Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis

Jisoo Yang Kang G. Shin
University of Michigan
{jisooy,kgshin}@eecs.umich.edu

Abstract

Hypervisors are increasingly utilized in modern computer systems, ranging from PCs to web servers and data centers. Aside from server applications, hypervisors are also becoming a popular target for implementing many security systems, since they provide a small and easy-to-secure trusted computing base. This paper presents a novel way of using hypervisors to protect application data privacy even when the underlying operating system is not trustable. Each page in virtual address space is rendered to user applications according to the security context the application is running in. The hypervisor encrypts and decrypts each memory page requested depending on the application's access permission to the page. The main result of this system is the complete removal of the operating system from the trust base for user applications' data privacy. To reduce the runtime overhead of the system, two optimization techniques are employed. We use *page-frame replication* to reduce the number of cryptographic operations by keeping decrypted versions of a page frame. We also employ *lazy synchronization* to minimize overhead due to an update to one of the replicated page frame. Our system is implemented and evaluated by modifying the Xen hypervisor, showing that it increases the application execution time only by 3% for CPU and memory-intensive workloads.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection

General Terms Design, Security

Keywords Hypervisor, Application Protection, Data Privacy, Virtualization

1. Introduction

Hypervisor, a virtual machine monitor that directly runs on bare hardware, is becoming popular and has already penetrated deeply into modern computing environments. The Xen hypervisor [3] and the VMware ESX Server [30] are representative examples of this popularity. Hypervisor is not only attractive in consolidating servers and planning server resources, but also advantageous in enhancing system security.

Hypervisor can provide a perfect implementation point for many security applications because it can be inserted between hardware and operating system. For instance, many intrusion detection systems based on hypervisors have been proposed [13, 17, 19]. Hypervisor has also been utilized for providing security services to upper-layer software. For example, hypervisor can provide virtual instances of Trusted Platform Module (TPM) of the trusted computing architecture [4, 12]. The hypervisor used for enhancing security relies on the property that it forms a relatively small and easy-to-secure trusted computing base.

In this paper, we propose a novel usage of hypervisor for implementing a new layer of protection. This protection layer directly secures the memory contents of user-level applications, guaranteeing protection even from a malicious or faulty operating system. This protection is achieved by encrypting the contents of the user memory pages; when a program accesses a memory page, the hypervisor determines which image of the page to provide to the program. Whether to use a decrypted image of the page or a verbatim (hence encrypted) image is determined by the access permission of the program accessing the page.

Our protection system results in a very powerful privacy protection infrastructure that can secure the entire execution of a user-level application. The sensitive information of a user application, including both code and data, is guaranteed to be protected against malicious or faulty operating systems. Therefore, unless the hypervisor itself is compromised, the privacy of an application's memory contents and relevant execution context is preserved even when the operating system has been compromised.

The semantics and interface of this page-based encryption system are abstractly defined in terms of a protection model, which we call *Software-Privacy Preserving Platform* (SP^3). As with any protection model, SP^3 defines the relationship between principals and their access permission. In SP^3, a protection *domain* is defined to be the principal in which a set of access permissions associates domains to a set of cryptographic keys. These access rights govern the ability to use the keys which are to encrypt/decrypt memory pages. If an application program is running inside an SP^3 domain, the application sees the decrypted memory contents through the virtual address space; programs outside the domain, including the operating system, may only see the encrypted memory contents.

In this paper, we focus on the hypervisor-based realization of this SP^3 protection model. Specifically, we describe modifications and extensions made to the hypervisor to implement SP^3. To encrypt pages and secure the SP^3 domain boundary, SP^3 extends the semantics of the paging system and interrupt interface of a CPU. We make the hypervisor emulate the extended paging and interrupt semantics. We also discuss and evaluate techniques to improve hypervisor's emulation performance.

Page-frame replication is a way to reduce the overhead of encryption. In this technique, a hypervisor retains page frame copies containing decrypted images of an original page. When a decrypted image of a page needs to be supplied to a program, the hypervisor

VEE'08 March 5–7, 2008, Seattle, Washington, USA
Copyright © 2008 ACM 978-1-59593-796-4/08/03...$5.00

manipulates the page table entry (PTE) to redirect requests to the page frame copy containing the decrypted image. This reduces the number of costly cryptographic operations that would have to be performed on the entire page frame.

Lazy synchronization is also used to further reduce the number of page-wide encryptions. Under the page-frame replication, synchronization is needed when an update occurs to one of the replicated images. This synchronization propagates the update to the other images by page-wide encryptions. However, with lazy synchronization, this costly synchronization is deferred as long as possible: the synchronization happens not when an image is modified, but when one of the other images is accessed. This technique turns out to be very effective because no synchronization takes place unless two SP^3 domains actively access the two related images simultaneously.

We modified the Xen hypervisor and Linux kernel to implement a hypervisor-based SP^3 system. The modified Xen, serving as the trusted computing base for user-application privacy protection, implements the full semantics of the SP^3 model. Linux, running on top of the modified Xen, is thus removed from the trust base of user applications that are running within SP^3 domains. If Linux violates the protection rules, it will at worst crash the system, but protection of the applications' memory privacy is guaranteed.

We evaluated the Xen-based implementation by measuring the runtime performance of SP^3 applications. Our evaluation results indicate that applications running in the modified Xen experience only a 3% slowdown compared to the same applications running in the native Xen environment. The result also confirms the efficacy of the page-frame replication and the lazy synchronization schemes.

The rest of the paper is organized as follows. Section 2 provides the background on Xen and Linux's memory management and interrupt handling. Section 3 defines the SP^3 protection model. Section 4 presents the key ideas for realizing SP^3 protection in a hypervisor. Section 5 details our modification of Xen. Section 6 evaluates the implementation. Section 7 discusses related work. Finally, Section 8 presents conclusion.

2. Background

In this section, we provide background on the internal workings of hypervisor and operating system. We focus on paging (Section 2.1) and interrupt interface (Section 2.2) as they are closely related to the description of our implementation. We summarize them with an example (Section 2.3) by stepping through what happens when we execute a program in a virtual machine environment. In the following discussion, we assume Linux running on the x86 architecture as the choice of computing platform. We also use Xen as our choice of hypervisor.

2.1 Paging

Paging is the fundamental facility for memory management in contemporary systems. Supported by a hardware Memory Management Unit (MMU), a physical memory page is mapped to a virtual address space via the Page Table Entry (PTE) structure. The MMU translates a virtual address to a physical address by page-table lookup using the virtual address to find a PTE that contains the physical page frame number. Each PTE also contains bit flags such as Present (`P`) bit (accessing a page with `P` bit cleared causes a non-present page-fault), Writable (`W`) bit (writing a page with `W` bit cleared causes a read-only access-violation page-fault), and Dirty (`D`) bit (the processor sets `D` bit when data has been written to the mapped page).

Operating systems, without a hypervisor, directly manipulate the MMU data structure to implement the virtual address space and various paging tricks such as demand-paging, copy-on-write, virtual memory, and disk buffer cache.

With a hypervisor present, an operating system runs on a virtualized hardware platform where the operating system is given a "physical memory" of virtual machine that is an illusion created by the hypervisor. Running between the bare hardware and operating systems, the hypervisor adds another layer of address translation. One way to implement this translation layer is to use shadow page tables [30]. In this technique, a guest operating system's page tables are "shadowed" by real page tables to be directly used by the processor. The hypervisor intercepts all references and updates to the guest operating systems' page tables, performing additional translation, which is called "physical-to-machine" translation.

In para-virtualized systems where operating systems are modified to run on a virtual machine, part of the "physical-to-machine" translation is performed by the guest operating system. This is to avoid complexity and overhead that would otherwise be incurred in a fully virtualized system. Although a para-virtualized system directly exposes MMU states to the guest operating system, the hypervisor still enforces strict rules regarding MMU and page table updates, thus guaranteeing safety to the hypervisor.

2.2 Interrupt

If the processor receives a hardware interrupt or generates an exception, it suspends its execution of current program in order to serve the interrupt or exception. The processor saves the context of the interrupted program for later use when the program is to be resumed. In the x86 architecture, this context, called "exception frame", is saved in the kernel-mode stack upon interrupt. The interrupt is usually the point where the kernel is entered; it causes the processor to vector to the kernel's interrupt/exception handler and the processor mode is switched from user-mode to privileged-mode.

Operating systems, without the hypervisor present, directly handle interrupts. An operating system is to directly program interrupt vector tables to cause the processor to jump to appropriate handler code in the kernel. The handler then performs appropriate actions to resolve the source of the interrupt or exception. Upon completion of handling the event, the kernel runs a scheduler to select the next program to run. To switch the context to the selected program, the kernel executes an instruction called "return from interrupt" with the saved exception frame as the argument of this instruction. The processor switches back to the user mode and resumes execution of the user program.

With the hypervisor present, however, the hypervisor intercepts every interrupt and exception. It examines the cause and nature of the interrupt and then decides whether to handle the interrupt itself or to forward the interrupt to the guest operating system. When it decides to forward the interrupt, it creates an exception frame on the guest operating system's kernel mode stack to emulate the processor's behavior. The content of this exception frame can be programmed by the hypervisor to suit its need.

From an operating system's perspective, the underlying hypervisor's involvement is completely hidden in the case of full virtualization. In a para-virtualized case, such as Xen, the operating system is required to be modified to use the para-virtualized interrupt interface. Nevertheless, the para-virtualizing hypervisor is able to intercept every interrupt and exception and thus fully protected from guest operating systems.

2.3 Example

Here we summarize paging and interrupt in a virtualized environment by using an example where we step through an application being executed. When a user application program is first executed by a process calling `exec()` system call, the kernel handling `exec()` loads the binary (e.g., ELF executable) to read the program header information. Then, the kernel maps code, data, and stack area to the process address space. At this time, the operating system only assigns virtual memory regions and memory is not assigned; the corresponding PTEs for the regions are with their P bit cleared. This is because of the demand-paging scheme. The actual mapping occurs when non-present page-faults on these unmapped pages are handled.

During these events of system call, PTE manipulation, and page-faults, the hypervisor intervenes to virtualize hardware by page-table shadowing and forwarding interrupts: each non-present page-fault vectors first to the hypervisor's handler. If the hypervisor determines the fault should be handled by the guest operating system, it forwards the fault to the guest operating system. Then the page-fault handler in the guest operating system allocates and maps a physical page to the faulting address by updating the corresponding PTE. This PTE update is too intercepted by (or submitted to) the hypervisor for its implementation of shadow paging.

3. SP^3 protection model

This section first provides an overview of the SP^3 system and then illustrates with an example how it provides data privacy protection to user-level applications.

3.1 SP^3 definition

As the principal of the SP^3 protection, we use the concept of protection *domain* [20]: access permission is determined based on the domain context. Each domain of a running SP^3 system is uniquely assigned and identified by an SID (*S*P^3 Domain *ID*) value. To identify the currently executing domain, the SP^3 system may keep a variable called *current* SID. The operating system is assigned SID of 0. Therefore, current SID is automatically switched to 0 when an interrupt or an exception occurs. In most cases, it is safe to consider a domain as a process, but a domain is not exactly the same as a process; multiple processes can share the same SP^3 domain. The kernel always execute with SID 0.

The definition of SP^3 is divided into three parts. First, the *secure paging* extends the interface of a general paging system to maintain the domain boundary. Second, the *secure domain switch* is responsible for safe domain crossing upon interrupts. Last, the *domain operations* handle the dynamics of domain creation and deletion as well as transferring access permissions for sharing. In this section, we only outline the SP^3 constituents, omitting details relevant to actual implementation.

Secure paging: The page table entry (PTE) structure is extended to include a new multi-bit field, called KID (*K*ey *ID*), which is used to locate a symmetric key. An SP^3 system internally keeps a database that stores symmetric keys, called the *key database*. The KID value of a PTE serves as an index to the key database. The SP^3 system also maintains a permission bitmap that tells which domain (identified by SID) can use which symmetric key (identified by KID). The operating system is prohibited from directly accessing the key database and the permission bitmap, but it is allowed to modify the KID field in a PTE. When a domain with SID s accesses memory, page tables are traversed for virtual-to-physical address translation. During the page traversal, the KID k of the matching PTE is checked against the permission bitmap to see if s can use k. If so, the SP^3 system renders the decrypted image of the physical page using the symmetric key indexed by k. Otherwise, the SP^3 system renders the verbatim image of the page. KID 0 is defined as a 'null' key, which always renders the verbatim image when it is used in a PTE. SID 0 is reserved for the domain of the operating system.

Secure domain switch: The current SID changes to 0 when interrupts or exceptions occur, since these events cause traps into the operating system. However, before the operating system takes over control, the execution context of an outgoing domain must be securely stored to prevent information leakage and hijacking of domain context. Thus, the value of machine registers and SID of the interrupted domain are encrypted, creating a *secure domain context*, which is passed to the operating system and then safely stored as an opaque data structure. The secure domain context is also tagged by an authentication hash to prevent overriding SID.

Domain operations: For creation and deletion of domains, we define two operations, Alloc and Free. Alloc creates a domain by assigning an SID, loading symmetric keys to the key database, and initializing KID permissions by setting appropriate bits in the permission bitmap. Symmetric keys may be loaded via a key exchange protocol: a unique public key pair (K_P^+, K_P^-) is assigned to an SP^3 system. To deliver a symmetric key K_s to the system, $\{K_s\}_{K_P^+}$ is passed as an argument to the Alloc, which uses K_P^- to extract K_s and store it to the key database. Free deletes a domain by revoking the key permission and releasing the SID. To transfer key access permissions, two operations, Grant and Release, are defined. Grant allows a domain to permit the other domain to use a key by setting the permission bitmap accordingly. Granting succeeds only when the current domain executing Grant already has permission to the key. To securely identify other domains, each SID is tagged with an identifier, which is loaded when Alloc is called. The identifier may be unique to each application and only known to trusted applications. Release clears the permission bitmap. The last two operations enable secure shared memory among trusted applications.

3.2 SP^3 example

Figure 1 illustrates how SP^3 renders different views of the virtual memory as the current SID changes. In the figure, there are three active domains in the system. Two of them, domains 1 and 2, were created by the Alloc operation, loading symmetric keys to the system along the operations. The remaining domain is the domain 0, which is the domain of the operating system. Using the Grant operation, the permission bitmap was set as shown in the figure. The three domains share the same page table. The figure also shows a section of the page table with the KID values of PTEs. When domain 1 was executing, it saw decrypted images of pages at virtual address 31 and 32. The two pages were decrypted by the symmetric keys referenced by KID 1 and 3. When an interrupt occurred, the current SID was changed to 0. Now, the operating system is running, but it cannot see the decrypted image at the virtual address 31 and 32, because it does not have permission to KID 1 and 3. Instead, the operating system sees the pages' verbatim images. The domain is switched again when the operating system returns from the interrupt. The operating system uses the saved encrypted domain context for domain 2, which will be executing after the domain switch. Domain 2 will see the decrypted images at the virtual address 31 and 33, according to the KID values of corresponding PTEs and permission bitmap entries.

We now illustrate how a user application is generated from source code, transferred to an SP^3-capable host, loaded to memory, and finally, executed on the host. First, an application source code is compiled normally. Compiler/linker tricks are used to ensure the sections are aligned with page boundaries. Then, the binary is encrypted using K_s. A special header that contains $\{K_s\}_{K_P^+}$ is attached to the executable, which is then transferred to an SP^3 -capable host and stored on the disk. When a user runs the application, the executable is loaded by the exec system call, which detects the special header and executes Alloc that creates domain

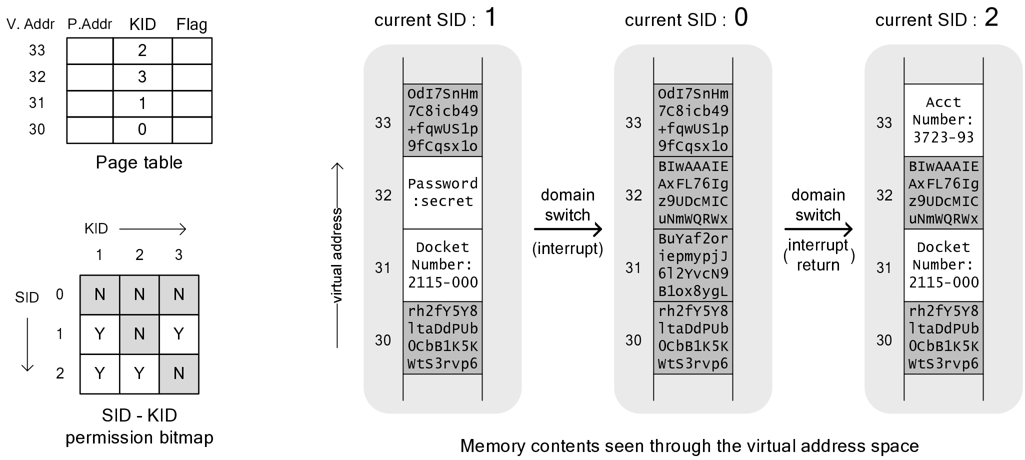

V. Addr
P.Addr
KID
Flag
33
32
31
30
2
3
1
0
Page table
KID
1
2
3
SID
0
1
2
N N N
Y N Y
Y Y N
SID - KID permission bitmap
virtual address
current SID : 1
current SID : 0
current SID : 2
33
32
31
30
OdI7SnHm 7C8icb49 +fqwUS1p 9fCqsx1o
Password :secret
Docket Number: 2115-000
rh2fY5Y8 ltaDdPub OCbB1K5K WtS3rvp6
domain switch (interrupt)
BIwAAAIE AxFL76Ig z9UDcMIC uNmWQRWx
BuYaf2or iepmypjJ 6l2YvcN9 Blox8ygL
domain switch (interrupt return)
Acct Number: 3723-93
Memory contents seen through the virtual address space

Figure 1. An example SP^3 system. SP^3 renders different views of the virtual memory as the current SID changes. Domain 1 has permission to KID 1 and 3, thus it sees decrypted images at virtual addresses 31 and 32. Domain 0 has no permission to any keys, thus the memory contents are rendered as verbatim images. Domain 2 has permission to KID 1 and 2, thus memory contents at virtual addresses 31 and 33 are rendered decrypted.

s, loads K_s on KID k, and sets the permission bitmap on (s,k). The binary is to be loaded to the user address space, but pages have not yet been mapped due to the use of demand paging. Later, when the application causes page-faults on these unmapped pages, the page is loaded from the disk as is, which is the verbatim image encrypted with K_s. The corresponding PTE is fixed to map the page and to contain k in the KID field. When the application resumes, it will see the decrypted image.

Running within the context of the SP^3 protection domain, the loaded application sees the decrypted content of the memory pages via its entire virtual address space. Note that this decryption is done transparently to the application. Also, minimal effort is required from the application in accessing this decrypted image: the application does not have to call special functions nor does it have to set up special barriers in its code. The application can use different cryptographic keys to access different virtual address regions by using different KIDs to PTEs that map the regions. Using a null KID, the application can also set virtual address regions that are not encrypted.

4. Design

In this section, we describe how to realize the SP^3 protection model using a hypervisor. We first present how to efficiently emulate SP^3 secure paging by introducing page-frame replication and lazy synchronization. Then we discuss how to realize SP^3 secure domain switch by changing interrupt semantics. Finally, we provide how to emulate SP^3 domain operations.

4.1 Emulating SP^3 secure paging

As defined in Section 3.1, the heart of the SP^3 system is the SP^3 secure paging, which is capable of rendering different views of the same page frame. That is, the page frame referenced by a PTE with non-zero KID should be rendered as decrypted if the page is accessed when the current SID has the permission to use the KID. We now discuss how to use a hypervisor to emulate such semantic of SP^3 paging.

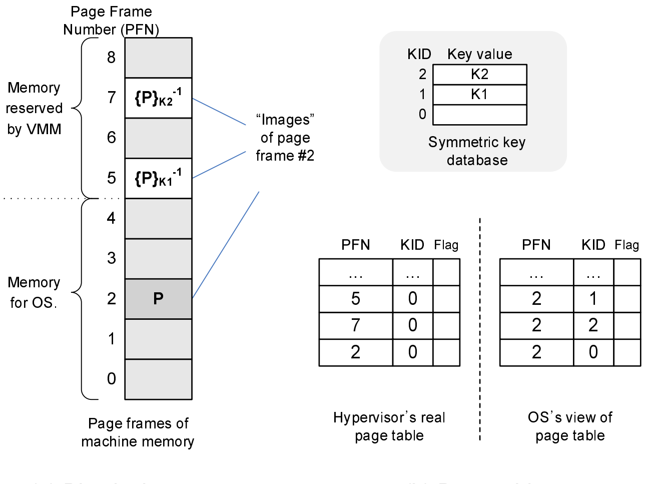
In the design of hypervisor-based emulation of SP^3 paging, we should consider the performance impact of encryption. To provide the decrypted view of a page, the hypervisor should perform a software decryption on the page the size of which typically 4KB. Obviously, a naive design would incur significant run-time performance overhead. Thus, we would like to minimize the performance overhead by using two schemes that can minimize the number of cryptographic operations as described below.

Page-frame replication is the primary vehicle for efficient emulation of SP^3 paging. In this scheme, the hypervisor maintains copies of decrypted images of a page frame. Each of the decrypted images contains the decryption result on the original page using a particular symmetric key. The hypervisor keeps these images in its privately-maintained memory area. Rendering a decrypted view of a page is thus realized by redirecting the page to one of the decrypted images. The hypervisor can realize this redirection by virtualizing access to the page tables; it intercepts modifications on page tables to realize the extended KID field, and it induces page-faults to provide the hypervisor the points to check the permission and to perform actual redirection. These operations are directly handled by the hypervisor, and thus hidden to the operating system.

Figure 2 illustrates how a hypervisor implements the page-frame replication scheme. In Figure 2(a), physical page frame number (PFN) 2 has two decrypted images located on PFN 5 and 7, each of which is the decryption of PFN 2 using the symmetric key selected by KID 1 and 2, respectively.[1] Figure 2(b) shows page tables virtualized by the hypervisor. On the right side is the virtualized page table which the operating system can modify. The virtualized page table is shadowed by the real page table which the MMU refers to. In the figure, the operating system programmed virtualized page table such that PFN 2 is mapped in three different PTEs with different KID values of 0, 1, and 2. The corresponding PTEs in the real page table then contain PFN 2, 5, and 7, and thus, the hypervisor renders decrypted views on the same page, realizing the SP^3 paging semantic.

Although keeping decrypted images reduces the number of cryptographic operations, those images must be synchronized if one of the images or the original page gets modified. The synchronization is necessary for providing consistent views on all images; if a program modifies a decrypted image, then the original page, although its content is encrypted, must reflect the change when ac-

[1] Actual hypervisor adds another layer of address indirection by which a "physical address" (the virtualized address local to a virtual machine) translates to a "machine address" (the physical address of the underlying hardware). To simplify the discussion, we ignore this translation layer.



(a) Physical memory (b) Page tables

Figure 2. Shown in (a), the hypervisor keeps decrypted copies of an original page frame (PFN 2) in different memory locations (PFN 5 and 7). The hypervisor uses one of these page frames when the original frame is mapped with a PTE with a KID value. The redirection of page frame is performed transparently by manipulating page tables as shown in (b).

cessed later. Obviously, this involves cryptographic operations and, unless properly handled, incurs high runtime overhead.

We solve this problem by employing *lazy synchronization* that reduces the number of synchronizations among the images by delaying update propagation until the last minute. Synchronization is performed only to the pages that need to be updated and only when it is necessary; the synchronization happens not when one image is modified, but when one of the other images is accessed. This is realized by keeping track of the most-recently updated image among the images including the original. Tracking the most-recently updated image is achieved by checking `D` (dirty) bit of PTE. The content of the most up-to-date copy is propagated to one of the 'stale' pages by means of the hypervisor's page-fault handler. The hypervisor clears `P` bit of those stale pages to induce a page-fault through which the hypervisor can propagate updates behind the scene.

The lazy synchronization scheme is highly effective because it exploits the fact that there are limited occurrences of active sharing in application programs: if a page is not accessed during the activation of the particular SP^3 domain, it will not generate any page-fault. Therefore, the images of a page frame are synchronized only when necessary, thereby reducing the runtime overhead of re-encryption for synchronization. Note that this lazy synchronization does not incur any encryption overhead for most of the normal application execution scenarios because page frames are not shared among different SP^3 domains.

4.2 Emulating SP^3 secure domain switch

The SP^3 secure domain switch extends the interface of interrupt and exception. To recap, the current domain switches to operating system's domain, SID 0, when an interrupt or exception occurs. Also, upon occurrence of these events, the execution context of the outgoing domain must be securely stored in the 'secure domain context' to prevent information leakage and hijacking of the domain context. We now discuss how to emulate such SP^3 interrupt semantic in a hypervisor.

We can realize the transition of current domain by intercepting every interrupt and exception generated by hardware. Hypervisors are, by definition, capable of intercepting all interrupts and exceptions. When the hypervisor forwards an interrupt to a guest operating system, the hypervisor can change the current domain by setting current *sid* variable to 0.

The secure domain context, which is to contain register contexts and SID of the outgoing domain, is realized by extending the exception frame structure. As briefed in Section 2.2, the processor generates an exception frame into the kernel mode stack upon an interrupt, and the hypervisor already simulates this behavior to virtualize interrupts. We extend this exception frame to contain a secure domain context. Thus, this extended exception frame has a new field for general-purpose registers (GPRs) and SID value of the outgoing domain. These fields are encrypted and hashed. When the hypervisor forwards an interrupt to a guest operating system, it generates this extended exception frame instead of the original one.

The GPRs are cleared when the hypervisor raises a virtual interrupt by generating a secure exception frame. Upon receipt of this interrupt, the guest operating system will find the GPRs to be zeroed out. This is to prevent information leakage upon domain switch, because the operating system is untrusted.

After handling the virtual interrupt, the guest operating system requests the hypervisor to perform a 'return-from-interrupt' operation using the extended exception frame that have been saved from a previous interrupt. Upon receipt of this request, the hypervisor processes the extended exception frame to restore GPRs and SID value.

4.3 Emulating SP^3 domain operations

In the hypervisor-based realization, the domain operations are basically requests made to the hypervisor. Therefore, the interface for the domain operations could be simply realized by creating a new hypercall entry for each domain operation. However, we can alternatively achieve this by creating virtual 'instructions' for the domain operations. Execution of this instruction opcode will generate an 'invalid-opcode' fault, which should be captured by the hypervisor. The hypervisor will then examine the opcode to perform the matching SP^3 domain operation.

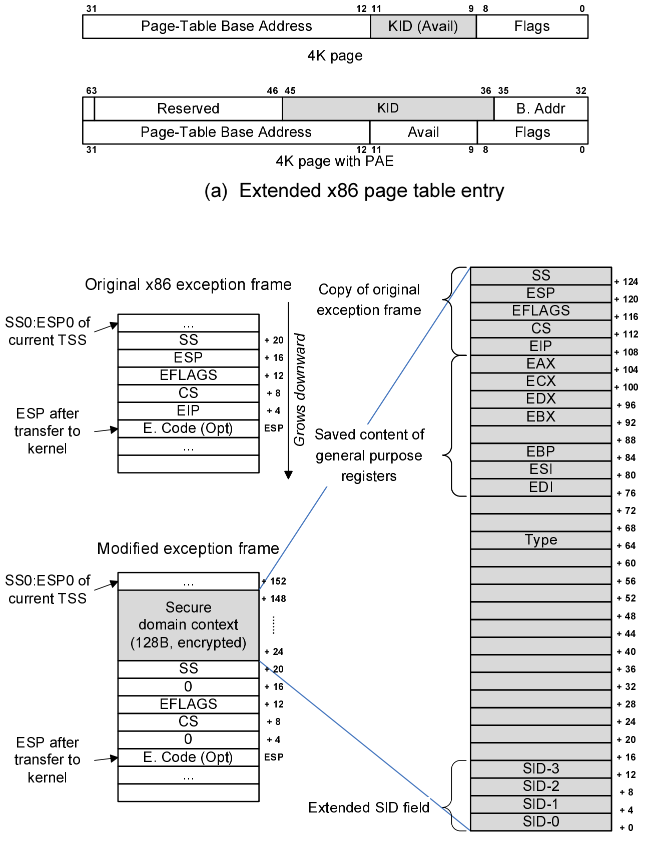
We favor defining new instruction opcodes than extending hypercall entries, because by creating new opcodes, the entire SP^3 interface looks as if the processor were supporting SP^3: from the perspective of an operating system, and hence, feels no functional difference between the hypervisor-based implementation and direct-hardware modification. Using the 'invalid-opcode' fault has no performance disadvantage over extending hypercall, because a hypercall is also implemented by generating a software interrupt.

5. Implementation

We modified Xen hypervisor [3] which runs on top of x86 (IA-32) architecture [16]. Xen runs with higher privilege than the virtual machines it manages, and thus, it has a safe perimeter against operating systems.[2] Note that Xen's administrative virtual machine, known as dom0, cannot access the private area of Xen, therefore guaranteeing safety.

One unfortunate name collision needs to be resolved before we proceed. In Xen-terminology, a "domain" refers to a virtual machine instance created by Xen. In this paper, this usage is discouraged to eliminate confusion with our SP^3 protection domain. Henceforth, Xen's "domain" is referred by 'virtual machine', and we use 'SP^3 domain' or simply 'domain' to refer to SP^3 domain.

[2] We can say that Xen and its SP^3 extension implemented within, can form a small and secure trust base, provided Xen is securely bootstrapped and attested. Secure bootstrapping and providing integrity measures are important for securing a trusted computing base, but it is out of scope of this paper. We refer the readers to secure bootstrapping [18] for safe and secure loading of the Xen hypervisor.



(a) Extended x86 page table entry

(b) Modified x86 exception frame

Figure 3. Extensions made to x86 PTE and exception frame structures. The extended PTEs includes a new multi-bit field for KID value. The secure exception frame, which is to be generated on the kernel stack upon interrupt, is larger than the original exception frame to contain fields for the GPRs and SID of outgoing domain.

In this section, we first describe the implementation of emulating the modified interface of extended x86 architecture for SP^3 support. Then, we detail the realization of our design on the hypervisor, focusing on the mechanisms to efficiently emulate the SP^3 secure paging.

5.1 Emulating the modified x86 interface

It is straightforward to incorporate into Xen the data structures directly related to the SP^3 protection model. We modified Xen to keep variables for storing the permission bitmap and cryptographic keys. To identify which SP^3 domain is executing in the system, an integer variable called current_sid is created to store the SID value of currently executing SP^3 domain.

It gets tricky when we make Xen emulate the new extensions to CPU-level interface, specified in Figure 3. The extensions are to reflect the new KID field in PTE structure, and to generate a secure interrupt frame upon interrupt. Obviously, we did not actually modify the hardware; the specification given here is used as the reference interface that Xen ultimately emulates.

Figure 3(a) shows the modified PTE structures into which the KID field is integrated. In its 'native' paging mode, the original x86 has 3 bits available for the KID field.[3] In its Physical Address Extension (PAE) paging mode, which has an expanded PTE structure, 27 bits available for the KID field. The actual number of bits required for the KID depends on the size of required KID space. For instance, when 10 bits from the PAE-enabled PTE structure are selected as KID field, as shown in the figure, it allows the KID space to range from 0 to 1023.

We modified Xen to emulate this PTE extension by adding a code that can interpret the KID field. This code is added to the Xen's handler routine responsible for PTE updates. This handler routine is always invoked when a guest operating system modifies a PTE to map a page. Since MMU updates are sensitive, Xen makes sure it intercepts all PTE updates. In the para-virtualized environment of Xen, operating systems can update a PTE either by making a PTE-update hypercall, or by directly modifying the PTE. Either way, Xen can always intercept the PTE update: a hypercall causes trap to Xen by definition; a modification to a PTE incurs a page-fault since the pages used as guest page tables are always mapped with W bit cleared, meaning any attempt to write to the guest page tables causes a access-violation page-fault, trapping into Xen. Therefore, by modifying the Xen's handler for PTE updates, the safe and transparent illusion of the extended KID field can be achieved. A guest operating system can update a PTE as if the hardware supported the KID extension.

Another modification we made to CPU-level interface is the secure version of x86 exception frame as specified in Figure 3(b). This secure exception frame, instead of the original x86 exception frame, is generated on the operating system's kernel mode stack when an application running in an SP^3 domain gets interrupted. As shown in the figure, the first top 128 bytes of the secure exception frame represent the secure domain context , which is encrypted using a key private to the SP^3 system. This encrypted part contains the entire register context of the interrupted program. The SID value of the interrupted SP^3 domain is also saved at SID-0 to SID-3 field. SID value is stretched and then hashed to avoid overriding SID. The secure domain context is followed by the plaintext part which is identical to the original x86 exception frame except for the zeroed EIP and ESP fields.

To generate this secure exception frame, we modified Xen's interrupt bouncer code that handles forwarding of an interrupt to a guest operating system. Xen monitors every interrupt by intercepting it. If Xen decides to forward an interrupt to a guest operating system, it "artificially" creates an exception frame by writing to the kernel mode stack of the guest operating system, emulating the behavior of the CPU. This forwarding is implemented by the interrupt bouncer code which we modified in such a way that if current_sid is not 0, it generates an secure exception frame instead of standard one. At the moment Xen transfers control to the guest operating system, General-purpose registers (GPRs) are cleared and current_sid is set to 0.

To perform a return-from-interrupt on this secure exception frame, we defined a new instruction, called S_IRET. Executing this instruction causes traps to Xen via invalid-opcode fault. We modified Xen's invalid-opcode handler to unwind the secure exception frame and resume the interrupted program. To restore SP^3 domain context, Xen reloads GPRs and sets current_sid back from the saved values of the secure exception frame. The SP^3 paging extension takes advantage of this to correctly prepare a data structure when the operating system requests page table update with a nonzero KID value.

A scheme is provided for the operating system and user applications to pass arguments and return values via GPRs. In this

[3] In fact, these bits are intended to be utilized by the operating system. But Linux, the operating system we use, does not utilize them.

scheme, GPRs are normally cleared unless the cause of exception is a software interrupt; a user process can pass system call parameters via GPRs. The Type field tells whether GPRs have been cleared or not, indicating that the secure exception frame was generated by a software interrupt or another type of exception. Upon receipt of an interrupt–return request, Xen reloads GPRs from the saved register values unless the Type indicates the the secure exception frame was generated by a software interrupt, enabling a convenient channel for passing system call return values. Note that this facility does not necessarily incur leakage of information through GPRs, because applications can always clear contents of registers unused in the system call before generating a software interrupt.

5.2 Implementation detail of SP^3 secure paging

During initialization, Xen reserves a pool of physical page frames for storing decrypted images. A page frame containing decrypted image is mapped by PTEs with PFN value of original page frame and non-zero KID field. It is important to recognize this class of PTEs with non-zero KID and the page frames mapped by them. Hence, we assign names for them to facilitate description. In the following discussion, we will refer to a page mapped with non-zero KID as *SP^3 page* and the PTE for SP^3 page as *SP^3 PTE*.

We use the P (present) bit of SP^3 PTE so that the processor can generate a non-present page-fault. These extra page-faults are intended to provide trap into Xen when accessing a SP^3 page needs attention of Xen, such as performing a check for PTE redirection. The page-fault handler of Xen is modified to separate this type of page-fault from other normal page-faults by examining the KID field of the PTE that caused the non-present page-fault.

Under the para-virtualizing architecture of Xen, this nontraditional usage of P can cause problems, since page tables are directly exposed to the operating system. We clear the P bit purposely even though the page is physically mapped by the operating system kernel. However, the operating system may be confused because it is possible for the operating system to see the P bit cleared when the bit was set before.

Without the hypervisor's shadow page table support, we would have only resolved this problem by modifying the operating system. However, Linux —our target operating system— already has a mechanism that can treat PTEs with P bit cleared as physically present. This facility fortunately enabled us to avoid excessive modifications. In the current version of Linux, a page is considered non-present only if both P bit and PAT bit (bit 7) are cleared.[4] We exploit this by setting PAT bit for SP^3 PTEs so that Linux can recognize the page as present. Also, Linux does not get any additional page-fault from this because Xen filters page-faults generated by SP^3 PTE.

When a page-fault is generated by SP^3 PTE, Xen fixes the fault by setting P bit with an appropriate value on PTE. Which page should be used is determined according to the SP^3 rule: if the current SID has access to the KID, Xen uses the decrypted image page. In other cases, original page is used. In this process, the D (dirty) bit of the PTE is checked to synchronize between the two copies. The synchronization entails 4Kb AES operation which is time-consuming. However, under our lazy synchronization scheme, it happens only when it is needed. In practice, the synchronization is under full control of a user program (e.g., the program explicitly shares an SP^3 page with another SP^3 domain), or it occurs if the operating system wants to swap out the page to disk, which is rare in modern platforms and already a very slow operation.

SP^3 PTEs have to be invalidated by clearing P bit whenever domain is changed. This ensures the access permission of SP^3 pages to be reevaluated when the other SP^3 domain accesses that SP^3 pages. Once the SP^3 page is made present, access on the page will not generate any page-fault and the program can proceed. However, if the SP^3 PTEs' P bits are not cleared when SID changes, the other domain will access the old page, which can possibly contain decrypted image. Therefore, this SP^3 PTE invalidation ensures the access permission of SP^3 pages to be reevaluated.

To implement this invalidation logic, Xen maintains a list of SP^3 PTEs that should be made non-present upon change of SID. When Xen reevaluates an SP^3 PTE by setting P bit, it also adds the PTE to the list. Later when SID changes, Xen goes through this list to clear the P bit, and the list is emptied. Exceptions and S_IRET can only change SID, the SP^3 PTE invalidation is performed when Xen handles those operations.

6. Evaluation

In our evaluation, we want to answer the following questions:

- How much performance degradation do SP^3 applications experience?
- How effective are the page-frame replication and the lazy synchronization?
- How does the performance overhead vary with application's memory access pattern?
- What is the impact of SP^3 secure interrupt on performance?

To evaluate the impact on the performance of using SP^3 protection, we first measured overall performance overhead with CPU- and memory-intensive workloads. Such a workload is chosen since our modifications are made on the CPU and the memory management part. We then performed a micro-benchmark measuring the performance impact of the locality of the applications' page reference patterns.

6.1 Methodology

The machine used in our evaluation has a 3.2 GHZ Pentium 4 (HT) processor with 1 GB of RAM. We used Xen version 2.0.4 and Linux kernel version 2.6.10, which is para-virtualized for Xen. Only single virtual machine instance, namely dom0, is used for all experiments. Xen allocates 512 MB of RAM for this guest virtual machine. For the Linux kernel setting, we used the default configuration in the original Xen distribution, which results in a uniprocessor kernel image without highmem support. We chose AES and RSA for our cryptographic primitives whose implementation was taken from OpenSSL version 0.9.7e as C code without additional optimization.

We measured the performance of the SP^3 system by executing benchmark programs on a system running our SP^3 enabled Xen. This modified Xen is allocated additional 256 MB of RAM dedicated for storing decrypted images. For each benchmark program, two executables are generated from the same source code: one is an encrypted executable that can be executed only on the SP^3 enabled system, and the other is a normal insecure executable that can be used for performance comparison with a system without SP^3 protection. Both executables are statically linked with a modified version of dietlibc C library [1].

6.2 The price of protection measured in performance penalty

We wanted to know how much an application needs to pay for the SP^3 protection in terms of performance penalty. Since our SP^3 im-

[4] This facility is devised for memory regions mapped with PROT_NONE type. Linux clears P bit but sets PAT bit when loading a PTE for a page of that type. This way, the page is considered present by the kernel but CPU generates a non-present fault upon access. This way the kernel can raise protection violation, realizing PROT_NONE semantic.

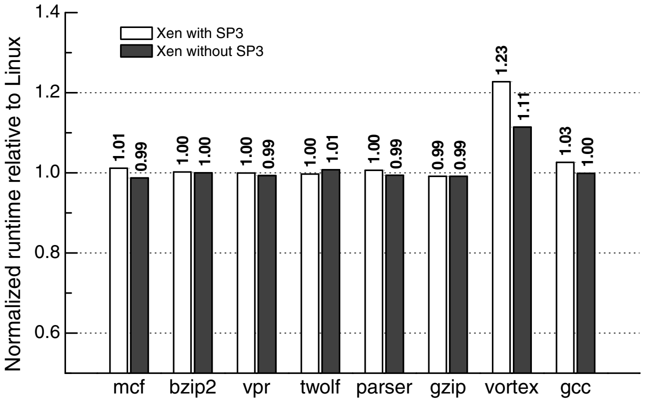


Figure 4. Application benchmark results. The bars and the numbers on top represent runtime of benchmark programs normalized to native Linux. Each value is the mean of 5 trials.

plementation changed the paging interface, we chose CPU- and memory-intensive workloads for measuring the impact on performance. For the workloads, we selected 8 programs from the SPEC CPU2000 integer benchmarks. Measuring the time to complete each program, we compared the running time of the workloads in three different setups. In the first setup, labeled as 'Without SP^3', normal insecure executables were executed on the native Xen. In the second setup, labeled as 'With SP^3', the encrypted SP^3 executables were executed on the modified Xen. In the last setup, normal executables were executed on native Linux without Xen. To avoid disk loading, all measurements were made right after a prior run of the same program.

Figure 4 shows the benchmark results. The performance overhead is presented as a relative runtime normalized to native Linux without Xen. Overall, it takes less than 3% longer to finish the same program with SP^3 protection, except for `vortex` benchmark.

This good performance result empirically confirms that both page-frame replication and lazy synchronization are indeed effective in reducing costly cryptographic operations. Since Xen keeps the copies of decrypted images, decryption is performed only when the image is initially created from the page that contains the original verbatim image. Once the decrypted image is created, it continues to be used without incurring any further decryption until there is a need to synchronize among images. However, this synchronization does not occur even after the application updates the decrypted image, thanks to the lazy synchronization. The update in a decrypted image propagates to the original page only when the operating system accesses the original page, which rarely happens because an operating system doesn't usually access application memory under the normal condition.

Since the overhead of page-wide encryptions is negligible, we can assume that the runtime penalty comes from the overhead of the PTE invalidation and subsequent page-fault for reevaluation. This type of penalty is paid less by a program with a small runtime footprint (i.e., accessing less pages during its activation between interrupts) than by one with a large footprint. If we assume that a statically larger program has also a larger runtime footprint, we can therefore expect that a statically larger program pays a higher penalty due to PTE invalidation than smaller one. In fact, it is found that there is a positive relationship between the runtime footprint and the performance penalty. In Section 6.3, we present a more clear relationship between them with a micro-benchmark varying the size of dynamic memory footprint.

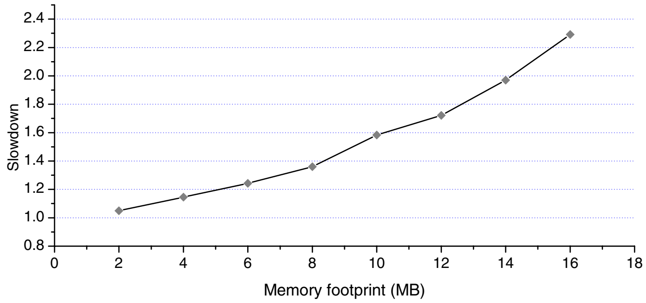


Figure 5. Impact of memory access locality. Performance of a test program is shown as the dynamic memory footprint of the test program increases. The y-axis shows increased runtime in SP^3 system normalized to the native Xen.

Securing interrupts can be another source of potential performance degradation. Although securing an interrupt involves cryptographic operations, in general it does not add much overhead because the frequency of interrupt is very low relative to the processor clock speed in modern computing systems, and also the overhead is overshadowed by the greater overhead of interrupt service routines and the resulting I/O operations. However, it is possible that securing interrupts can degrade performance of certain applications, such as the one that requests many simple system calls that the kernel can quickly return.

Secure interrupts are the culprit of the anomaly of the `vortex` benchmark program in this experiment. `vortex` is an object-oriented database program which is modified for the inclusion to the SPEC benchmark suite. The result of the modification is a program that runs in a tight loop of database transactions, which incur a lot of system calls. In Section 6.4, we analyze the system call overhead in detail.

We were curious how disk buffer cache affects performance since loading an executable from the disk carries initial decryption overhead in addition to the disk access overhead. We therefore performed a comparison between 'cold' and 'hot' runs of the workloads. In 'cold' run, we execute a workload program in boot-clean state without prior execution of the program, whereas in 'hot' run, we execute the program right after executing the same program.

When we measured and compared the two, we failed to find any significant difference. Although it is a non-trivial overhead to decrypt a page for creation of a decrypted image copy, it is obvious that the encryption penalty is hidden under the heavier overhead of disk I/O operation.

6.3 Impact of memory access locality on performance

To obtain a more clear relationship between the runtime footprint and the PTE invalidation penalty, we performed a micro-benchmark with a varying runtime footprint size. The benchmark program touches all of the allocated pages continuously through a loop, therefore we can artificially control the dynamic memory footprint.

Figure 5 shows the results. As expected, runtime penalty increases as dynamic memory footprint increases. If the dynamic footprint is small enough (less than 4MB), the performance degradation is less than 15%. The performance penalty increase as the footprint increases, and when the footprint hits 14MB, it takes twice as long.

Since many applications exhibit strong locality in accessing main memory, as can be seen in our SPEC benchmark, users of SP^3 system should not generally concern the performance degradation. Also this result is obtained from the un-optimized implementation: we didn't aggressively optimized the invalidation and reevaluation logic. It is probable that we can further reduce the impact of invalidation on the performance by optimizing the invalidation logic. For example, we are considering invalidating a page directory entry instead of page table entry to reduce the number of entries in the invalidation list.

6.4 Impact of frequent system call on performance

SP^3 applications that request system calls frequently are expected to suffer from the encryption overhead of SP^3 secure interrupt. To assess the increased cost of system calls in SP^3, we performed a micro-benchmark that measures the overhead of system calls. We used system call latency benchmark of `lmbench`, which was slightly modified to fit to the SP^3 environment.

Syscall Type	With SP^3	Without SP^3	Native Linux
`null`	10.6	0.952	0.322
`open`	22.9	3.27	2.07

Table 1. System call latency measured in microsecond.

Table 1 shows the benchmark results. 'null' measures the round trip overhead between user and kernel mode with minimum work required inside the kernel. 'open' measures how long it takes to open and then close a file, thus more time is spent in the kernel.

As expected, the system call overhead is significant higher in SP^3 compared to both native Xen and Linux. This increased latency is due to the increased round trip time for user/kernel crossing, which is caused by the encryption of SP^3 secure interrupt frame. This result also confirms the slowdown of `vortex` benchmark in Section 6.2: `vortex` calls the system more than 500k times until its completion. Since it takes less than 4 minutes to compete in native Linux, the rate of system call is roughly 2,100 requests per second, which explains the anomaly of the `vortex` benchmark.

7. Related work

Virtual machine monitors are being utilized to solve systems security problems. IntroVirt [17] used virtualization to log/replay system events, achieving a perturbation-free intrusion detection system that can also detect past intrusions. Garfinkel and Rosenblum [13] designed an intrusion detection system based on virtual machine introspection. The proposal of using hypervisor in commodity mobile systems [11] is motivated by the advantage of using hypervisor for implementing security services.

SecVisor [26] is a tiny hypervisor that protects the code integrity of kernels. It achieves small code size by supporting only one operating system instance and virtualizing only memory. SecVisor ensures integrity of the code executing in privileged mode, thereby preventing unauthorized kernel modification. In contrast, our system protects *user-mode* application's *data secrecy*, not the kernel integrity. In addition, SP^3 protects applications even in the presence of a compromised kernel.

Proxos [29] is a hypervisor-based trust-partitioning system in which users can configure the trust on the operating system. A trusted application runs in a private trusted operating system created by underlying hypervisor. A set of system calls, which the user can specify, is dynamically forwarded into another operating system instance, which is full-fledged operating system but untrusted. In contrast, our system provides protection to user memory in per-page basis, and does not require a private operating system instance.

The protection ring [24, 25] defines multiple levels of privilege mode on a processor. Using more than two protection rings (the least privileged ring is given to user space), multiple layers of kernel can be constructed with varying degrees of privilege, and hence, importance. Although many processors support multiple privileged rings, they are not widely used except for the layer of hypervisors [3, 5]. Many other architectures have been proposed to protect the more important kernel part against failures of less important kernel parts [8, 31]. In contrast, our approach has a completely different goal: SP^3 aims to protect the user applications running in the least privileged ring against operating system compromises.

Secure processors are a class of processors with hardware implementation of various cryptographical primitives. Some of them can provide secrecy and integrity protection directly to individual processes bypassing all or most of the operating system. AEGIS [28] and Cerium [7] are among them and focus primarily on physical tamper-resistance. The XOM secure processor [21] can host a fully-untrusted operating system, and can thus protect applications from operating system compromises. However, XOM requires special hardware and heavy compiler/assmembler support, limiting its practicality.

Many of approaches to protection of applications' sensitive information can be viewed as code obfuscation [9, 10, 22]. Unfortunately, obfuscating a program is considered a weak form of protection, as shown by Barak *et al.*[2]. In contrast, our system is stronger than obfuscation since we use cryptography directly for the protection of information.

Reducing the size of trust base as well as separating trust dependency have been general strategies to enhance system security and robustness. Those strategies are used to solve problems in many areas such as file system [33], kernel construction [31], application partitioning [6, 32] and relocating service to the virtual machine monitor [12, 19].

The negative impact of the size and complexity of a trusted computing base (TCB) on system security has been widely recognized [15]. Härtig's Nizza architecture [14], Singaravelu's AppCore [27], and IBM's PERSEUS [23] are example efforts to reduce TCB size and complexity.

8. Conclusion

In this paper, we presented a novel way of using hypervisors to protect application data privacy. Even if the operating system is not trustable, the hypervisor prevents application information from unauthorized exposure. This protection is achieved by encrypting the contents of the user memory pages; when a program accesses a memory page, the hypervisor determines which image of the page to provide to the program. Whether to use a decrypted image of the page or a verbatim image is determined by the access permission of the program accessing the page.

We detailed the modifications and extensions made to the hypervisor to realize this page-granular secrecy protection. To encrypt pages and secure the protection boundary, we extended the semantics of paging system and interrupt interface of a processor. We made the hypervisor emulate the extended paging and interrupt semantics.

We also employed performance-improving techniques, such as page-frame replication and lazy synchronization: page-frame replication reduces the number of cryptographic operations by keeping multiple decrypted versions of the same page frame; lazy synchronization further minimizes the overhead triggered by an update on one of the replicated page frames. Our system is implemented and evaluated by modifying the Xen hypervisor, showing that it increases the application execution time only by 3% for CPU and memory-intensive workloads.

Acknowledgments

We would like to thank Thuy Vu and the anonymous reviewers for their help and feedback. The work was supported in part by the US Airforce Office of Scientific Research under Grant FA9550-07-1-0423.

References

[1] Diet Libc a libc optimized for small size, http://www.fefe.de/dietlibc/.

[2] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 1–18, Aug 2001.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steve 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 (SOSP)*, pages 164–177, Oct 2003.

[4] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, pages 305–320, Aug 2006.

[5] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–11, Dec 1995.

[6] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, Aug 2004.

[7] Benjie Chen and Robert Morris. Certifying program execution with secure processors. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, May 2003.

[8] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 140–153, Dec 1999.

[9] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, pages 184–196, Jan 1998.

[10] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *Transactions on Software Engineering*, 28(8):735–746, 2002.

[11] Landon P. Cox and Peter M. Chen. Pocket hypervisors: Opportunities and challenges. In *Proceedings of the 8th IEEE Workshop on Mobile Computing Systems and Applications (HotMobile)*, Feb 2007.

[12] Tal Garfinkel, Ben Pfaf, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine–based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206, Oct 2003.

[13] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS)*, pages 191–206, Feb 2003.

[14] Hermann Härtig. Security architectures revisited. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, pages 16–23, Sep 2002.

[15] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components – small kernels versus virtual-machine monitors. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, page 22, Sep 2004.

[16] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual.

[17] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 91–104, Oct 2005.

[18] Bernhard Kauer. OSLO: Improving the security of trusted computing. In *Proceedings of the 16th USENIX Security Symposium*, pages 229–237, Aug 2007.

[19] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 223–236, Oct 2003.

[20] Butler W. Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Mar 1971.

[21] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 178–192, Oct 2003.

[22] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 27–31, Oct 2003.

[23] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, Zurich Laboratory, Apr 2001.

[24] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, Jul 1974.

[25] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep 1975.

[26] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350, Oct 2007.

[27] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the EuroSys 2006*, Apr 2006.

[28] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, pages 160–171, Jun 2003.

[29] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating system configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 2006.

[30] Carl A. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–194, Dec 2002.

[31] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isonation for linux using mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 31–44, Oct 2005.

[32] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, Oct 2001.

[33] Xin Zhao, Kevin Borders, and Atul Prakash. Towards protecting sensitive files in a compromised system. In *Proceedings of the 3rd International IEEE Security in Storage Workshop*, Dec 2005.