Virtual Machine-Provided Context Sensitive Page Mappings

Nathan E. Rosenblum Gregory Cooksey Barton P. Miller
Computer Sciences Department, University of Wisconsin–Madison
{nater,cooksey,bart}@cs.wisc.edu

Abstract

Context sensitive page mappings provide different mappings from virtual addresses to physical page frames depending on whether a memory reference occurs in a data or instruction context. Such differences can be used to modify the behavior of programs that reference their executable code in a data context. Previous work has demonstrated several applications of context sensitive page mappings, including protection against buffer-overrun attacks and circumvention of self-checksumming codes. We extend context sensitive page mappings to the virtual machine monitor, allowing operation independent of the guest operating system. Our technique takes advantage of the VMM's role in enforcing protection between guest operating systems to interpose on guest OS memory management operations and selectively introduce context sensitive page mappings.

In this paper, we describe extensions to the Xen hypervisor that support context sensitive page mappings in unmodified guest operating systems. We demonstrate the utility of our technique in a case study by instrumenting and modifying self-checksumming tamper-resistant binaries. We further demonstrate that context sensitive page mappings can be provided by the VMM without incurring extensive overhead. Our measurements indicate only minor performance penalties stem from use of this technique. We suggest several further applications of VMM-provided context sensitive page mappings, including OS hardening and protection of processes from malicious applications.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.4.6 [*Operating Systems*]: Security and Protection

General Terms Design, Security

Keywords context sensitive memory, virtual machine monitor, xen, self checksumming code

1. Introduction

Tamper-resistent software seeks to defend against modification of its behavior by an external agent. One technique for providing this resistance is *self-checksumming code*. A self-checksumming program incorporates computations whose results depend on contents of memory representing the program's code. If the code has been modified (for example, by an instrumentation program as part of a reverse-engineering effort), the program will detect the modification and can take appropriate defensive action, such as terminating execution. Surveys of malicious software such as viruses have revealed that authors frequently disguise their applications with commodity *packing* software, some of which employs self-checksumming as a tamper resistance mechanism [10]. Circumventing such protections is an important first step in analyzing novel malicious software.

Self-checksumming techniques rely on the underlying hardware having a *von Neumann memory architecture* [15], in which the program data and instructions reside in the same storage object. This unified view of memory enables the application to access its executable code as a data reference, enabling the self-checksumming technique. Modern operating systems such as Microsoft Windows and Linux on the x86 architecture export this unified view of a program's address space. Previous work by Wurster, et al. [16] subverted self-checksumming codes by modifying the kernels of Linux and Windows operating systems to induce a virtual *Harvard memory architecture* from the target program's perspective, as depicted in Figure 1.

In this paper, we introduce virtual machine monitor-based *context sensitive page mappings*. The aim of our work is to transparently manipulate the guest operating system's view of memory, and by extension that of processes within the guest OS. Context sensitive page mappings are a technique to introduce different mappings from virtual to physical addresses at page granularity based on the type of memory reference. The virtual machine monitor is an ideal point to introduce context sensitive mappings, because of its central role in managing virtual memory hardware to enforce protection. Our modified VMM, along with a user-level utility process, allow us to instrument and examine a tamper-resistant process running on an unmodified operating system.

Our implementation extends the Xen hypervisor [2] on the Intel x86 platform to enable selective introduction of context sensitive page mappings into unmodified guest operating systems. Careful manipulation of the *translation lookaside buffers (TLBs)* can introduce virtual to physical address translation that is sensitive to data or instruction execution context. Implementing this facility in a virtualized context introduces several challenges that are not present when similar techniques are applied at the operating system level, such as target process identification. In addition, the x86 architecture does not provide software managed TLBs, so a well-defined interface does not exist to manipulate their contents. We discuss our approaches to these challenges in Section 3 and give implementation details in Section 4.

Although our implementation focuses on the Xen hypervisor running on the x86 architecture, the general technique is not limited to a particular virtual machine monitor or hardware platform. VMM-provided context sensitive page mappings have only two prerequisites: a mechanism for interposition between the operating system and the virtual memory hardware, and a hardware facility that can provide alternate mappings from virtual to physical ad-

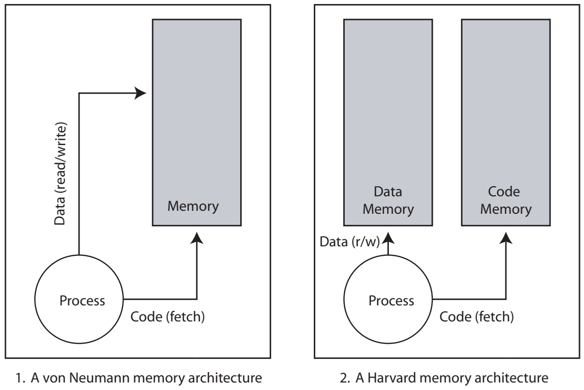


Figure 1. Two possible application views of memory. In (1), the process sees the same address spaces regardless of context (instruction vs. data). In (2), separate address spaces exist depending on context. Self-checksumming code implicitly assumes the model of (1).

dresses based on the context of a memory reference. Architectures such as SPARC and Power that, like x86, have data and instruction TLBs should be suitable for this technique. Indeed, platforms with software managed TLBs (e.g., SPARC) simplify such an implementation.

In addition to operating system independence, extension of context sensitive page mappings to the virtual machine monitor suggests additional applications. Because the context sensitive mappings are provided by the VMM, they remain transparent to privilege processes within the guest operating system. We suggest two uses of this fact in Section 7: protection of sensitive guest OS data and code, and protection of particular processes within the guest OS.

2. Related Work

Our technique for inducing context sensitive page mappings follows the direction used by Wurster et al. [16, 14] to attack self-checksumming codes. The authors presented two methods for inducing a virtual Harvard memory architecture from point of view of Windows and Linux processes on the x86 architecture, both involving manipulation of virtual memory-related hardware. Wurster et al. describe a manipulation of the paging hardware by modifying the operating system virtual memory management code.

Sparks and Butler [12] present a method similar to that of Wurster to induce a split code and data view of memory on the Intel x86 architecture. The authors describe *Shadow Walker*, a stealth rootkit that patches the page fault handling routines in the Windows kernel to hide or misrepresent the executable contents of malicious programs [9].

The PaX project [11] is an extension of the Linux kernel that attempts to reduce certain types of security exploits by randomizing the location of a program's layout in memory, including the placement of non-position-independent code. PaX redirects accesses of the expected code location to its new location. Unlike the technique we present here, PaX makes no attempt to obscure this mapping.

Garfinkel and Rosenblum [5] introduced a VMM-based intrusion detection system. They describe a *kernel memory enforcer* component that protects sensitive portions of the guest OS kernel from tampering, even by software running at the same privilege level as the OS kernel. Our context sensitive page mappings can provide similar protection against malicious privileged malicious software as we describe in Section 7.

Giffin, et al. [6] addressed Wurster's attack by extending the self-checksumming protection to include *self-modifying code*. By requiring that the anti-tampering test compute a value contingent on consistent execution and reads of program code, the authors eliminate the implicit reliance on a von Neumann architecture. This defense depends on a program's ability to self-modify, which may not be possible. For example, self-modifying code is not available in environments that provide and rely upon hardware protection against code injection (e.g., `NX` bit protection against execution on the heap [13]).

3. System Overview

Our system provides context sensitive views of memory at page granularity within guest operating systems in a virtualized environment. Our goal is to provide such mappings in an efficient manner, while ensuring transparency from the perspective of the guest operating system and processes running within it. Implementation of this feature relies on certain fundamental characteristics of the underlying hardware. In particular, the virtual memory hardware must provide mapping mechanisms that are sensitive to memory reference context. There must be a mechanism (explicit or implicit) to manipulate the contents of the virtual memory hardware to effect a disparity between execution and data reference views.

In Wurster's [16] approach to circumvention of self-checksumming code, a modified operating system managed the paging hadware manipulation. Because our context sensitive page mappings are provided by the VMM, we must be able to interpose on operating system activity that affects the virtual memory hardware. In designing modifications to the Xen virtual machine monitor to support our requirements, we had to address the following issues:

- VMM interposition and control of OS paging operations
- Separate manipulation of instruction and data TLBs
- Storage of extra pages needed to provide context sensitive data
- Identification of process access to context sensitive pages

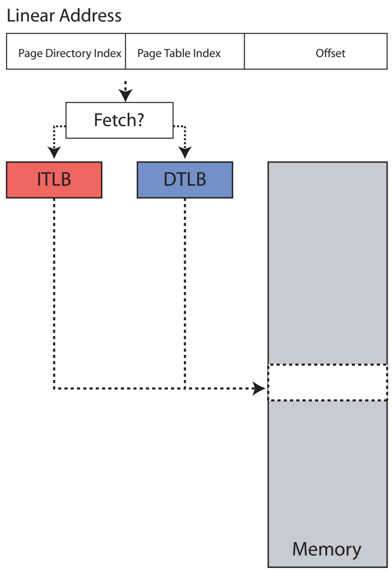
We provide background on our implementation environment and give an overview of our approach to each of these challenges in this section.

3.1 MMU Update Interposition

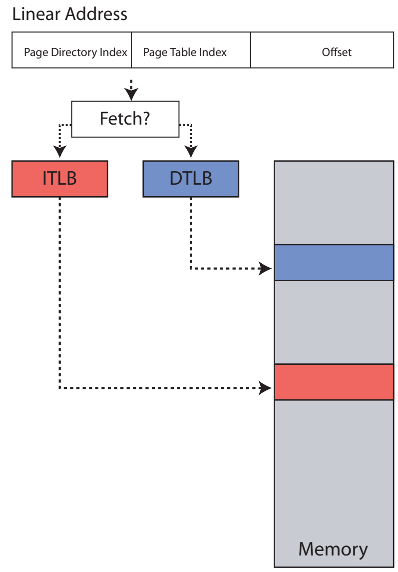
Our implementation comprises modifications of the Xen virtualization environment [2] on the Intel x86 architecture. The Xen virtual machine monitor hosts multiple guest operating systems as *guest domains*. One privileged domain (*dom0*) interacts with the hardware directly and is tasked with creation and management of the other guest domains (*domU*s). The virtual machine monitor implements protection in the x86 memory management unit. The VMM virtualizes privileged MMU operations, such as updates to the page tables that provide a virtual to physical address mapping. When the operating system attempts such modifications, the VMM interposes on the operation and ensures that inter-domain protection is not being violated. It is this low-level control of virtual memory hardware that enables our context sensitive page mapping technique. We can control guest OS modifications to the page tables because the Xen hypervisor must already manage guest access to to the virtual memory hardware.

3.2 Virtual Memory Subsystem

Our mechanism for implementing context sensitive page mappings is depicted in Figure 2. The paging hardware on the x86 platform translates from segmented linear addresses to physical addresses through a *page table*. This mechanism allows for sharing and protection of the linear address space at the granularity of a page, usu-



(a) Normal linear address space



(b) Split address space

Figure 2. An address space split induced by desynchronized instruction and data TLBs. In (a) the ITLB and DTLB entries indexed by the first 20 bits of the linear address are identical. After application of our technique, the entries are desynchronized and index different physical pages, as in (b).

ally 4096 bytes. Each process has associated with it a *page directory* pointed to by the *page table base register* `cr3` that represents a two or greater level tree of *page tables* identifying pages owned by that process. When a reference to a virtual address is executed by the processor, the hardware memory management unit uses the page tables to translate from the virtual page to the physical page frame containing the referenced data.

To reduce the cost of memory references, the *page table entries (PTEs)* for recently accessed pages are cached in a *translation lookaside buffer*. Only when an address translation cannot be satisfied from the TLB is a full page directory traversal necessary. In the case of a TLB miss, the hardware performs a walk over the page directory and fills in the missing TLB entry with no software intervention. The x86, like several architectures, has two separate TLBs: one for data references (the DTLB), and one for instruction references (the ITLB). This contextual distinction at the lowest levels of the paging hardware provides a handle for controlling the view of memory exposed to a program.

Normally the ITLB and DTLB entries mapped to a particular virtual address are identical. By carefully manipulating the processor state we are able to break this symmetry, causing translations for a particular virtual address to depend on context. This mechanism depends on our ability to manipulate the contents of the instruction and data TLBs independently, a feature not directly supported by the x86 architecture. Instead, we use a combination of page table modifications and specially crafted memory references to load differing PTEs into the ITLB and DTLB. The VMM uses the virtual address space of the target process to make these memory references. This ensures that the loaded TLB entries are used to resolve subsequent memory references by the target. Details of these techniques are discussed in Section 4.

3.3 Storage Alternatives

Providing context sensitive data raises the question of where that data should be stored. It is likely too expensive in both time and space to provide context sensitive mappings for all processes, let alone all pages in a particular process. Instead, the set of mappings are represented as tuples $(V_{cs}, P_{data}, P_{exec})$, where V_{cs} is the virtual address of the context sensitive page and each of P_{data} and P_{exec} corresponds to physical memory visible in a data or instruction context, respectively. We refer to such a tuple as a *page correspondence*. One physical page in the correspondence represents the original page that is rendered context sensitive by our technique. The other physical page must be allocated and managed separately. The page can either be allocated within the guest operating system, or by the VMM itself.

Assume that P_{exec} corresponds to the original page allocated within the guest operating system, and that P_{data} must be allocated within the guest OS or the VMM. Wherever P_{data} is allocated, data-context access to locations within V_{cs} must reference the correct data in P_{data}. This presents particular challenges if P_{data} is allocated within the guest OS. Recall that the mapping from V_{cs} to P_{date} is controlled by the VMM; in particular, the guest operating system has no knowledge of it. If the guest OS has control over P_{data}, it can chose to reclaim that memory at any time by writing the contents of P_{data} to the backing store and loading new values in its place. The guest would maintain a mapping from another virtual address, V_{alt}, to P_{data}, and would not invalidate the V_{cs} mapping when it released P_{data}. Our VMM has no way of influencing the guest operating system's behavior in this regard. We cannot guarantee that the contents of a P_{exec} page allocated within the guest OS remain valid.

Instead we allocate storage within the virtual machine monitor's address space. The VMM ensures that the contents of P_{data} are available for the lifetime of the page correspondence. This method increases system resources reserved by the VMM, but ensures that guest operating system behavior cannot affect context sensitive page mappings.

3.4 Target Process Identification

The principal remaining challenge is lack of information about guest operating system data structures, in particular those pertaining to the target process. Depending on availability of low-level documentation, guest OS data structures might or might not be opaque to us, but in either case we have no direct way to associate that information with a particular process. The VMM does not have access to guest operating system abstractions such as process identifiers with which to select target processes. Also, dependence on such OS internal information would reduce portability across guest operating systems and across versions of the same system.

One way the VMM could identify individual processes is by address space identifiers (ASIDs) that uniquely identify a process in both time and space. Although x86 architecture lacks the notion of a proper ASID, our technique only requires a means to uniquely identify a process at a particular moment in time. It is irrelevant in our system whether a particular identifier is reused by multiple processes, so long as only one process (or group of processes sharing an address space) uses it at a given time. The x86 page table base register (`cr3`) provides just such semantics. Each unique address space has its own set of page tables, pointed to by `cr3` while the process owning that address space is executing.

4. Implementation Details

Our modifications to Xen include three main components: installation of context sensitive mappings for target processes, interposition on operating system access to the memory management hardware, and interface extensions to selectively enable context sensitive page mappings for particular pages in an address space.

4.1 Installation of Mappings

Installation of context sensitive page mappings for a target process is divided into two parts: intercepting access to context sensitive pages for which mappings have not been installed, and installing the mappings. The former involves straightforward use of the x86 memory management hardware, while the latter is complicated by the lack of software programmable TLBs. Both are implemented in the page fault handling routines of the Xen hypervisor.

Processor faults on x86 are passed to software running at the highest privilege level, in our case the VMM. Page faults are generated when memory accesses fail for several reasons, such as protection violations or *page not present* conditions. Except when the fault is caused by inter-domain protection violations, the VMM refers page faults to the guest operating system for service. When we create a context sensitive page, we update the corresponding page table entry and clear the User/Supervisor bit. Unprivileged access to a page with a clear User/Supervisor bit causes a page fault due to protection violation.

We modified the page fault handling routine on Xen to test whether such page faults are caused by access to a context sensitive page. Our VMM maintains a hash of all page correspondence tuples associated with particular address space. A hash lookup determines whether the faulting process was accessing a context sensitive page. If so, the VMM must install a context-sensitive page mapping in the faulting process' TLBs. The only complication arises if the page in question is also marked as not present. In this case, the VMM depends on the guest operating system to retrieve the page from the backing store. We modify the page fault exception code to indicate that the fault was caused by a *page not present* condition and refer the fault to the guest OS. A repeat memory reference will cause another page fault and the VMM will then install the context sensitive mapping.

The steps necessary to install different mappings in the x86 instruction and data TLBs are outlined in Figure 3. The basic strategy is to modify the page table entries and then execute and read code from the page, loading the modified page table entries in the the appropriate TLBs. We modify the instruction context page to allow safe execution by the VMM by inserting a jump instruction that returns control to our page fault handler.

Wurster et al. appear to take a similar approach to installing different ITLB and DTLB mappings on the x86 architecture [14]. They indicate that their VMM executes a `NOP` (no operation) instruction in a target page to load the mapping into the ITLB. It is not clear from their description how control would be transferred to this `NOP` instruction or returned to the VMM after executing it. We surmise that they use a sequence of jump intsructions similar to the technique that we describe below.

First, the page table entry for the context sensitive page is updated to temporarily unprotect the page by setting the User/Supervisor bit (line 5). The ITLB is loaded whenever an instruction context memory reference is made to a virtual address and the corresponding PTE is not already cached in the ITLB. We can safely execute an instruction out of a page in the target process if we control its contents. However, there are two complications: writing instructions to a virtual address causes the DTLB to be loaded, and entries cannot be flushed from the DTLB independent of the ITLB.

To modify the target page's contents, we first create a new virtual to physical mapping in the VMM's address space (line 10). We then flush the target entry from the TLBs and install a *trampoline* into the target physical page through the VMM's virtual mapping (line 19). The trampoline is simply a branch instruction that jumps back to the `return_loc` address in our pseudo code representation. Finally, we branch to the trampoline code through the target virtual address (line 24), causing the ITLB to be loaded with the PTE we have temporarily unprotected.

Installation of the DTLB entry is considerably easier. First, we modify the the page table entry to point to a different physical page (P_{data}) (line 32). The VMM then reads a single byte of memory from the target virtual address, loading the DTLB. Our final step is to restore the page table entry to its original, protected state (line 40). The ITLB and DTLB mappings for the target virtual page are now desynchronized.

These mappings persist until the guest operating system schedules a different process or explicitly flushes the TLBs (e.g., when it updates page tables). Subsequent accesses to the the context sensitive page while these mappings exist will not incur additional page faults because the User/Supervisor bit is set in the PTEs cached in the TLBs.

4.2 OS MMU Interposition

We are able to modify and maintain modified page table entries because the VMM is responsible for enforcing protection between domains by use of the hardware memory management unit. Because the MMU traverses the page tables when translating from virtual to physical addresses, the Xen hypervisor must restrict updates to the page tables to ensure inter-domain protection. When the guest operating system attempts to modify page table entries, the VMM intervenes to check the validity of updates. We extend the checking routines to ensure that PTEs for context sensitive pages are not modified by guest OS updates.

When the guest OS installs or modifies a page table entry, we test whether it is one of the page correspondences for the updated address space. If so, we install or modify the page table entry as requested, first explicitly clearing the User/Supervisor bit as described above. In this way we ensure that subsequent access by the target process will continue to cause page faults.

4.3 VMM Interface Extensions

We assume that there will be a process running on the guest operating system that will control the context sensitive page mapping facilities provided by our VMM. For example, the case study we present in Section 5 features a dynamic instrumentation program that requests context sensitive page mappings from the VMM to circumvent self-checksumming. Implementing such an application as

```
// Temporarily unprotect PTE entry by
// setting User/Supervisor bit
pte = get_guest_pte(addr)
orig = pte
set_pte_bit(pte, U/S)

// Create a new virtual to physical in
// mapping in VMM space
paddr' = physical_addr(addr)
addr' = map_domain_page(paddr')

// Flush the TLB entry for this virtual
// address only
flush_tlb_one(addr)

// Write a branch back to return_loc
// into the physical page through the
// VMM-space mapping
install_trampoline(addr')

// Branch back to the installed code
// through the guest-space mapping
// (installs ITLB entry)
jump(addr)

:return_loc
// remove changes
remove_trampoline(addr')

// prepare PTE for DTLB (change
// physical page)
modify_pte(pte, clean_page)

// install DTLB entry by reading from
// physical page with guest-space
// mapping
read_byte(addr)

// restore
pte = orig
```

Figure 3. Installation of split ITLB/DLTB entries. First the ITLB is loaded with the address of the modified page (the page pointed to by the protected PTE in our system), then the DTLB is loaded with a modified PTE pointing to a different physical page.

a process running on the guest operating system has the advantage that it reduces the number of changes needed in the Xen hypervisor. It is significantly easier to implement tools as normal processes rather than subsystems of a operating system kernel or virtual machine monitor. To support such applications, we define an interface between the guest process and the hypervisor.

We define four interfaces to the hypervisor. The first two install and remove context sensitive page mappings. The user-level process provides an address space identifier that names the target process, and a virtual address that specifies the context sensitive page. They hypervisor is responsible for allocating a page within its pool of memory to represent the P_{data} member of the resulting page correspondence tuple.

- *create_cs_mapping(INPUT asid, INPUT vaddr)*
  Requests that the hypervisor create a copy of the page corresponding to the input virtual address in the given address space, and to maintain a context sensitive page mapping.
- *remove_cs_mapping(INPUT asid, INPUT vaddr)*
  Requests destruction of the indicated context sensitive page mapping.

For the purposes of the tamper-resistance circumvention we describe in Section 5, the P_{data} page stored by the hypervisor need only be a copy of the original page specified in the *create_cs_mapping* call. In general, it may be useful to modify the contents of the P_{data} page after installation of the context sensitive page mapping.

- *update_cs_page(INPUT asid, INPUT vaddr, INPUT vupdate)*
  Instructs the hypervisor to update the P_{data} member of the specified mapping by copying the contents of the page indicated by `vupdate`.

The final interface stems from our use of the `cr3` register to identify target processes and is particular to our specific needs in the case study described below. Access to the `cr3` value is restricted to privileged processes (including the guest operating system), so a strictly user-space process cannot read the contents of the register. Hypervisor assistance is necessary to retrieve the value associated with a target process.

- *get_cr3(OUTPUT asid)*
  Returns the `cr3` value associated with the calling process.

Our use of this interface in our case study requires some explanation. The value of interest is only contained in `cr3` when the target process is executing. We must induce the target process to make the *get_cr3* call itself, prior to enabling the context sensitive page mappings that allow us to modify the tamper resistant process. We avoid this apparent contradiction by forcing the target process to execute the*get_cr3* call with no chance of its own code executing in the interim. Self-checksumming can only detect modifications to a program's code that persist while the self-checksumming algorithm runs. Our dynamic instrumentation tools enable us to halt an executing process, inject and execute arbitrary code, and remove that code prior to resuming normal execution of the target process. This technique gives us the ability to induce execution of the *get_cr3* call without detection. We are now free to modify the code of the target process. We discuss this and other details of our instrumentation tools in the following section.

5. Case Study: Self-Checksumming Code

To validate our implementation of context sensitive page mappings, we performed a case study in which we added instrumentation to a program that uses self-checksumming to detect modification to its code. In this section, we provide background on Dyninst, the instrumentation library we use to modify the target process. We then introduce the target tamper-resistant program and the program used to instrument it. Our instrumentation program runs as a user-level process within a Linux guest operation system, communicating target pages to the modified Xen hypervisor. The individual components of our system are depicted in Figure 4.

5.1 Dyninst Background

Dyninst is a cross-platform dynamic instrumentation and modification library [7, 3]. Dyninst differs significantly from other instrumentation systems. It creates or attaches to a running process, statically analyzes the binary, and then *dynamically modifies* the program as it executes. Dyninst's instrumentation is patched into the executable code of the program as necessary. For example, to obtain a trace of program execution, Dyninst can be used to patch the entry and exit points of all functions in the binary. When the program is allowed to run, it executes as normal, but the inserted instrumentation is executed whenever it is encountered. This method of instrumentation is flexible and incurs extremely low cost, but is clearly visible to introspective techniques such as self-checksumming code.

Dyninst provides another facility we have previously mentioned. Dyninst's ability to insert and remove code, combined with its control of process execution, facilitates *one-time codes*. A one-time code in Dyninst is transient; after execution it is removed, and no lingering effects remain in the binary. When Dyninst injects a one-time code into a process, it first halts the process if it is currently executing. It then saves a backup copy of instructions to be overwritten and installs the instructions that make up the one-time code snippet at the current point of execution in the process and continues the process execution. The one-time code saves the current process state before performing its intended action, and restores the state after it has completed. It then returns control to Dyninst, which removes the one-time code and restores the original instructions. The process is then continued from where it was originally halted with nothing to indicate that it been forced to execute some arbitrary amount of code. We previously described how this technique allows us to briefly hijack a tamper-resistant process to execute the *get_cr3* routine and remain undetected.

5.2 Instrumentation Program

We introduce a helper tool, Igor, that is responsible for loading and instrumenting the target process and communicating as necessary with the modified Xen hypervisor. Igor uses Dyninst to insert instrumentation into the target process at runtime. Without the context sensitive page mappings provided by the VMM, the instrumentation Igor inserts would be identified by the tamper-resistance mechanisms of the target process.

Igor starts the target application and immediately suspends its execution before the `main` function is executed. We use the Dyninst library to statically parse the target executable, identifying functions within its code section. Our goal is to instrument the entry and exit points of all functions in the binary, similar to how an analyst might go about tracing execution of a suspect process. Before any modifications are made to the target process, Igor communicates the set of affected pages to the Xen hypervisor using our modified interface. The hypervisor copies the contents of the target pages and creates page correspondence tuples for the target process. Once Igor has inserted the desired instrumentation, it releases the target process. Subsequent memory references within the target to modified pages are sensitive to context.

5.3 Tamper-resistent Target

Our target application is a simple proof-of-concept that uses self-checksumming to test whether its code has been modified. The application computes a checksum over the bytes comprising its `main` function. If the computed value differs from the expected value, the process terminates with an error condition. This is similar to the type of tamper-resistance provided by some software packers such as ASProtect [1].

We verify that our instrumentation is successfully applied to the target application by modifying its behavior. The `main` function invokes a routine that prints a known value to `stdout`. Another function exists in the binary that prints an alternate value to `stdout` but is never called. In addition to the entry and exit instrumentation we insert into the process, we dynamically modify the call in `main` to refer to the second function. Successful termination of the test program along with display of the alternate value verifies that the instrumentation was installed but remained undetected by the self-checksumming routine.

6. Performance Impact

We performed several micro benchmarks to ascertain the performance impact of our modifications to the Xen system. Our modifications add additional cost in two facilities of the Xen VMM: memory management unit updates and page fault handling. We were interested in two possible changes in these facilities: addition time consumed and increase in incidents of usage. In particular, we were interested in measuring the increase in page faults due to context sensitive page mappings.

Our system incurs at most one additional page fault per access to a modified page when the ITLB or DTLB does not cache the corresponding page table entry. Since the TLBs are flushed at context switch, the least number of additional page faults we incur is one per execution time slice. However, because collisions are possible in the ITLB and DTLB, cache interactions may increase the likelihood that our modified entries are ejected, necessitating an additional page fault on the next access.

We instrumented the VMM to record the time spent in these facilities and to count the number of times they were used. We took measurements for target programs with and without modified pages. The most significant increase is in handling page faults on modified pages, due to the additional operations necessary to cause the DTLB and ITLB to be loaded. All other overhead stems from checking whether the faulting process was referencing a context sensitive page. The overhead in performing MMU updates similarly stems from checking whether the updated or installed PTE refers to an element of a page correspondence tuple. The results of our micro benchmarks of the MMU update and page fault handling facilities in unmodified Xen and our extended VMM are summarized in Table 1.

	MMU Updates (ns per update)	Page Fault Handling (ns per fault)
Unmodified Xen	903	395
Normal Pages	974 (+7.9%)	447 (+13.2%)
Context Sensitive Pages	1,003 (10.9%)	918 (+132.4%)

Table 1. Direct VMM overhead due to the context sensitive page mappings extension. The percentage for normal and context sensitive pages indicates the increase over unmodified Xen. The highest cost is in servicing page faults on context sensitive pages, as it includes the cost of manipulating the TLBs.

We created three simple artificial workloads to help evaluate the performance of our context sensitive page mappings. The workloads exercise the additional layers of abstraction we introduce with our TLB manipulations. Our goal is to model the impact that context sensitive page mappings would have on code that frequently accesses a data store in multiple contexts, such as the self checksumming code from our case study. The test program maps a variable number of full-page functions into its address space. Using our modified hypervisor, we create context sensitive mappings that redirect read requests to different pages with known values that would abnormal execution if interpreted as instructions. The read and execute accesses in our workloads both target the original, executable pages. We verify correctness by checking for the alternate values read from the mapped in pages.

The three workloads represent a spectrum of load for our system. The executable pages consist primarily of `NOP` instructions that are represented by a single byte in the IA-32 instruction set. Our read function similarly accesses the page one byte at a time. Each was run with a varying number of context sensitive page mappings with the particular access pattern repeated 1,000 times.

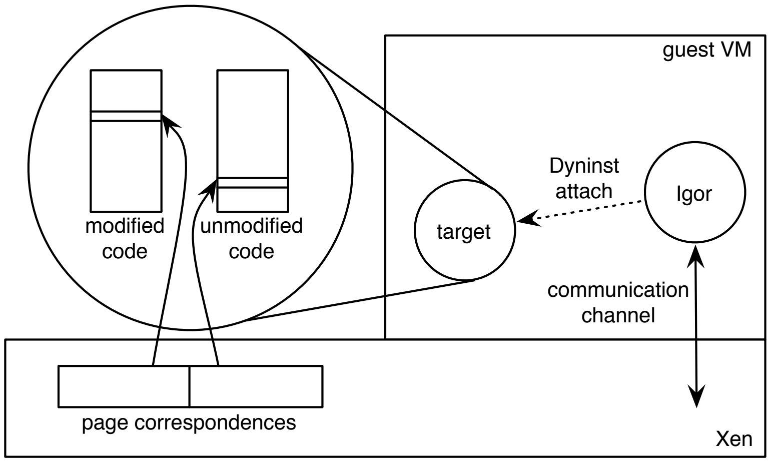


Figure 4. System component overview: the helper process Igor loads and modifies the target program, requesting context sensitive page mappings for particular pages from the Xen hypervisor. Then Igor modifies pages within the target process address space. Xen redirects memory references by the target process depending on context.

- *Page-interleaved access*
  This workload executes the contents of a page, then reads the contents. This pattern is very basic but does map to some realistic applications, for example just in time compilation or integrity self-checking. Figure 5 compares this and the other workloads to comparative workloads without context sensitive page mappings as the number of mapped pages is increased.
- *Completely serial access*
  All of the pages are executed, then all are read. This pattern of access performs slightly less well than the page-interleved access, mainly due to the increased number of page faults. By linearly traversing a large number of pages we cause a large number of collisions in the TLB. We are forced in this case to take a page fault and re-apply the context sensitive mapping in the TLBs.
- *Fine-grain interleaved access*
  We assumed that this access pattern would paced the most stress on our system and it most closely resembles the real access patterns of self-checksumming or otherwise introspective code. The shorter run times are in part due to minor configuration differences in our test application, but the good performance overall nicely demonstrates the efficacy of the dual TLB mappings. Without the ability to load arbitrary values independently into the instruction and data TLBs, each fine-grain interleved access would cause a page fault and greatly increase the cost of creating the context sensitive view of code and data.

7. Other Applications of Context Sensitivity

Implementation of context sensitive page mappings in the virtual machine monitor instead of natively at the operating system level means that such mappings can be transparent to privileged process running in a guest operating system. In this section we describe two potential uses of this technique to defend against malicious software.

7.1 OS Tamper Resistance

Stealthy malicious applications such as *rootkits* attempt not only to subvert operating system control, but also to conceal themselves from detection [4] . One common method of hiding from system introspection utilities is to patch system call entry points. Custom routines are inserted that mask the malicious process, for example by modifying the output of a task listing to remove telltale entries. Detecting processes hidden in this manner has been a topic of recent research [8].

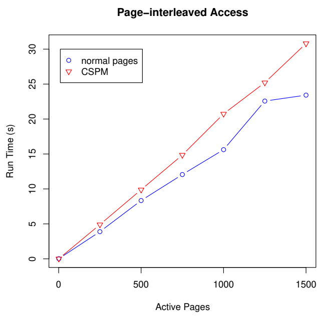
Context sensitive page mappings could be used to protect common attack points in the operating system, such as system call entry points. Garfinkel and Rosenblum describe virtual machine monitor-based protection of sensitive guest OS data in their Livewire VMM-based intrusion detection system [5]. They use the VMM to block changes to such critical pages. Using our context sensitive page mappings, the VMM achieves the same result by redirecting malicious modifications (necessarily made in a data context) to a different page.

In addition to preventing modification, protection in this manner could offer valuable insight into malicious software behavior. Attempted modifications could be captured without compromising the guest OS. Perhaps more interestingly, the changes would *appear* to have been successful from the perspective of the malicious process. If the malicious process functionally depended on the changes taking effect (i.e., if it depended on some specific result of executing its modifications) it would likely crash. However, if the changes were a purely defensive mechanism (as in the case of task hiding modifications), the malware might continue to operate oblivious to the fact that it had not accomplished its aim. Such a result might be of use in identifying or analyzing malicious software.

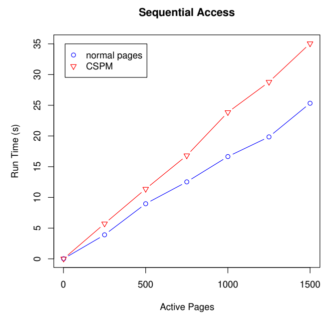
7.2 Shielded Processes

Context sensitive page mappings could be used similarly to protect the memory of particular user processes within a guest operating system. Such protection could be of particular use in defending software components critical to detection of and response to malicious software activity. Virus scanners and intrusion detection systems are susceptible to attack and subversion by malware that gains elevated privilege on the machine. Virtual machine monitor protection of process code or data would prevent modification even by more privileged processes within the guest domain. Such a *shielded process* could be readily examined by privileged software, but would be resistant to attempts to modify its behavior because its memory would be protected by the VMM.

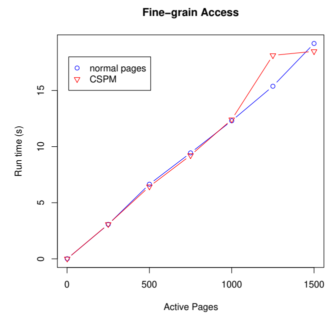
Introducing shielded processes would require collaboration between the VMM and user-level processes within the guest operating system similar to the mechanism we described in Section 4.3 and in our case study of self-checksumming code. Because of the sen-



(a) Page-interleaved Access



(b) Sequential Access



(c) Fine-grained Access

Figure 5. A comparison of context sensitive page mapping performance under our workload with performance using unmodified pages. Each data point represents one thousand iterations of the access pattern over the specified number of pages. All three access patterns consist of small reads from the selected pages. The access patterns in (a) and (b) differ in whether the reads and executes are interleaved or not, but they both operate in isolation on pages. In contrast, (c) staggers reads and executes. This workload is suggestive of applications such as just in time compilation and self checksumming code. The split TLBs allow redirection of reads without incurring the prohibitive expense of trapping and emulating each read or execute operation.

sitive nature of this application, a more secure interface between the user-level process and the VMM would be needed. Otherwise, however, the basic technique remains the same: the VMM manages the page mappings of the shielded processes, presenting one view of memory from a data context and another from execution context. The pages protected in this manner cannot be modified within the guest operating system, as data context access is redirected to a dummy page. It is important to note that the split view of memory would necessarily apply to the shielded process itself, as well. Otherwise, the process hijacking we described in Section 5 could be used to modify the executable code of the process.

As we mentioned in discussing OS tamper resistance, two potential benefits beyond preventing modification of shielded processes spring to mind. Attempted modifications would be captured in the data-context page in which they were made. Additionally, any modifications would appear to have been successfully made from the view of the modifying process. Only observation of the actual execution behavior of the cloaked process would reveal that its code had been unaffected.

8. Conclusions

We introduced context sensitive page mappings provided by the virtual machine monitor. We showed how inherent context sensitivity in memory management hardware and a method for interposing on memory-related guest operating system operations can induce context sensitive views of memory in a target process. Unlike previous work, our VMM accomplishes these tasks independent of the guest operating system, rendering the multiple views transparent to privileged processes within the guest OS.

We described our implementation of context sensitive page mappings in the Xen hypervisor running on the x86 architecture. Our technique loads different values into the data and instruction TLBs, providing context sensitive memory access at page granularity. To demonstrate one application of this technique, we used our VMM in concert with a user process executing within the guest OS to instrument a tamper-resistant test program, successfully circumventing its self-checksumming protection.

By utilizing virtual memory hardware to implement context sensitive page mappings, we avoid incurring significant overhead. We tested the performance impact of our extensions within the Xen hypervisor by creating a benchmarking program that accesses context sensitive pages in a variety of data and execution patterns. Our results indicate that our modifications incur only a slight performance penalty over programs running on unmodified guest operating systems in the normal case. In particular, we see that the technique scales well in the number of context sensitive pages within a program, even in pathological cases such as interleaved data and execution accesses.

Further applications of context sensitive page mappings include protection of sensitive OS and user-level processes. In general, manipulation of memory mappings without assistance from the guest operating system can be useful whenever privileged processes (or the guest OS itself) play a potentially adversarial role.

Acknowledgments

This work is supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510, and National Science Foundataion grant 0627501 The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] ASPACK SOFTWARE. ASProtect Website. `http://www.aspack.com/asprotect.html`.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM Press, pp. 164–177.

[3] BUCK, B., AND HOLLINGSWORTH, J. K. An api for runtime code patching. *Int. J. High Perform. Comput. Appl. 14*, 4 (2000), 317–329.

[4] BUTLER, J., UNDERCOFFER, J., AND PINKSTON, J. Hidden processes: the implication for intrusion detection. In *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society* (2003), pp. 116–121.

[5] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium* (February 2003).

[6] GIFFIN, J. T., CHRISTODORESCU, M., AND KRUGER, L. Strengthening software self-checksumming via self-modifying code. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 23–32.

[7] HOLLINGSWORTH, J. K., NIAM, O., MILLER, B. P., XU, Z., GONCALVES, M. J. R., AND ZHENG, L. MDL: A language and compiler for dynamic program instrumentation. In *PACT '97: Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 1997), IEEE Computer Society, p. 201.

[8] JONES, S. T. *Implicit Operating System Awareness in a Virtual Machine Monitor*. PhD thesis, University of Wisconsin–Madison, 2007.

[9] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting kernel-level rootkits through binary analysis. *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference* (2004), 91–100.

[10] MORGENSTERN, M., AND BROSCH, T. Runtime Packers: The Hidden Problem? In *Black Hat USA* (Las Vegas, USA, 2007).

[11] PAX TEAM. PaX. `http://pax.grsecurity.net`.

[12] SPARKS, S., AND BUTLER, J. "Shadow Walker": Raising the bar for rootkit detection. In *Black Hat Japan* (Tokyo, Japan, 2005).

[13] VAN DE VEN, A. New security enhancements in red hat enterprise linux v.3, update 3. Tech. rep., Red Hat, Inc., 2004.

[14] VAN OORSCHOT, P. C., SOMAYAJI, A., AND WURSTER, G. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Trans. Dependable Secur. Comput. 2*, 2 (2005), 82–92.

[15] VON NEUMANN, J. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput. 15*, 4 (1993), 27–75.

[16] WURSTER, G., VAN OORSCHOT, P., AND SOMAYAJI, A. A generic attack on checksumming-based software tamper resistance. In *IEEE Symposium on Security and Privacy* (2005), IEEE Computer Society, pp. 127–138.