periklis Akritidis, Anil Madhavapeddy, Manuel Costa, and Paul Barham. The authors would also like to thank our shepherd Gilles Muller, and Fabrice Bellard for his excellent work in creating a versatile open-source emulator that proved both enjoyable and educational to extend.

7. REFERENCES

[1] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[2] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[3] Wei Xu, Sandeep Bhatkar, , and R. Sekar. A Unified Approach for Preventing Attacks Exploiting a Range of Software Vulnerabilities. Technical Report Technical Report SECLAB-05-05, Department of Computer Science, Stony Brook University, August 2005.

[4] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, pages 85–96, 2004.

[5] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[6] Shuo Chen, Jun Xu, Nithin Nakka, Abigniew Kalbarczyk, and Ravi Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN-2005)*, June 2005.

[7] Dana Madsen. An Operating System Analog to the Perl Data Tainting Functionality. In *Proceedings of the 23rd National Information Systems Security Conference*, June 2000.

[8] Randal L. Schwartz. Perl Advisor: Taint so Easy, Is It? *Unix Review*, August 2000.

[9] David Thomas and Andrew Hung. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison Wesley Longman, first edition, 2001.

[10] David Flannagan. *JavaScript: The Definitive Guide*. O'Reilly, second edition, January 2001.

[11] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeffrey Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference (SEC2005)*, May 2005.

[12] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, pages 321–336, August 2004.

[13] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex Snoeren, Geoff Voelker, and Stefan Savage. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[14] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[15] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. *Argos*: an Emulator for Fingerprinting Zero-Day Attacks. In *Proceedings of the first EuroSys Conference*, April 2006.

[16] Pax project. `http://pax.pgsecurity.com/`.

[17] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 143–156, October 1997.

[18] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.

[19] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[20] Paul Starzetz. Quick Analysiss [sic] of the recent crc32 ssh(d) bug. Email to bugtraq@securityfocus.com, February 2001.

[21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[22] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, April 2005.

[23] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI 2002: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[24] P. H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[25] Judith S. Hall and Paul T. Robinson. Virtualizing the VAX Architecture. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 380–389, New York, NY, 1991.

[26] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, May 2005.

[27] Andrew Warfield, Steven Hand, Keir Fraser, and Tim Deegan. Facilitating the Development of Soft Devices. In *Proceedings of the 2005 USENIX Annual Technical Conference*, April 2005.

[28] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. Cil: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th Annual Conference on Compiler Construction*, April 2002.

[29] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Proceedings of the 2nd USENIX Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[30] Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, February 2005.