AFRL-OSR-VA-TR-2012-0494

Design and Implementation of Application Software and Data Protection in an Untrusted Environment

Kang G. Shin
UNIVERSITY OF Michigan

01/05/2007
Final Report

DISTRIBUTION A: Distribution approved for public release.

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ RTA1
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to the Department of Defense, Executive Service Directorate (0704-0188). Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
01-05-2007	Final	01-05-2007 -- 30-11-2010

4. TITLE AND SUBTITLE

Design and Implementation of Application Software and Data Protection in an Untrusted Environment

5a. CONTRACT NUMBER

5b. GRANT NUMBER

FA9550-07-1-0423

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Kang G. Shin

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The University of Michigan
2260 Hayward St.
Ann Arbor, MI 48109-2121

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research (AFOSR/RSL)
Suite 325, Room 3112
875 N. Randolph Street
Arlington, VA 22203-1768

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-OSR-VA-TR-2012-0494

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unlimited – Approve For Public Release

13. SUPPLEMENTARY NOTES

None

14. ABSTRACT

Application protection is traditionally provided by the underlying operating system through process isolation. However, it has become practically impossible to completely secure modern operating systems due to their extremely large size and diversity.
In this project, we introduce a new system of protection for applications, called the Software-Privacy Preserving Platform (SP3), that directly guarantees application privacy. Even when the operating system is compromised, SP3 prevents the unauthorized exposure of application information. By extending the conventional paging system, SP3 achieves a general, flexible, and easy-to-use protection interface with minimal intrusion to existing systems. We have implemented a prototype SP3 system by modifying Xen hypervisor, running modified Linux on top of it. Our experimental evaluation shows that \sppp~increases the application execution time by 0-22% for CPU and memory-intensive workloads.

15. SUBJECT TERMS

Application protection, untrusted operating systems, virtualization, page-granular encryption and decryption

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
Unclassified	Unclassified	Unclassified	None		Kang G. Shin
					19b. TELEPHONE NUMBER (Include area code) 734-763-0391

Protection of Applications Secrecy by Extending Paging System

KANG G. SHIN
Department of Electrical Engineering and Computer Science
The University of Michigan
{jisooy,kgshin}@eecs.umich.edu

Application protection is traditionally provided by the underlying operating system through process isolation. However, it has become practically impossible to completely secure modern operating systems due to their extremely large size and diversity. In this report, we introduce a new system of protection for applications, called the *Software-Privacy Preserving Platform (SP^3)*, that directly guarantees application privacy. Even when the operating system is compromised, SP^3 prevents the unauthorized exposure of application information. By extending the conventional paging system, SP^3 achieves a general, flexible, and easy-to-use protection interface with minimal intrusion to existing systems. We have implemented a prototype SP^3 system by modifying Xen hypervisor, running modified Linux on top of it. Our experimental evaluation shows that SP^3 increases the application execution time by 0–22% for CPU and memory-intensive workloads.

Categories and Subject Descriptors: D.4.6 [**Security and Protection**]: Access controls

General Terms: Design, Security

Additional Key Words and Phrases: data secrecy, virtualization, paging

1. INTRODUCTION

Privacy is perhaps the most important security concern for many user-level applications; users would prefer to crash their applications, rather than revealing their private data. This is because exposing sensitive information to an unauthorized users/entities is considered worse than failing to execute their applications. In fact, except for safety-critical real-time systems, most applications and users can cope with execution failures with various recovery and backup strategies. For instance, a word processor can recover from a backup copy after a crash, or a disconnected secure shell client can simply reconnect. However, the exposure of sensitive information in these applications is considered far worse than their execution failure. The word file may contain classified or confidential information. Exposure of the encryption key of a shell client leads to compromising the secure communication.

Privacy protection for applications is usually provided by operating systems through process isolation. Ironically, operating systems themselves can be the biggest threat to the applications; a modern operating system is very difficult to secure because of its extreme size and diversity. Once compromised, the operating system becomes a powerful weapon for thieves of applications' sensitive information. We argue (and show) this problem can be solved by devising a new application protection system that is not under the operating system's control and can thus directly secure the information of applications. Even if the operating system were compromised, the protection system can prevent the exposure of application information. The worst an attacker can do is to crash the system, thus causing the applications to fail to execute, which is usually less harmful than exposing privacy information.

Designing such a protection system, however, is a difficult task, especially if it is to be *practical*. Any practical protection system must be simple, general and easy to use. Moreover, to be practical, a protection system should require minimal or no changes to the software environment so that it can be applied or deployed easily to existing systems. We have designed a protection system that can directly secure the application information. This protection system, which we call *Software-Privacy Preserving Platform* (*SP*^{3}), provides a simple and general interface, requires only minimal changes to existing systems, and creates an easy-to-deploy trust base. We introduce *software-privacy* as a new protection measure that SP^3 guarantees to applications by encrypting the contents of memory. SP^3 guarantees that a correctly-written application can protect the secrecy of its memory contents.

SP^3 chooses the *information* contained in the memory, not the memory itself, as the target of protection. In addition, it uses a memory *page* as the unit of protection, which is also the unit of memory management. These design choices make it easy to separate protection from management without sacrificing flexibility and generality. SP^3 ensures that an operating system sees only encrypted images of the pages that are private to applications. The operating system, which is usually indifferent to the user's memory content, is not obstructed in managing memory with these encrypted images; that is, protection is "orthogonal" to management. SP^3 achieves a high degree of protection orthogonality, which also allows data secrecy to be an intrinsic property of operating system services. For example, SP^3 turns an unmodified regular file system into a secure one that can write encrypted data to disk.

SP^3 extends the conventional paging system, so an application process sees different memory contents in its virtual address space depending on its execution context. SP^3 guarantees the correctness and safety of the protection mechanism by carefully controlling the keys used to encrypt memory pages: the actual encryption keys are hidden from the operating system. The operating system is only allowed to *associate* each page with a key. Permissions to the keys are completely controlled by individual applications, *not* by the operating system.

We have implemented a prototype system of SP^3 by modifying the Xen hypervisor and Linux to demonstrate the following features of SP^3.

—Using virtualization techniques, SP^3 can be realized as a software layer which is efficient and readily deployable.
—SP^3 can be implemented on a mature modern operating system with minimal intrusion.
—SP^3 provides a broad, general, and easy-to-use privacy protection interface for user applications.

The modified Xen hypervisor implements SP^3 and serves as the trust base for software-privacy protection. Running on top of the modified Xen, the Linux kernel is also modified so as to function correctly by avoiding operations that violate the protection rule imposed by SP^3. If Linux violates the rule, it will at worst crash the system, but the applications's privacy is guaranteed unless the SP^3 itself is compromised. To facilitate application development, we have also built a toolkit that can help create an SP^3 program from an application source code. We evaluated the Xen-based SP^3 realization measuring the runtime performance of SP^3 applications. Our evaluation results indicate that the prototype implementation causes only a 3% slowdown to the applications for CPU- and memory-intensive workloads.

The rest of the report is organized as follows. Section 2 motivates our work by discussing problems with current practices and challenges in protecting application information. Section 3 presents the design principles of SP^3. Sections 4 and 5 detail our SP^3 implementation on the x86 architecture with Xen. We present the impact of SP^3 on operating system and user application programming in Section 6. Section 7 evaluates the performance of our Xen implementation. Section 8 discusses the effectiveness of the SP^3 design, while Section 9 discusses the related work. Finally, the report concludes with Section 10.

2. MOTIVATION

We motivate this work by reviewing problems of current practice in protecting application information (Section 2.1), and then discussing challenges in creating a new protection system for existing computing systems (Section 2.2).

2.1 Problems

The operating system is responsible for providing an execution environment for user applications. It also provides the data privacy of applications through process isolation in which each process is protected from other processes. This requires the applications to trust the operating system, and therefore, the system security depends on how secure the operating system is. The operating system is thus strongly secured and the operating system kernel runs with privilege, wielding unlimited power.

Unfortunately, from the perspective of user applications, a large part of operating system code must be trusted. Moreover, many parts of the trust base, such as core kernel and device drivers, run with privilege. For these reasons, relying on the operating system for application data secrecy has the following problems.

First, there is no effective second-line of defense that applications can resort to in case the operating system is compromised. Many applications need to protect sensitive information, but once an attacker seizes control of the operating system, it is very easy for her to steal information from the application. Any effort to protect the information will be futile as the attacker can exploit the operating system's omnipotence to subvert, reverse-engineer, or simply disable the protection mechanism.

Second, it is very difficult to guarantee the operating system's trustworthiness. Verifying the correctness of an operating system has become intractable as its size and functionality continuously grow to meet the increasing demand for more functionalities and features. Furthermore, the operating system is increasingly built with components from diverse sources.

Third, many negative social and technical side effects arise when someone forces this type of protection by, for instance, using trusted computing. The biggest problem of the current trusted computing approach is that it severely limits the users' freedom to choose and install operating systems or programs, regardless of whether they are certified or not.

These problems can be solved by a new protection system that is not under the operating system's control and that can directly secure the applications' information. This protection system can always guarantee the data privacy of applications even when the operating system is compromised. Therefore, it serves as an effective second-line of defense for applications. For the same reason, the trustworthiness of an operating system does not have to be verified; a malicious or faulty operating system may fail the execution of applications, but it cannot steal or divulge the applications' information. Since the new protection system runs independently, users can choose any operating system and program to run without

risking the exposure of sensitive information.

2.2 Challenges

In general, creating a practical protection system is a challenging task. A careful design is required to create a protection system that is simple, general and easy to use. Moreover, to create a protection system that directly protects the information of applications, we must overcome the following two challenges. First, we must preserve the operating system's usual management power, incurring minimal changes to existing systems. The new protection system may restrict the operating system by enforcing certain rules and hence preventing the operating system from performing operations against the rules. However, the restrictions should not obstruct the operating system from performing legitimate management jobs. Moreover, an existing operating system may not be compatible with the new rules, thus requiring adaptation. The adaptation should entail minimal changes to the operating system. Second, we must find an implementation that is small and simple to verify. With the new protection, the operating system can be verified less stringently, since applications can still be protected even when the operating system fails (as a result of its compromise). However, the mechanism that implements the new protection should be fully trusted, and hence, the correctness of the implemented protection is critical to the security of the entire system.

3. DESIGN

We now present the details of SP^3 design. We begin with the design objectives and principles of SP^3 in Section 3.1. We then define *software-privacy* as the protection model and present SP^3 as the protection system in Section 3.2. An example of how applications interact with SP^3 is illustrated in Section 3.3. Section 3.4 discusses the salient features of SP^3 while Section 3.5 details the alternatives for implementing an SP^3 system.

3.1 Principles

Practicality is our primary concern in creating a new protection system. It is usually difficult to design a protection system since its usability depends greatly on the simplicity and generality of the protection. Many past and current protection systems with complex interfaces have failed to survive, or are simply not used. For instance, although segmentation provides more features and fine-grained control, it has not been chosen for memory protection over the simpler, easier-to-use paging system. A protection system must, therefore, be simple and general so that its benefits can outweigh the accompanying inconvenience and overhead.

We have another practicality-related goal: the protection system must be easily applicable and deployable to existing systems. Any required modifications to an existing system must be 'orthogonal' extensions so that the system may preserve its original function and structure, and the extensions should not restrict the users.

Therefore, we take *simplicity*, *generality* and *orthogonality* as our design objectives, which have led us to the following design principles:

—*Choose a memory page as the unit of protection*: Memory is the prime resource in a computing system. Many other resources are mapped to memory. The protection can be generalized if we can provide protection for memory. In addition, *page* is the fundamental unit of memory management. Many other operating system constructs,

such as file systems, treat pages as their internal or external unit of resource abstraction. We can, therefore, achieve a simple and orthogonal interface by using a memory page as the unit of protection.

—*Protect information instead of resources*: As the target of protection, we choose the *information* stored in a page rather than the page itself, enhancing the generality and orthogonality of protection. This is achieved by encrypting the contents of the page. By allowing an operating system full access (i.e., read/write/relocate) to the pages in their encrypted form, the operating system is not obstructed in managing memory and address space, yet the information stored in the pages is protected.

—*Avoid using operating system abstractions*: Operating system abstractions, such as process and address space, are artificial. At first, it seems easy and simple to use the process and address space as the principal and perimeter of the protection. However, relying on them imposes limitations unwittingly, and actually hurts the simplicity and generality of protection. For instance, inside of the kernel, the notion of address space becomes imprecise. The process abstraction is temporally imprecise especially when it is created. Therefore, independent notions for protection principal and perimeter help enhance the generality and simplicity of protection.

3.2 Software-privacy and SP^3

We propose *software-privacy* as a precise protection model and introduce *Software-Privacy Preserving Platform* (SP^3) as a protection mechanism.

3.2.1 *Software-Privacy.* We define *software-privacy* as the state of a software program being free from information leakage to unauthorized parties. Three aspects of this protection model require clarification. First, software-privacy is primarily concerned with data confidentiality. Second, the protection model does *not* prevent an unauthorized party from accessing protected data, as long as there is no unauthorized information leakage. SP^3 achieves this by encrypting memory page contents and allowing only authorized entities to 'see' a decrypted image. The operating system is allowed to 'access' every page, but without permission, it can only see the encrypted image. Finally, software-privacy is not accountable for the voluntary information leakage by the application itself. Applications are responsible for controlling information flow [Zdancewic et al. 2001] and reducing bugs [Cadar et al. 2006].

3.2.2 *SP^3 definition.* As the principal of SP^3 protection, we use the concept of protection *domain* [Lampson 1971]; 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 process 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 *paging extension* 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 interrupt. Last, the *domain operations* handle the dynamics of domain creation and deletion as well as transferring

access permissions for sharing. Here we only outline the SP^3 constituents, omitting details relevant to actual implementation. The implementation details will be provided in Section 5, describing the implementation of SP^3 on the Xen hypervisor that extends the x86 architecture and runs Linux on top of it.

3.2.2.1 *Paging extension.* The page table entry (PTE) structure is extended to include a new multi-bit field, called KID (*Key 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.

3.2.2.2 *Secure domain switch.* Interrupts and exceptions cause traps into the operating system. Thus, when these events occur, the current SID changes to 0. However, before the operating system takes over control, the execution context of the 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.

3.2.2.3 *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^-) are 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. `Grant` 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.

3.3 SP^3 example

Figure 1 illustrates how SP^3 renders different views of the virtual memory as the current SID changes. In Figure 1, there are three active domains in the system. Two of them,

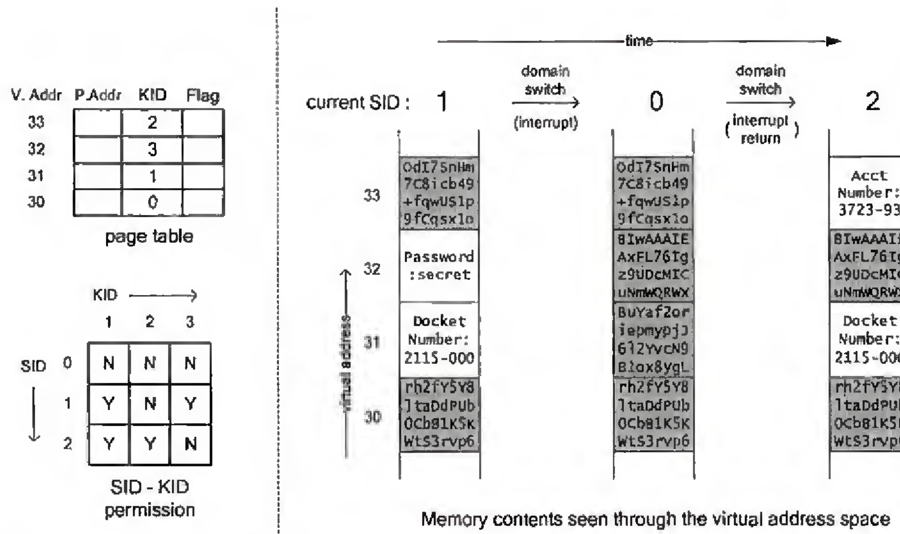


Fig. 1. An example SP^3 system. SP^3 renders different views of the virtual memory as the current sid changes. Domain 1 has permission to KID 1 and 3, thus it sees decrypted images at virtual addresses 31 and 32. Domain 0 has no permission to any keys, thus the memory contents are rendered as verbatim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

Domains 1 and 2, were created by the `Alloc` operation, loading symmetric keys to the system along the operations. The remaining domain is 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 the permission to the KID 1 and 3. Instead, the operating system sees the verbatim page 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 the corresponding PTEs and permission bitmap entries.

We now illustrate how a user application is generated from source code, transferred to a SP^3 -capable host, loaded to memory, and finally, executes on the host. First, an application source code is compiled normally. Compiler/linker tricks are used to align the sections 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 in 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 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. This decryption is done transparently to the application. It requires minimal effort 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.

3.4 Why SP^3 ?

To highlight the advantages of SP^3, we first show the soundness of SP^3 with respect to the above-mentioned design objectives. A more detailed discussion is presented in Section 8. We then demonstrate the effectiveness of SP^3 by showing how it can secure operating system services with almost no additional effort.

3.4.1 *Soundness.* Software-privacy elegantly resolves conflicts between the need for management power and the restrictions imposed by the protection model. With SP^3, the operating system retains virtually all power of managing processes and memory. It is only prohibited from seeing the plaintext content in the protected user memory, which is not required for memory management anyway. The operating system is usually indifferent to the user memory content, but some constructs were implemented to use user memory, such as using a user stack for passing system-call parameters or saving processes' contexts upon signals. They require minor modifications in order not to use user memory.

Simplicity: SP^3 defines a simple and narrow interface, thus minimally affecting existing software as demonstrated next. First, for operating systems, adapting to the KID field only requires an orthogonal extension to the memory-management routines. Second, no compiler modification is needed to create SP^3-enabled applications. *Binutils* and a few compiler tricks suffice for generating application executables. Last, the protection is provided in terms of the data's *position* and the process's *context*. For programmers, providing data privacy in their program is simple and easy since they only have to place sensitive data in the protected memory.

Generality: SP^3 provides broad protection, yet imposes no restriction. Applications can protect the entire virtual address space if they want to, yet important operating system constructs, such as shared memory and disk swapping, apply equally to the protected pages.

Orthogonality: With SP^3, data privacy becomes a universal system property. Operating system services, such as file system and networking, can be secured transparently as detailed next.

3.4.2 *Effectiveness.* The straightforward and intended use of SP^3 is for applications to safely keep sensitive information in the protected memory during their execution. However, SP^3 also provides new ways of doing things secretly. Described below are the various ways of exploiting the SP^3 protection environment.

3.4.2.1 *Secure file system.* Securing files in disks is an important area of research and many secure file systems have been designed [Blaze 1993; Zadok et al. 1998]. However, these systems must be trusted and hence, suffer the same problems of trusting an operating system.

Using SP^3, files can be securely stored without trusting the file system. In fact, SP^3 turns a regular (insecure) file system into a secure one without any modification as we elaborate next. An application allocates memory consisting of pages mapped with KID k, to which the application has exclusive access. The memory is filled with sensitive information. Then, it calls `write` to a file with a pointer to the memory. The file system will eventually read the memory in order to write them to disk, but all it can see is the encrypted image of the memory, since the operating system does not have access to k. Nonetheless, the file system is indifferent to what it sees, and proceeds to write the encrypted image to the disk. SP^3 provides strong file protection against both hardware and software attacks. Information is secured not only from stolen disks or lost memory sticks, but also from a compromised or hijacked file system.

3.4.2.2 *Secure communications.* We can also apply the concept of secure file system to network communications. In such a case, the application passes a pointer to the memory mapped with k to the network stack. The other end of the communication can be either an SP^3-protected application or any other system, as long as it can decrypt the communication packets. However, the network stack might require modifications to support this. Unlike the file system that is a block device, the network is stream-oriented which does not fit the page boundary well. Network stack implementation generally meddles with user packet data in a way that is incompatible to the encryption method. Despite this difficulty, SP^3 has great potential for secure communications. The host-based trust establishment in distributed systems can be replaced with a process-oriented trust, allowing a pool of trusted applications running on untrusted hosts to securely communicate with each other via a privacy-preserving channel.

3.4.2.3 *Encryption service.* When an application needs to encrypt data, SP^3 can serve as an encryption facility as follows. First, the application prepares the data in pages mapped with KID k. Then, an alias map (i.e., a different virtual address) is created on the same physical pages using a null KID. According to the SP^3 page access rule, the application then sees the encrypted data from the aliased memory. This usage is very different from the conventional encryption method which is highly procedural.

3.5 Implementation alternatives

The system that implements SP^3 forms a trust base whose execution must be more privileged than the operating system. In today's computing environment, SP^3 can be realized in three different ways: using a hypervisor (software), direct hardware modification, and a software/hardware hybrid. Next, we present these alternatives and discuss their pros and cons.

3.5.1 *Using a hypervisor.* SP^3 can be implemented using a hypervisor, or a virtual machine monitor (VMM). A hypervisor, positioned between hardware and operating system, is a system software that can create multiple virtualized hardware instances to be multiplexed upon a single physical machine, making it possible to run multiple operating systems concurrently. A hypervisor is also used to realize hardware extensions without ac-

tually changing the real machine, or to implement system services [Chen and Noble 2001] below the operating system. For its own protection, a hypervisor runs with more privilege than the operating system and has a safe perimeter. Therefore, SP^3 can be safely implemented using a hypervisor. Section 5 will detail a prototype SP^3 system that we built by modifying a full-fledged hypervisor.

Using a hypervisor has the following advantages. A hypervisor-based SP^3 system is readily deployable in existing systems as it does not require any hardware modification. Compared to the hardware-based approach, it does not suffer from such machine-specifics as multiprocessing or cache consistency issues since the hypervisor abstracts away such details. The hypervisor can easily support multiprocessor or multi-core systems by sharing the permission bitmap and the key database among processors and by having each processor have its own current SID variable.

The biggest disadvantage of this approach is its poorer performance than the hardware-based counterpart since software has to perform encryption and emulate the extended paging system. Another disadvantage is that the size of the trust base may not be as small as we would like to see, because the hypervisor usually comes with other features. However, we don't need the ability to run multiple operating systems since in our case, a hypervisor is used only for the purpose of implementing a service below the operating system. Therefore, we may reduce the size of trust base by implementing an SP^3 system only with the virtualization techniques required to achieve a safe perimeter.

3.5.2 *Modifying hardware.* One can realize SP^3 by directly modifying the hardware. Obviously, the hardware itself is more 'privileged' than the operating system, and thus, provides a safe and secure implementation base for SP^3. The implementation of the extended PTE field, access control logic, and domain operations is straightforward. For the actual encryption/decryption of memory pages, one may utilize the encryption hardware that can perform two-way encryption on memory-cache boundary [Lie et al. 2000; Suh et al. 2003]: to render the content of a decrypted page, the encryption hardware is activated to fill the corresponding cache line with the decrypted memory content.

The greatest advantage of this hardware-based implementation is its superior performance: hardware can perform much faster cryptographic operations than software. Hardware can also directly support the extended paging interface. Another advantage of this is that hardware can provide a more secure trust base than the software approach.

However, this approach has the biggest practical disadvantage: it cannot be deployed on existing systems due to its requirement of hardware (especially the processor) modifications. Another disadvantage of this is that specific machine-details become highly relevant to the viability of the actual construction of hardware. For instance, encryption based on cache line works only in a uniprocessor system.

3.5.3 *Using a hybrid of hardware and software.* A software-hardware hybrid solution can make best of the advantages of both software- and hardware-based approaches. There are many possible ways in deciding which part should be implemented in hardware or software, but we do not discuss it any further since it is not within the scope of this paper. The disadvantage of this approach is that it still needs hardware modifications, albeit to a much lesser extent than a pure hardware approach.

4. BACKRGOUND: HYPERVISOR AND OPERATING SYSTEM

This section provides background on the internal workings of hypervisor and operating system. This background information is intended to help understand the description of our hypervisor-based implementation of SP^3, which will be presented in Section 5. In this section, we particularly focus on paging (Section 4.1) and interrupt interface (Section 4.2) as they are closely related to the description of our implementation. We summarize with an example (Section 4.3) by stepping through what happens when we execute a program in a virtual machine environment.

In what follows, we assume Linux running on the x86 architecture as the choice of computing platform. We also use Xen as our choice of hypervisor. Despite this choice of specific platforms, our discussion below can be equally applied to many other general processor architectures, operating systems and hypervisors.

4.1 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 [Waldspurger 2002]. 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.

4.2 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, 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 handle the source of the interrupt or exception. Upon completion of handling the interrupt 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 case of para-virtualization like 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.

4.3 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's 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 first vectors to the hypervisor's handler. After determining the fault should be handled by the guest operating system, the hypervisor forwards the fault to the guest operating system. The page-fault handler in the guest operating system then allocates and maps a physical page to the faulting address by updating the corresponding PTE. This update is intercepted by (or submitted to) the hypervisor for its implementation of shadow paging.

5. IMPLEMENTATION

In this section, we detail the design and implementation of hypervisor-based realization of SP^3. Section 5.1 presents our basic design by which an hypervisor can emulate SP^3 semantics efficiently and securely. In particular, we introduce *page-frame replication* and

lazy synchronization schemes for enhancing performance. In Section 5.2, we detail the modifications on the Xen hypervisor for realizing our design.

5.1 Design

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

5.1.1 *Emulating SP^3 paging.* As mentioned earlier, the heart of the SP^3 system is its secure paging semantic, called SP^3 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 and the current SID has the permission to use the KID. We 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 software decryption on the page of typical size 4KB. A naive design would incur a significant run-time performance overhead. We would thus 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 of 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 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 of 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.

[1] An actual hypervisor employs another layer of address indirection by which a "physical address," which is the virtualized memory address local to a virtual machine, translates to a "machine address," which is the physical address of the underlying hardware. To simplify the discussion, we omit this translation layer.

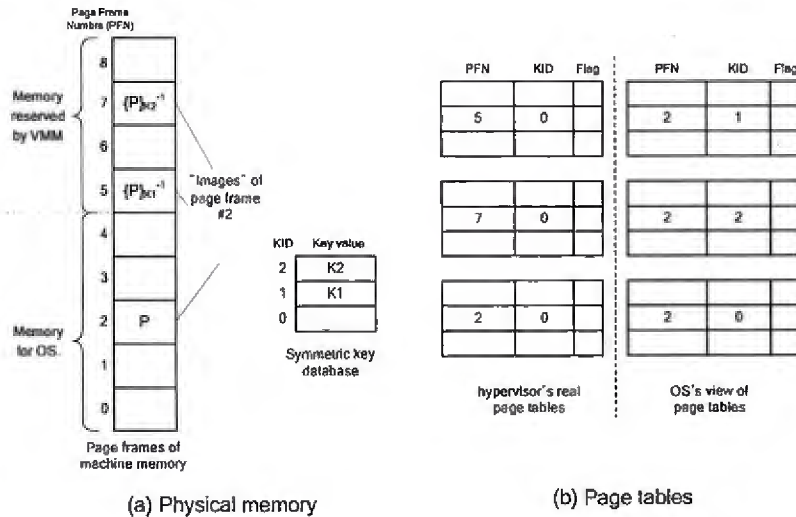


Fig. 2. In Figure (a), a hypervisor keeps decrypted copies of an original page frame (PFN 2) in different memory locations (PFN 5 and 7). The hypervisor uses one of these page frames when the original frame is mapped with a PTE with a KID value. The redirection of page frame is performed transparently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

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 accessed 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.

5.1.2 *Emulating SP^3 secure domain switch.* The secure domain switch defined in SP^3 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 occur. 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 in a hypervisor.

We can realize the transition of current domain by intercepting every interrupt and exception generated by hardware. As discussed in Section 4, hypervisors are, by definition, capable of intercepting all interrupts and exceptions. When the hypervisor forwards an interrupt to a guest operating system, it can change the current domain by setting a variable representing the current SID 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 discussed earlier, 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 has been saved from a previous interrupt. Upon receipt of this request, the hypervisor processes the extended exception frame to restore GPRs and SID value.

5.1.3 *Emulating 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 a 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.

5.2 Implementation

We modified Xen hypervisor [Barham et al. 2003] which runs on top of x86 (IA-32) architecture [Intel Corporation a]. 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 privacy protection domain. Henceforth, Xen's "domain" is referred by 'virtual machine', and we use 'SP^3 domain' or simply 'domain' to refer to our SP^3 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 paging.

5.2.1 *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.

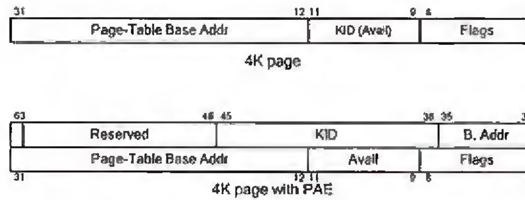
It gets tricky when we make Xen emulate the new extensions to CPU-level interface, specified in Figure 3. The extensions are to reflect the new KID field in PTE structure, and to generate a secure interrupt frame upon interrupt. Obviously, we did not actually modify the hardware; the specification given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

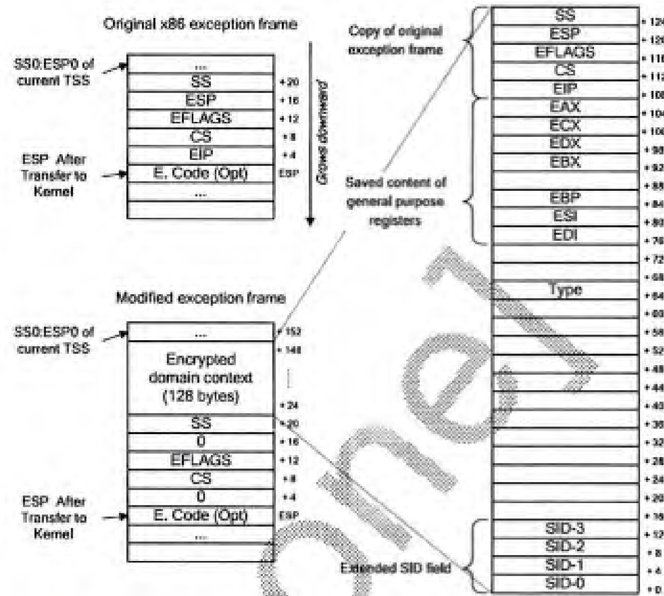
We modified Xen to emulate this PTE extension by adding a code that can interpret the KID field. This code is added to the Xen's handler routine responsible for PTE updates. This handler routine is always invoked when a guest operating system modifies a PTE to map a page. Since MMU updates are sensitive, 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

[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 outside of the scope of this paper. We refer the readers to [Kauer 2007] for safe and secure loading of the Xen hypervisor.

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



(a) Extended x86 page table entry



(b) Modified x86 exception frame

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

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 are 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 128-byte-long encrypted part 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 it 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 non-zero KID value.

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

5.2.2 *Implementation detail of SP^3 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 uses 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 non-traditional 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 doesn't 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 dirty (D) bit of the PTE is checked to synchronize between the two copies. The synchronization entails 4Kb of AES operation which is time-consuming. However, under our lazy synchronization, 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 re-evaluated 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 re-evaluated.

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 re-evaluates 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. OS/APPLICATION SUPPORT

In this section, we discuss the impact of SP^3 on operating system and user application programming. Section 6.1 details how an operating system can be adapted to the SP^3 environment by presenting our modification to Linux. Section 6.2 considers the application programming in the SP^3 environment by discussing how application developers can easily take advantage of SP^3 protection.

6.1 Modifications on Linux

SP^3 introduces a number of changes to the underlying interface that operating systems are built upon. To adapt to these changes, a conventional operating system must be modified.

[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.

The purpose of these modifications is three-fold: (1) support the extension of paging interface, (2) make the operating system function correctly, and (3) support user applications.

First, to support the extension of paging interface, the operating system must modify its memory management logic to consider the KID field in the PTE. The modifications range from the low-level routines that directly modify page tables, such as the page-fault handler, to virtual address space management routines, to high-level memory mapping routines that interface user applications, such as `mmap()`.

We modified Linux, making it run on top of the modified Xen. The KID integration in memory management routines only required an orthogonal extension of the virtual memory routines of Linux. At the lowest level of the virtual memory routines is the page-fault hander. We modified the page-fault hander to recognize and differentiate KID field of PTE structure. At the next level, Linux manages the virtual memory regions allocated to the user address space by means of a data structure known as `vm_area_struct`. Each `vm_area_struct` describes a memory region with the same property in the user's virtual address. We extended this structure to include KID for the region property. At the top of the virtual memory routines, Linux utilizes `mmap()` for the general handling of mapping of memory to virtual address spaces. We modified the interface of `mmap()` to accept KID value as an additional parameter.

Second, for the operating system to function correctly, it must understand the restrictions imposed by SP^3 and then adapt accordingly. The restriction is that the operating system cannot freely read from, or write to the memory for the applications; it can do so, but the data it reads may be encrypted and thus unintelligible. Writing to the application's memory may corrupt the data since the memory may contain unintelligible 'random' data when the application reads the memory back. Another restriction is that the operating system cannot arbitrarily read/modify saved application contexts such as the program counter (EIP), because they are securely saved in the secure domain context (i.e., x86 extended exception frame).

Adapting Linux to these restrictions is easy because Linux, like other operating systems, is indifferent to the contents of the user memory. Only a few parts, such as signal delivery mechanism and program binary loader, meddle with user stack memory, and thus need to be modified as we discuss below.

The complication with the original Linux signal dispatcher in SP^3 originates from the fact that the operating system cannot directly modify the extended exception frame. That is, it is impossible for the operating system to directly specify the return address by modifying EIP of saved exception frame. Thus, for the implementation of a user callback mechanism (e.g., signals) or other cases that require modification of return address in general (e.g., `fork`), the operating system must find ways other than overriding the return address. To solve this problem, we devised a simple yet elegant solution which is to use a *user-level dispatcher*, whose entry point is shown in Figure 4. When the user process first requests installation of a signal handler, the operating system saves the modified exception frame (Figure 3(b)) caused by the system call for later use. The operating system saves the signal handler's address and then returns to the user process with return value 0. When the operating system wants to signal the process later, it uses the saved exception frame but at this time, the return value contains the handler's address. Then, the user-level dispatcher, a thin layer between the system call interface and C library, will *test* the return value and take appropriate actions.

```
/* for initial entry, %ebx is 0 and %ebp is addr to stack
   for signal, %ebx is signal number, %esi is handler  */
_sp3_start:
    cmpl    $0x0, %esp
    jnz     1f                      /* not an sp3 program  */
    movl    %ebp, %esp
    xorl    %ebp, %ebp
    cmpl    $0x0, %ebx
    jnz     3f                      /* signal dispatcher   */
    push    %esi                    /* arguments           */
    jmp     2f
1:  xorl    %ebp, %ebp
    push    %esp
2:  call    __sp3__clear_bss        /* extern C function   */
    call    __sp3__cstart           /* extern C function   */
    push    %eax
    call    _exit
3:  push    %eax                    /* signal parameter    */
    push    %ebx                    /* signal number       */
    push    %esi                    /* handler address     */
    call    __sp3__dispatch_signal  /* extern C fn         */
    popl    %eax                    /* sigreturn sequence  */
    movl    $119, %eax              /* __NR_sigreturn      */
    int     $0x80
```

Fig. 4. The unified entry point for all callbacks to SP^3 programs. The operating system can safely enter to user space only at _sp3_start. Instead of directly entering to the signal handler, the operating system specifies signal number in %ebx and handler address in %esi

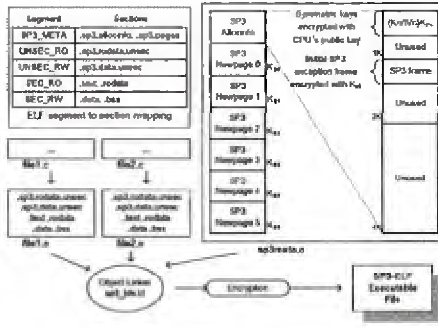


Fig. 5. SP^3 ELF binary construction. The SP^3 specific metadata information is contained in SP3_META section. The actual code and private data which is encrypted by a symmetric key is contained in SEC_RO, SEC_RW section. The unencrypted part of the program that the operating system is allowed to see is contained in SEC_RO, SEC_RW section.

When a signal is delivered, the operating system needs to save the current execution state of the process so as to resume the process after handling the signal. In Linux, this state information is saved on the user stack of the process, which is problematic in SP^3. We solved this problem by saving the state information separately in the kernel memory and makes the user-signal handler use a separate stack for signal.

Linux's signal system is one of the few places where its programming interface is significantly changed: the user program is responsible for signal dispatch, and signals cannot be nested due to the use of a separate stack for signals. However, these differences are not a problem for typical user applications since the user-level dispatcher and all of the start setup requirements can be handled by the C library. Also, few programs rely on nested execution of the signal handler.

Last, to support the user applications, the operating system must define a binary executable format to facilitate program loading from a disk. Although the file system itself requires no modification since the binary image from the disk should be loaded into memory as is, the program binary loader routine should be modified to correctly map the memory containing the binary image to the application address space. In addition, we need to provide additional information specific to SP^3 applications.

To achieve this, we utilized the standard ELF file format to load and execute SP^3 program. Figure 5 shows how an SP^3 application binary can be contained in an ELF file. We modified the Linux ELF binary loader to correctly load this ELF format.
which is detailed next.

6.2 Application programming

In the perspective of application programmers, SP^3 introduces a new programming environment. Using the protection interface of SP^3, software can be guaranteed to protect its privacy. Application programming in SP^3 raises interesting issues, several of which are

discussed next.

Although it is technically trivial to create an SP^3 program, transforming a non-SP^3 program to an SP^3 program requires a change to the program since SP^3 does not protect incorrectly written programs. The analogy to multi-thread programming seems appropriate: running a single-thread program in a multi-threaded operating system does not make the program multi-threaded. The program has to be changed to fully utilize the multi-threaded environment. Given below are the things a programmer must consider when s/he writes an application program with the SP^3 framework.

First, it is the application programmer's responsibility to determine which data to be kept private and which data to be made public. For example, strings used as the `printf` argument are mostly public data, whereas variables holding sensitive information are private. Also, this logical data classification should be realized by physical classification: private data should reside in the memory region mapped with KID, and public data should be located in the memory region with KID 0. Some data can be located statically at compile time, but in other cases, the program may need to move data around dynamically. Systematic solutions from the field of information-flow analysis and control [Zdancewic et al. 2001] can be used to overcome this challenge.

Second, although SP^3 provides an interface for a user application to protect its own code and data, it is entirely the application programmer's responsibility to write a correct code that does not reveal sensitive information.Programmers can utilize static/dynamic information-flow checkers and program integrity verification schemes [Seshadri et al. 2005; Park and Shin 2005] to enhance security.

Last, the initial configuration at the program entry is very different from the original application binary interface (ABI) specification. The signal-delivery mechanism is also different, but a C library modified for SP^3 applications can mask most of these singularities: the entry code shown in Figure 4 is a part of C library.

To facilitate application development and packaging, we have developed a toolchain for creating an SP^3 application. The toolchain consists of script files for compile and link, encryption utilities, and a C library adapted to SP^3 framework. Note that no modification on the platform compiler/linker suite is required.

We avoided modification on compiler/linker by utilizing features of existing tools. For the generation of ELF executable of SP^3 programs, we defined special object file sections to separate encrypted and unencrypted data, as can be seen in Figure 5. Using C attribute syntax, programmers can specify the section to which code and data should belong, thus avoiding use of special compiler. As to the C library adapted to SP^3 framework, we modified dietlibc C library [DietLibC].

7. EVALUATION

In our evaluation on our hypervisor-based implementation of SP^3, 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 discussed in Section 5.1.1?

—What causes performance degradation most?

—How does the performance overhead vary with application's memory access pattern?

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.

7.1 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 paravirtualized 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 were 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 [DietLibC].

7.2 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 implementation 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. We measured and compared the running time of the workloads in two different setups. In the first setup, labeled as 'Without SP^3', normal insecure executables were executed on the unmodified Xen. In the second setup, labeled as 'With SP^3', the encrypted executables were executed on the modified Xen. All measurements were made in boot-clean state, meaning that there were no prior run of the same workload since the system boot.

Figure 6 shows the benchmark results. The performance overhead is presented as a percentage increase in the running time of 'With SP^3' experiments relative to that of 'Without SP^3'. 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.

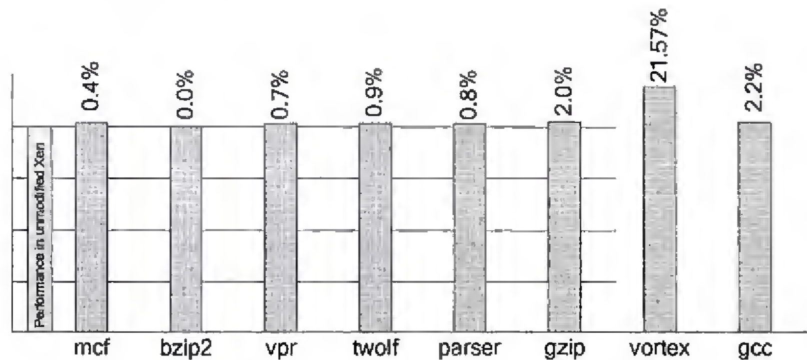


Fig. 6. Application benchmark results. The bars represent the running time of benchmark programs normalized to that of the equivalent programs executed in unmodified Xen environment. The numbers on top of the bars indicate the increased running time in percentage point. Each benchmark result is sorted in increasing order of its static footprint size.

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 re-evaluation. 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.

Our experiment confirmed this effect of footprint. Figure 6 lists benchmark results in an increasing order of the size of the executable. It is found that there is roughly a positive relationship between the footprint and the performance penalty. In Section 7.3, we present a more clear relationship between the runtime footprint and the PTE invalidation penalty.

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 incurs a lot of system calls.

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.

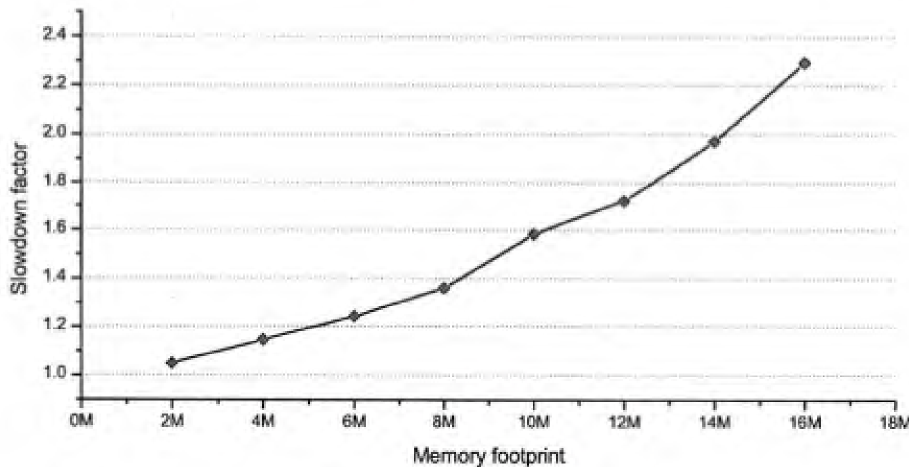


Fig. 7. Impact of memory access locality. X-axis represents dynamic memory footprint of the test program in megabyte unit. Y-axis represents the slowdown factor, which is the increased running time of the test program in SP^3 system normalized to that of the same program in unmodified Xen.

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.

7.3 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 7 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 re-evaluation 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.

8. DISCUSSION

One may raise the following qualitative questions with respect to the design of SP^3.

—How simple is it to realize and use SP^3?

—How general is the protection provided by SP^3?

—What degree of impact does SP^3 make on the existing system?

These questions directly relate to our main design objectives: simplicity, generality, and orthogonality. We answer them by discussing our experience with the SP^3 implementation, and by comparing alternative design choices that SP^3 could have made.

We found it relatively easy and simple to implement SP^3 in terms of the number of lines of code to modify; only 3,000 and 1,100 lines of code are modified for Xen and Linux, respectively. These numbers exclude code for cryptography routines, which is worth 9,300 lines of code. However, the modifications, especially on Linux, are distributed very widely and require to check if a code is compatible with the rules of SP^3. If one doesn't have detailed understanding of code, s/he has to either scan the code in search for violations if any, or look for the "bug" post-mortem. The wide distribution of modifications is the expected difficulty due to the high level of orthogonality of the SP^3 interface.

SP^3 exposes a very simple and easy-to-use interface to user applications that require data secrecy. The SP^3's paradigm of introducing data secrecy in the code is completely different from traditional methods which are highly procedural and intrusive. One usually has to explicitly call a procedure to request a protection service or embed markups in the code to set the protection perimeter. Instead, SP^3 protects data secrecy based on the *location* of data (i.e., the memory page holding the data), and based on the *context* of the domain in which the program executes. The programmer can thus easily blend data secrecy into his data without calling procedures or heavily modifying the code. Besides, management of the protected memory regions via the KID field is neatly integrated into the virtual memory management interface, which is elegant, intuitive, and easy to use. Moreover, coding and compiling SP^3 applications does not require any special compiler or linker.

SP^3 chooses not to use operating system abstractions, such as process and address space. SP^3 defines its own protection domain separately from processes. SP^3 is unaware of address space: whether to render the decrypted or verbatim image of a page to a domain is determined only by the KID of the page, not by the address space of the domain. At a first glance, these design choices seem to introduce an unnecessary layer of abstraction that makes the protection interface and the implementation more complex. However, our design choices actually help enhance simplicity, since the protection is positioned *below* the operating system kernel. If we rely on operating system abstractions, the protection system becomes dependent on the operating system specifics, and has to be modified according to the specifics. For instance, a process is imprecise when it is being created. Until the operating system finishes constructing the process, the process contexts are shared between the creating process and the process being created. However, if a protection system uses process as the principal of protection, it must resolve the temporal sharing during the process creation. Resolving this involves modifications of both the protection system and the operating system, which usually results in a complex and inflexible interface. In contrast, SP^3 defines its own abstraction and does not suffer any of these problems. Although we presented `exec()` as the point of a domain creation (i.e., executing `Alloc`), `Alloc` can actually be executed at any other point, generating many possible design opportunities for further utilizing SP^3 protection. Moreover, by not using address space as the perimeter of protection, SP^3 is designed to support memory sharing, which is one of the fundamental functionalities required by the kernel [Liedtke 1995].

The required operating system modifications are relatively small, simple, and straightforward, compared to those introduced by the generality of protection. Many systems that

change interfaces below the operating system require the operating system itself to adapt. The more general the change is, the more the impact it has on the operating system. As discussed earlier, SP^3 makes very broad and general impacts throughout the system, yet it requires relatively small changes to the structure of the operating system and existing software environment. In this regard, the SP^3's orthogonality helps minimize its impact on the existing system in spite of the generality of SP^3's protection.

9. RELATED WORK

The protection ring [Saltzer and Schroeder 1975; Saltzer 1974] 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. Therefore, a more important kernel part, which is running with more privilege, can still be protected from failure in the less important kernel, which runs with less privilege. Although many processors support multiple privileged rings, they are not widely used except for the layer of hypervisors [Barham et al. 2003; Bressoud and Schneider 1995]. Many other architectures have been proposed to protect the more important kernel part against failures of less important kernel parts [Witchel et al. 2005; Chiueh et al. 1999]. 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.

TCG's trusted computing [TCG], which is an industry standard encompassing chip manufacturers, software vendors, and content industry, has become an umbrella term for related technologies such as Microsoft's NGSCB and Intel's LaGrande [Microsoft ; Intel Corporation b]. Customizing the trusted computing standards using the virtual machine monitor has been explored in Terra [Garfinkel et al. 2003] as well as vTPM [Berger et al. 2006]. By verifying integrity and signature, the Trusted Platform Module (TPM) provides a secure bootstrapping and attestation mechanism to guarantee that applications can elect to run only on vendor-certified hardware and software. In contrast, SP^3 can protect applications without requiring any vendor-certified genuine operating system or hardware. Moreover, the trusted computing does not guarantee the *trustworthiness* of a certified system, meaning that the system can be compromised via undetected exploits. SP^3 can however, protect applications even if the operating system is compromised. TPM is useful for secure bootstrapping. OSLO [Kauer 2007] is a recent implementation of a secure boot loader based on TPM's bootstrapping mechanism. Our system may utilize OSLO to securely boot the modified Xen.

Secure processors are a class of processors with hardware implementation of various cryptographical primitives. Some of them can perform encryption and hashing on the main memory and cache boundary. Some of them can provide this secrecy and integrity protection directly to individual processes bypassing all or most of the operating system. AEGIS [Suh et al. 2003] and Cerium [Chen and Morris 2003] are among them and focus primarily on physical tamper-resistance. However, they both assume a trusted microkernel, but how to compose such a microkernel and why the perimeter is safe are unclear. The XOM secure processor [Lie et al. 2003] can host a fully-untrusted operating system, and can thus protect applications from operating system compromises. However, XOM is not a practical system for the following reasons. Applications must use the special hardware, which is difficult to realize in a multiprocessing system. Moreover, its intrusive and

wide interface renders the system less pragmatic; the protection interface is hard to use, and using the protection restricts many functionalities. For instance, it uses special versions of load/store instructions for accessing secure memory. Since these instructions are general-purpose and frequently-used, heavy compiler/assembler support is needed. XOM also defines too many special instructions to resolve corner-case conflicts, lowering the generality of the architecture.

Many of the approaches to protection of applications' sensitive information can be viewed as code obfuscation [Collberg et al. 1998; Collberg and Thomborson 2002; Linn and Debray 2003]. Unfortunately, obfuscating a program is considered a weak form of protection, as shown by Barak et al. [2001]. In contrast, SP^3 uses cryptography directly for the protection of information.

As a general strategy to enhance system security and robustness, reducing the size of trust base as well as separating trust dependency have been used to solve problems in many areas, such as file system [Zhao et al. 2005], kernel construction [Witchel et al. 2005], application partitioning [Brumley and Song 2004; Zdancewic et al. 2001] and relocating service to the virtual machine monitor [King and Chen 2003; Garfinkel et al. 2003].

Virtual machine monitors are being utilized to solve many system problems. IntroVirt [Joshi et al. 2005] used virtualization to log/replay system events, achieving a perturbation-free intrusion detection system that can also detect past intrusions. Garfinkel and Rosenblum [2003] designed an intrusion detection system based on virtual machine introspection. The proposal of using hypervisor in commodity mobile systems [Cox and Chen 2007] is motivated by the advantage of using hypervisor for implementing security services.

Ta-Min's Proxos [2006] 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 the underlying hypervisor. A set of system calls, which the user can specify, is dynamically forwarded into another operating system instance, which is a full-fledged operating system but untrusted. In contrast, our system provides protection to user memory on a per-page basis, and does not require any private operating system instance.

The negative impact of the size and complexity of a trusted computing base (TCB) on system security has been widely recognized [Hohmuth et al. 2004]. Härtig's Nizza architecture [2002], Singaravelu's AppCore [2006], and IBM's PERSEUS [Pfitzmann et al. 2001] are example efforts to reduce TCB size and complexity.

10. CONCLUSION

Data privacy is a primary security concern for many user-level applications. Traditionally, applications rely on operating systems for data-privacy protection. However, a modern operating system is too large and too complex to secure, thereby making the traditional approach unreliable.

In this report, we have introduced a new system of protection, *Software-Privacy Preserving Platform (SP^3)*, that can directly provide data privacy to applications. Even if the operating system was compromised, SP^3 prevents application information from exposure to unauthorized parties. By extending the paging interface, SP^3 achieves general and easy-to-use data privacy protection that can be implemented in existing systems with minimal intrusion. We have demonstrated that a practical and efficient SP^3 system can be built by modifying a virtual machine monitor.

The protection provided by SP^3 is parallel to the classical protection rings: just as the separation of privilege of a processor serves as a linchpin of operating system protection, the privacy protection of SP^3 serves as an elemental and substratal security platform for application protection.

REFERENCES

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. 2001. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*. 1–18.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, R. N. I., and Warfield, A. 2003. Xen and the art of virtualization. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 164–177.

Berger, S., Cáceres, R., Goldman, K. A., Perez, R., Sailer, R., and van Doorn, L. 2006. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*. 305–320.

Blaze, M. 1993. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS)*. 9–16.

Bressoud, T. C. and Schneider, F. B. 1995. Hypervisor-based fault-tolerance. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 1–11.

Brumley, D. and Song, D. 2004. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*.

Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., and Engler, D. 2006. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*.

Chen, B. and Morris, R. 2003. Certifying program execution with secure processors. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS)*.

Chen, P. M. and Noble, B. D. 2001. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*.

Chiueh, T., Venkitachalam, G., and Pradhan, P. 1999. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 140–153.

Collberg, C., Thomborson, C., and Low, D. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL)*. 184–196.

Collberg, C. S. and Thomborson, C. 2002. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *Transactions on Software Engineering 28*, 8, 735–746.

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

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

Garfinkel, T., Pfaf, B., Chow, J., Rosenblum, M., and Boneh, D. 2003. Terra: A virtual machine–based platform for trusted computing. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 193–206.

Garfinkel, T. and Rosenblum, M. 2003. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS)*. 191–206.

Härtig, H. 2002. Security architectures revisited. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. 16–23.

Hohmuth, M., Peter, M., Härtig, H., and Shapiro, J. S. 2004. Reducing TCB size by using untrusted components – small kernels versus virtual-machine monitors. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*. 22.

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

Intel Corporation. Intel trusted execution technology, http://www.intel.com/technology/security/.

JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. 2005. Detecting past and present intrusions through vulnerability specific predicates. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 91–104.

KAUER, B. 2007. OSLO: Improving the security of trusted computing. In *Proceedings of the 16th USENIX Security Symposium*. 229–237.

KING, S. T. AND CHEN, P. M. 2003. Backtracking intrusions. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 223–236.

LAMPSON, B. W. 1971. Protection. In Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems. *Princeton University*, 437–443.

LIE, D., THEKKATH, C. A., AND HOROWITZ, M. 2003. Implementing an untrusted operating system on trusted hardware. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 178–192.

LIE, D., THEKKATH, C. A., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. 2000. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 169–177.

LIEDTKE, J. 1995. On micro-kernel construction. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 237–250.

LINN, C. AND DEBRAY, S. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*. 27–31.

MICROSOFT. Microsoft, Next-Generation Secure Computing Base, http://www.microsoft.com/technet/archive/security/news/ngscb.mspx.

PARK, T. AND SHIN, K. G. 2005. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Transactions on Mobile Computing 4*, 3 (May/June).

PFITZMANN, B., RIORDAN, J., STÜBLE, C., WAIDNER, M., AND WEBER, A. 2001. The PERSEUS system architecture. Tech. Rep. RZ 3335 (#93381), IBM Research Division, Zurich Laboratory. Apr.

SALTZER, J. H. 1974. Protection and the control of information sharing in multics. *Communications of the ACM 17*, 7 (Jul), 388–402.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (Sep), 1278–1308.

SESHADRI, A., LUK, M., SHI, E., PERRIG, A., VAN DOORN, L., AND KHOSLA, P. 2005. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 1–16.

SINGARAVELU, L., PU, C., HÄRTIG, H., AND HELMUTH, C. 2006. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the EuroSys 2006*.

SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. 2003. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*. 160–171.

TA-MIN, R., LITTY, L., AND LIE, D. 2006. Splitting interfaces: Making trust between applications and operating system configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*.

TCG. Trusted Computing Group, http://www.trustedcomputinggroup.org/.

WALDSPURGER, C. A. 2002. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. 181–194.

WITCHEL, E., RHEE, J., AND ASANOVIĆ, K. 2005. Mondrix: Memory isonation for linux using mondriaan memory protection. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 31–44.

YANG, J. AND SHIN, K. G. 2008. Using hypervisor to provide application data secrecy on a per-page basis. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*.

ZADOK, E., BADULESCU, I., AND SHENDER, A. 1998. Cryptfs: A stackable vnode level encryption file system. Tech. Rep. CUCS-021-98, Computer Science Department, Columbia University.

ZDANCEWIC, S., ZHENG, L., NYSTROM, N., AND MYERS, A. C. 2001. Untrusted hosts and confidentiality: Secure program partitioning. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP). *ACM SIGOPS Operating Systems Review*, 1–14.

ZHAO, X., BORDERS, K., AND PRAKASH, A. 2005. Towards protecting sensitive files in a compromised system. In *Proceedings of the 3rd International IEEE Security in Storage Workshop*.

[none]