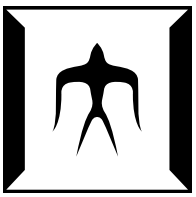


A Dissertation Submitted to Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of Doctor of Science in Mathematical and Computing Sciences

---

Coordinated and Secure Server Consolidation Using Virtual Machines
(仮想マシンを用いた調整可能で安全なサーバ統合)

Hidekazu Tadokoro

Dissertation Chair:
Osamu Watanabe

July 2012

Abstract

Server consolidation using virtual machines (VMs) can improve resource utilization by sharing physical resources. Each VM is isolated from the others for security and VMs can be easily migrated for load balancing. Since there are several VMs in a physical machine, the virtual machine monitor (VMM) multiplexes the physical resources among VMs according to system settings. The administrators determine the system settings and manage the VMs for suspension, resumption, or migration using a privileged VM called the management VM. In this situation, each VM is influenced from VMs coexisting in the same machine. For performance, processes in one VM can compete with processes in other VMs for CPUs because VMs share physical CPUs. For security, due to the privileges of the management VM, sensitive information in the VMs may leak via the management VM. If the attackers intrude in the management VM, they can easily steal sensitive information from the VM's memory.

To address these problems, this thesis proposes coordinated and secure server consolidation. Our VMM provides a system-wide process scheduler called the *Monarch scheduler* and a secure memory manager called *VMCrypt*. The design principle of these systems is reducing the functionalities implemented in the VMM. The Monarch scheduler uses the existing process schedulers in guest operating systems (OSes) as a part of it and changes the behaviors of the minimum number of processes. It mediates CPUs among processes in different VMs to achieve system-wide scheduling policies. To control the execution of processes, it suspends and resumes processes by using a technique called direct kernel object manipulation (DKOM). To hide the details of DKOM for various guest OSes, the Monarch scheduler provides a high-level API for writing scheduling policies. On the other hand, VMCrypt

encrypts the VM's memory only for the management VM and uses the existing management software as is. By the memory encryption, the management VM cannot steal sensitive information in the VM's memory. Although the existing management software can basically run for encrypted memory, it requires unencrypted contents only for several memory regions. Therefore, VMCrypt does not encrypt such memory regions, which are automatically identified and maintained during the life cycle of a VM.

We have implemented the Monarch scheduler and VMCrypt in the Xen VMM. The Monarch scheduler supports not only open-source Linux but also closed-source Windows as guest OSes, and VMCrypt supports paravirtualized Linux, which is tightly coupled with the management VM. From our experimental results, the Monarch scheduler could achieve useful scheduling policies such as idle-time scheduling even in multi-OS environments. The overheads incurred by the Monarch scheduler were small enough. Using VMCrypt, the administrators could perform VM management including live migration securely and correctly. VMCrypt prevented the administrators in the management VM from finding cryptographic keys and passwords in the VM's memory. The downtime due to live migration was still less than one second and its overhead was 13%.

Acknowledgements

光来先生には、何から何までお世話になりました。駄目な学生を指導するのは、大変な苦労があったと思います。感謝しています。また、東工大時代にお世話になった千葉先生、主査を務めて頂いた渡辺先生に感謝します。論文を書くにあたり、審査委員である、佐々先生、渡辺先生、脇田先生、首藤先生、千葉先生、光来先生には、重要なコメントを頂きました。

光来研と千葉研のメンバーに感謝します。近くに人がいるだけで安心しました。千葉研時代は、土日や休日に赤井君や武山君が研究室に来ただけでほっとしたのを覚えています。

卒業研究を始めたとほぼ同時期に、ニコニコ動画のサービスが始まったと記憶しています。ニコ動でりっちゃんの魔法をかけて！を聞きながら、作っているシステム (後に Monarch Scheduler と呼ばれるようになる) をデバッグするのは楽しかったです。研究の横にはいつもニコニコ動画がありました。

ニコニコ生放送を見はじめたのは、博士進学を決めたときでした。後悔で胸がいっぱいの僕が逃避できる場所でした。ニコ生を生活の中心にすることで、不安や劣等感から逃げました。

雪白さんは、僕の希望でした。名札を付けての上野公園は、遠足みたいで楽しかったです。

福岡が好きです。福岡は第二の故郷だと思っています。福岡空港に着くたび、“帰ってきた” と感じます。ラーメンはあまり好きにはなれなかったですが、魚介類がおいしいです。

所属が東工大なのにもかかわらず福岡に住んでることで、いろいろ余分な事務処理が発生したと思います。大学や内定先の事務の方など、普段はあまり意識しないだろう方々の存在を強く意識しました。とても感謝しています。

体の調子が悪いことが、よくありました。そんなときにすぐに医療機関に行けるのは、とても恵まれていると感じました。診察して頂いた医師と医療の進歩、国民皆保険制度に感謝しています。

僕がOSやシステムソフトウェア、プログラミング言語に興味を持つようになる上で、平野さんの存在はとても大きかったです。使っているソフトウェアで分らないことがあれば、すぐにコードを読み始める姿が印象的でした。コードを読む重要さを学びました。

母には、謝りたいです。痩せていく姿を見るのが辛くて、最後はほとんど会えませんでした。ごめんなさい。でも「大学くらいは出ないとね」という言葉は強く覚えています。なんとか、大学くらいは出られたようです。

最後に、父に感謝します。たくさん心配をかけたと思います。どんな言葉でも足りませんが、感謝しています。

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Server consolidation is widely applied to improve the resource utilization of each server machine. Particularly, the virtual machine (VM) technology is promising for consolidating legacy systems. Multiple physical servers can be easily migrated to multiple VMs using physical-to-virtual conversion (P2V) tools. The administrators of the VMs can continue to use legacy systems including legacy operating systems (OSes) as is in VMs. In addition, VMs can be easily moved to other machines if necessary. Such VM migration is used for hardware maintenance and load balancing. Furthermore, VMs are strongly isolated from each other for security. Even if the system in a VM crashes or a VM is compromised by the attackers, the other VMs are not influenced by the VM.

Since there are multiple VMs in a physical machine, the virtual machine monitor (VMM) multiplexes the physical resources among VMs, such as CPUs, disks, and networks. The VMM is a software layer underlying VMs. To configure resource allocation to VMs, the administrators of physical machines often use a privileged VM called the *management VM* in type I VMM such as Xen [70], which directly runs on hardware. For CPU resources, they can determine how much CPU time can be consumed by each VM. For example, the credit scheduler in Xen allows the administrators to assign a proportional share of CPUs called a weight and an upper limit called

a cap to each VM. In addition, the management VM is used for managing VMs such as suspension, resumption, and migration. Management software in the management VM accesses the memory, disks, and networks of VMs to achieve VM management.

1.1 Motivating Problem

In server consolidation, each VM is influenced from VMs coexisting in the same machine. From the performance viewpoint, processes in one VM can compete with processes in other VMs for CPUs even in different VMs because VMs share physical CPUs. For example, consider that a process such as AntiVirus is configured to run only at idle time [16]. When multiple VMs exist in one physical machine, the process would run even when the VM in which it runs is idle but the other VMs are not. As a result, the process may prevent the execution of more important processes in other VMs. Such a situation can be avoided by exclusive allocation of physical CPUs. Allowing each VM to occupy several physical CPUs could prevent CPU contention among VMs. However, exclusively-allocated CPUs are not used at all while the VM is idle. This is unacceptable for server consolidation because one of the motivations of server consolidation is to improve CPU utilization of physical machines.

Since physical CPUs have to be shared among VMs for high CPU utilization, system-wide process scheduling is necessary to schedule processes across VMs. When VMs share physical CPUs, busy VMs can use them even if there are idle VMs. In compensation for this flexibility, the above problem of CPU contention arises. A system-wide process scheduler can solve this problem by monitoring and controlling all the processes in all VMs properly. For example, the process that runs only at idle time is scheduled only when any processes are not running in all the VMs. As a result, more important processes can be executed without being affected by less important processes.

The VMM underlying VMs is a possible place of implementing such a system-wide process scheduler. Since the VMM manages all the VMs and mediates physical CPUs, the sole process scheduler in the whole system could control the execution of all the processes in all VMs. However, it requires large modification to guest OSes because the process schedulers in guest OSes have to be moved into the VMM. To enable the VMM to schedule the processes, guest OSes also have to be modified to give process information to the VMM. The process is the concept in an OS and the VMM cannot

directly recognize the process. The necessity of such modification makes it difficult to apply system-wide process scheduling to legacy systems.

From the security viewpoint, on the other hand, the privileges of the management VM can be security flaws, so that sensitive information in VMs can leak via the management VM. Using the abilities of the management VM, the attackers inside it can steal the whole contents of the memory, disks, and networks of the VMs in the same physical machine. For example, they can dump the whole memory of the VMs into their disks. They can mount the disks of the VMs and read all the files. Network sniffers running in the management VM can easily capture packets to/from the VMs. Fortunately, disks and networks could be encrypted by the guest OSes themselves in the VMs to protect their contents. The VMs can use encrypted file systems like Windows EFS and virtual private networks (VPNs).

Unlike disks and networks, memory is crucial because it is difficult for the guest OSes to encrypt their own memory. Without hardware support, neither OSes nor applications can run with encrypted memory. There are various types of sensitive information in memory [79]. Clear-text passwords and cryptographic keys can be extracted from the VMs' memory. Such information resides in buffers in the kernel or processes temporarily. In addition, the memory usually contains the buffer cache, which is used for maintaining copies of the data of file systems in memory. Through the analysis of the buffer cache, the attackers can read the data blocks of specific files if the blocks are on the buffer cache. The encryption of the file systems in the VMs is not sufficient because the buffer cache cannot be encrypted.

Therefore, the confidentiality of the VMs' memory should be preserved while the administrators perform VM management. A simple solution to prevent information leakage is to disable the management functions of the management VM, but this would not be acceptable under server consolidation. At least, live migration, which migrates VMs with negligible downtimes, is indispensable for the reasons of load balancing and power saving. Another solution is to put management software in the VMM like VMware vSphere [102]. Only the trusted management software in the VMM can securely access the VMs' memory, whereas the management VM does not have the privileges for accessing it. However, this results in a larger trusted computing base and lower manageability. The management software embedded in the VMM enlarges the VMM and this may make the whole system vulnerable. In addition, the management software in the VMM cannot be easily changed. It is difficult for the administrators to use other management software.

1.2 Solution by This Thesis

To address these two problems, we propose the *Monarch scheduler* [24][25][26] and *VMCrypt* [27]. The design principle of these systems is reducing the functionalities implemented in the VMM because the VMM should be smaller for security, known as the principle of least privilege. We put only the minimum mechanisms in the VMM. To achieve system-wide process scheduling, the Monarch scheduler changes the behaviors of the minimum number of processes. It does not implement the full-fledged process scheduler in the VMM, but it uses the existing process schedulers inside guest OSes as a part of it. To prevent information leakage from the management VM, VMCrypt encrypts the memory of VMs only for the management VM. It does not implement the whole management software in the VMM, but it runs the existing management software in the management VM securely.

Monarch Scheduler

The Monarch scheduler is a system-wide process scheduler running in the VMM. It mediates CPUs among processes in different VMs according to system-wide scheduling policies. To achieve such custom scheduling policies, it monitors the execution of processes in VMs and changes the scheduling behavior of guest OSes. The Monarch scheduler exploits the existing process schedulers inside guest OSes as part of it. It also uses the existing VM scheduler in the VMM. Each process is basically scheduled by a process scheduler in the VM that it belongs to. The VM is scheduled by the VM scheduler. The role of the Monarch scheduler is to watch the whole system consisting of multiple VMs and slightly change process scheduling in guest OSes so that a system-wide scheduling policy is achieved.

To control the execution of processes, the Monarch scheduler suspends and resumes processes by using a technique called direct kernel object manipulation (DKOM). It manipulates run queues of process schedulers in guest OSes or rewrites the state of processes because the VMM cannot control processes of guest OSes with their standard interface. At that time, it guarantees that kernel data are manipulated consistently. To obtain the execution time of each process, the Monarch scheduler measures the CPU time used for the execution in the context of the corresponding virtual address space. Unlike a process, its virtual address space is uniquely identified by the page directory in the VMM. Then, the Monarch scheduler binds virtual address spaces to real processes by using process information in guest OSes. Since DKOM is

OS-specific and needs deep knowledge of guest OSes, the Monarch scheduler provides a high-level API to hide the details of DKOM. Using this API, the developers of scheduling policies can suspend and resume processes in various guest OSes transparently.

We have implemented the Monarch scheduler in the Xen VMM. The Monarch scheduler currently supports not only open-source OSes such as Linux but also closed-source OSes such as Windows. Supporting closed-source OSes is achieved by the fact that the Monarch scheduler does not require modifying the source code of guest OSes. This is suitable to consolidate the existing systems into VMs using P2V tools. From our experimental results, it was shown that the Monarch scheduler could achieve useful scheduling policies. Also, the overheads due to the Monarch scheduler were small enough such as idle-time scheduling even in multi-OS environments.

VMCrypt

VMCrypt is a secure memory manager running in the VMM. It encrypts the memory of VMs only when the management VM accesses the memory. It preserves the data secrecy of the VMs' memory by providing a dual memory view: a *normal view* and an *encrypted view*. A normal view is provided for each VM and is not encrypted as usual. An encrypted view is the memory view that is generated by encrypting a normal view, and it is provided for the management VM. Thanks to an encrypted view, the management VM cannot steal information from the VM's memory, whereas the VM can run normally with a normal view. These two views coexist so that both VMs can concurrently access the VMs' memory to achieve live migration, which requires that the management VM transfers the VM's memory while the VM accesses its memory. The existing management software can basically run for the encrypted memory. For VM migration, it can simply transfer the encrypted memory to destination hosts. For several memory regions, however, it requires accessing their unencrypted contents.

To allow the management VM to inspect such memory regions, VMCrypt exceptionally gives a normal view to the management VM only for specific regions. Examples of such regions are memory shared between a VM and the management VM and the page tables inside a VM. VMCrypt automatically identifies such unencrypted memory regions by monitoring the interaction between the management VM, the VMs, and the VMM. For example, a shared memory region is passed from the management VM to the VM via the VMM. The memory regions for the page tables are registered to the VMM.

As such, the VMM can recognize all the unencrypted memory regions. The information on the identified memory regions is cached during the life cycle of a VM, including VM migration, to preserve the compatibility with the existing management software. With the cache, the VMM can restore the VM's memory correctly even at another host.

We have implemented VMCrypt in the Xen VMM. VMCrypt currently supports para-virtualized Linux as guest OSes, which is customized for Xen and aggressively interacts with the management VM. Our experimental results show that VMCrypt allows the administrators to manage the VMs securely and correctly. With VMCrypt, the administrators in the management VM could not find cryptographic keys or passwords from the VM's memory; otherwise, they could do. Nevertheless, they could boot, suspend, resume, and migrate VMs as usual. They could perform even live migration for running VMs. The execution performance of VM management was degraded mainly by the overheads of cryptographic operations. However, the downtime due to live migration was still less than one second and its overhead was 13%.

1.3 Position of This Thesis

Position of the Monarch Scheduler

The position of the Monarch scheduler is that the Monarch scheduler provides high controllability of processes and high compatibility with legacy systems, as illustrated in Figure 1.1. It can monitor all the processes in all VMs and change their behaviors as necessary to achieve system-wide scheduling policies. Since it manipulates only data in guest OSes using DKOM, it does not require adding code for a system-wide process scheduling to the legacy systems in VMs. If physical CPUs could be allocated to VMs exclusively, higher controllability could be achieved by the VM scheduler with no modification to the systems inside VMs. This is not an option under server consolidation because CPU utilization lowers when there are idle VMs. However, when physical CPUs are simply shared among VMs for high CPU utilization, the controllability becomes very low.

To achieve high controllability under shared CPU allocation, the approach of running the full-fledged process scheduler in the VMM is suitable. If all process schedulers in guest OSes are moved into the VMM, the integrated process scheduler can schedule all processes in the whole system strictly.

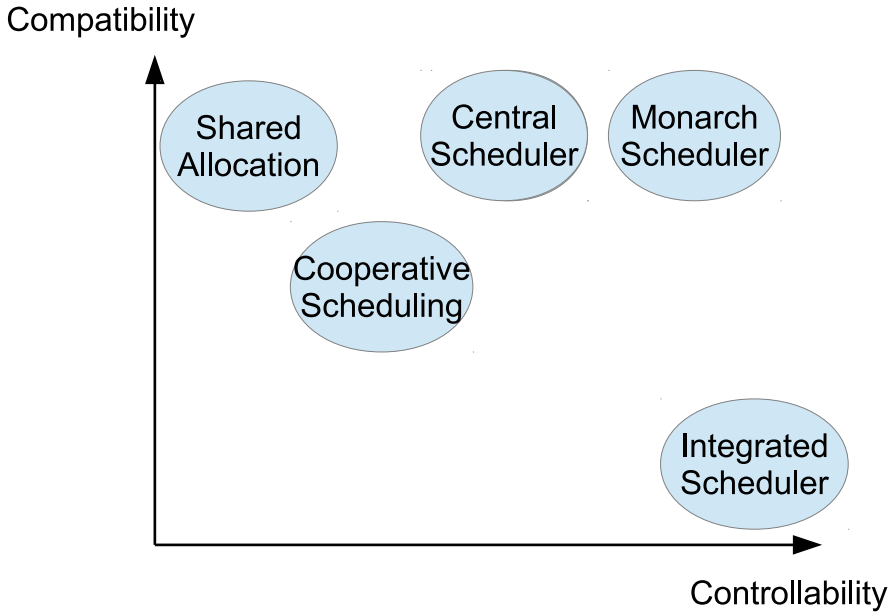


Figure 1.1. The Position of the Monarch scheduler.

This exceeds the Monarch scheduler in terms of controllability because the Monarch scheduler controls processes only periodically. However, this approach requires large modification to every guest OS.

There are two approaches that do not rely on the VMM. One approach is to run a central scheduler in the management VM. The central scheduler monitors processes in the other VMs and change their behaviors. If it uses DKOM like the Monarch scheduler, high compatibility is achieved. However, the controllability is lower than that of the Monarch scheduler because it is difficult to control processes frequently due to high overhead of DKOM from the management VM. The other approach is cooperative scheduling, in which the systems in VMs exchange process information with each other and adjust process execution according to a system-wide scheduling policy. Cooperative scheduling requires adding dedicated scheduler threads in all VMs and treating them specially in system-wide process scheduling. This results in lower compatibility and controllability.

Position of VMCrypt

The position of VMCrypt is that VMCrypt achieves two goals, security and manageability, at the same time as illustrated in Figure 1.2. VMCrypt pre-

vents information leakage via the management VM by encrypting the VM's memory even if the management VM is compromised by the attackers. In spite of the memory encryption, VMCrypt allows the administrators to use the existing management software in the management VM. To achieve high manageability, Xen is suitable because the management VM can access any regions of the VM's memory. For example, management software in the management VM can scan the VM's memory for virus checks. In VMCrypt, it is difficult to perform such memory scan due to memory encryption. However, Xen does not provide any security mechanisms for accessing the VM's memory by the management VM.

To achieve high security, self-management by guest OSes is suitable. Self-management allows guest OSes to migrate themselves [35], for example. It is not necessary to give the privileges of accessing the VM's memory to the management VM. This means that the attackers in the management VM cannot steal any information including even encrypted one from VMs. However, self-management has to assume that the memory state of guest OSes reaches a fixed point before migration. This lowers the manageability of VMs. Another approach of using no management VM is to embed management software in the VMM like VMware vSphere. Management software in the trusted VMM manages VMs by accessing their memory. This can prevent information leakage from the management VM, but the overall security may lower due to a larger VMM. In addition, manageability also lowers because management software in the VMM cannot be changed easily.

1.4 The Structure of This Thesis

From the next chapter, we present background, design details, and the implementations of the Monarch scheduler and VMCrypt. The structure of the rest of this thesis is as follows:

Chapter 2: Background

This chapter gives an overview of existing issues in server consolidation. W We discuss related work and the several approaches to prevent these problems.

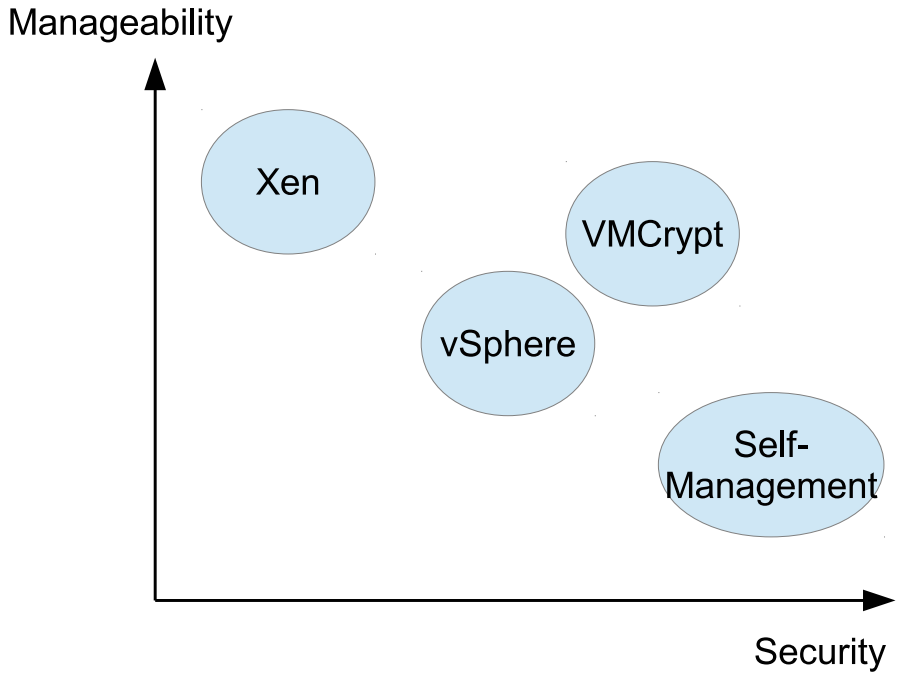


Figure 1.2. The Position of VMCrypt.

Chapter 3: Monarch Scheduler

To schedule processes in VMs, we propose the Monarch scheduler running in the VMM. The Monarch scheduler enables the administrators to adjust the execution of processes in the whole system when multiple servers are consolidated using VMs.

Chapter 4: VMCrypt

Chapter 4 presents VMCrypt to prevent information leakage from the VM's memory. We describe its design and how to prevent information leakage, and then describe the implementation details.

Chapter 5: Experiments for the Monarch Scheduler

In this chapter we describe the experiments for the Monarch scheduler.

Chapter 6: Experiments for VMCrypt

In this chapter we describe the experiments for VMCrypt.

Chapter 7: Conclusion

Finally, we conclude this thesis in Chapter 7.

Chapter 2

Background

2.1 Process Scheduling in a VM Environment

Recently, multi core processors become popular, but the CPU resource would be still shared among VMs. The number of cores is likely to be more than that of VMs already. In such a situation, it is not necessary to share cores among VMs and cores can be exclusively allocated to each VM. Nevertheless, sharing cores among VMs may be more efficient. If several cores are allocated to one VM exclusively, they are not used at all when the VM is idle. If they are shared with other VMs, they can be used even when some of the VMs are idle. In a VM environment, the allocation of the CPU resource is hierarchical. The VMM allocates the resource to VMs by VM scheduling. In each VM, a guest OS allocates it to processes by process scheduling.

To optimize the performance of the whole system consisting of VMs, each process should be scheduled in the whole system according to a system-wide scheduling policy. For example, let us consider that two VMs are used for a web system. A front-end VM runs the Apache web server and a back-end VM runs the Tomcat application server [96]. The front-end VM also runs Tripwire [44, 99], which is an intrusion detection system, because it is subject to attacks from the outside. Tripwire should run without largely affecting the whole system, but it degrades the performance of not only Apache in the

same VM but also Tomcat in the other VM. To prevent such performance degradation, a required scheduling policy is that the Tripwire process runs in the lowest priority among all the processes in the whole system.

However, it is difficult to schedule processes among VMs in the hierarchical scheduling. A process scheduler in each guest OS does not allow system-wide scheduling because it locally schedules only the processes inside one VM. In other words, it cannot prioritize processes in the other VMs. For example, a process scheduler in the front-end VM could give a lower priority to the Tripwire process than the Apache processes. By this local prioritization, Tripwire does not affect the performance of Apache, but the Tripwire process uses the whole CPU time allocated to the VM when all the Apache processes are idle. If the CPU time is allocated to these two VMs equally, the Tripwire process in the front-end VM can run in the same priority with the Tomcat processes in the back-end VM. As such, the required scheduling policy is violated.

On the other hand, a VM scheduler in the VMM is ineffective for such process scheduling. The VMM cannot recognize the process because it is the concept in an OS. A VM scheduler can give priorities or reserve the CPU resource only to VMs. For example, it can give a lower priority to the front-end VM than the back-end VM. As a result, the priority of the Tripwire process in the front-end VM becomes lower than the others. At the same time, however, the priority of the Apache processes also becomes lower than that of the Tomcat processes. It is desirable that Apache and Tomcat run in almost the same priority to optimize the total performance of the web system. If we could elaborately configure both the VM scheduler and the process schedulers in all VMs, the required scheduling policy might be achievable. Even if it is possible, the policy may be easily violated when one process simply becomes idle.

2.1.1 Related Work

Researchers have proposed VM scheduling mechanisms that can give global priorities to processes across VMs. In guest-aware VM scheduling [42], each guest OS notifies the VMM of the highest priority among processes. In proportion to the notified priority, the VM scheduler adjusts the priorities of VMs. Since the VM scheduler considers only the highest priority in each VM, the other processes in the same VM can take too much CPU time. In task-grain scheduling [45], on the other hand, each guest OS notifies the hypervisor, the L4-embedded microkernel, of the priorities of all processes.

Whenever a context switch occurs, the hypervisor schedules the guest OS running the process with the globally highest priority at that time. For regular VMMs such as Xen, switching VMs so frequently causes large overheads. Unlike the Monarch scheduler, these scheduling mechanisms require the modification of guest OSes.

In task-aware VM scheduling [43], on the other hand, the VM scheduler preferentially schedules VMs that execute I/O-bound processes without modifying guest OSes. The scheduler detects I/O-bound processes in VMs by using the same technique as Antfarm [37] and other gray-box knowledge [1]. When a network packet arrives to the VMM, the scheduler immediately schedules the VM where a process to receive the packet exists. This mechanism is for better VM scheduling, not for process scheduling.

Geiger [90] is a technique to get the state of the buffer cache in guest OSes without modifications from the VMM. Like as Antfarm [37] or other gray-box knowledge [1], the VMM can obtain some information of the guest OSes. This is not a technique to schedule processes. FoxyTechnique [29]and FoxyLargo [94] uses a gray-box knowledge to trick guest OSes with a VMM.

A system-wide process scheduler can be also implemented using a technique similar to coordinated scheduling, which is used in distributed systems [69]. A local scheduler running in each VM obtains information on processes in the whole system by communicating with the other VMs. Then each local scheduler controls the execution of processes in the same VM. Since legacy OSes cannot be modified, local schedulers are often implemented as processes, using techniques for user-level scheduling [65, 66]. However, if the attackers compromise a local scheduler, they may easily perform DoS attacks by telling a lie to the other VMs. In addition, the process time recorded inside VMs may be inaccurate, as described in Section 3.1.

Virtual machine introspection (VMI) is a technique of inspecting guest OSes from the VMM. For example, Livewire [21] enables executing intrusion detection systems in the outside of a VM. It examines the internal state of the OS kernel in a VM using VMI. To recognize kernel data structures, Livewire uses debug information of the OS kernel like our Monarch scheduler. In addition to inspecting kernel data by using VMI, the Monarch scheduler modifies them by using DKOM. IntroVirt [38] enables obtaining more information such as file contents from the VMM by executing kernel functions in a guest OS. To prevent kernel data from being modified by the execution of kernel functions, IntroVirt uses checkpoint and rollback. By contrast, the aim of the Monarch scheduler is to modify kernel data to change the behavior of a guest OS. As such, typical applications of VMI are security while the

Monarch scheduler uses VMI (and DKOM) for scheduling. The Monarch scheduler is another kind of application of VMI.

Virtuoso [6] is a system that automatically generates introspection software. The generation consists of three phases, training, analysis, and run time. In training phase, a small in-guest training program is executed repeatedly in the guest OS to retrieve information of the guest OS as instruction traces. For example, A process that only gets the handle of a process is executed. In analysis phase, the trace analyzer analyzes the log to generate a code for introspection. There are some limitations in the system. One is that the generated code is not always accurate, because traced logs are retrieved only when the program is executed, a user land. To avoid this limitation, introspection code can only executed when the guest OS is in a user mode. Another is that this is not for process scheduling. It is impossible to run a scheduler as a training program. So it is unsuitable to implement a system wide process scheduler using introspection.

VMwatcher [36] detects malwares by comparing information obtained in a VM with that obtained from the VMM using VMI. For example, if the number of processes is different between the result of the `ps` command and that of traversing the process list in the kernel, some processes are hidden. As a guest OS, VMwatcher supports not only Linux but also Windows. VMwatcher and our Monarch scheduler use the same technique for inferring process objects inside the Windows kernel [8]. The Monarch scheduler, furthermore, infers a run queue from inferred process objects.

Lares [72] executes security applications in another VM by inserting hooks into the code of a guest OS from the VMM. Since the hooks are protected by the VMM, they can securely call the VMM at arbitrary points and transfer the control to a security VM. Security application in the security VM inspects the guest OS using the XenAccess library [71], which supports the Windows guest OS. XenAccess provides an API similar to the low-level API provided by the Monarch scheduler, but it does not provide any API for modifying a guest OS. Lares achieves active monitoring by using hooks while most of other VMI systems including the Monarch scheduler achieve only passive monitoring by using polling. Lares may not be able to coexist with kernel integrity checkers because it modifies the code of a guest OS.

There are other libraries to obtain the internal structure of Windows guest OSes, EagleEye [62], VIX [7]. The Monarch scheduler cannot use them in two reasons. First, it is a library available only in domain 0 of Xen. In general, domain 0 is a reasonable place to inspect domain Us. However, it is not suitable for a system-wide process scheduler because of its overheads.

Domain 0 has to map memory pages into its address space to access the memory of domain Us. Second, XenAccess does not support to modify guest OSes. There are also many researches to use the internal structure of guest OSes, such as SBCFI [61], PsycoTrace [2].

Direct kernel object manipulation (DKOM) [9] is a technique that manipulates data in guest OSes by directly modifying the kernel memory. This technique has been often used for attacks. We use this technique for changing the scheduling behavior of guest OSes from the VMM. It was challenging to change the behavior of guest OSes by using only the DKOM technique because this technique can change it only indirectly through the modification of kernel data.

In user-level scheduling [65] and ALPS [66], a scheduler process periodically controls the execution of processes in an OS to achieve custom scheduling policies. It suspends and resumes a process by sending signals such as `SIGSTOP` and `SIGCONT` to it, respectively. These approaches require no modification of the OS kernel and exploit process scheduling by the OS as much as possible like the Monarch scheduler. Using these approaches, the Monarch scheduler can make its scheduler process control the execution of processes in domain 0.

Credit Scheduler [103] is a proportional fair share CPU scheduler in Xen. It is now the default scheduler in Xen. It can use SMP effectively. In Credit Scheduler, parameters `WEIGHT` and `CAP` is assigned to each domain. Weight describes relative CPU allocation. For example, if the weight of domain A is 512 and one of domain B is 256, the domain A is assigned 2 times CPU of the domain B. Cap describes the limitations of CPU assignment. The domain cannot use any more CPU than the cap.

Borrowed-Virtual-Time (BVT) [41][14] is a scheduling policy used in the Xen VMM. This scheduler is used as the default scheduling policy in the Xen 2.0 and 3.0. Each domain has a specific minimum time, and a domain running during the time is guaranteed to be never preempted by other domains. The BVT scheduling distributes weighted CPU time among domains according to the system policy while it provides applications requiring the real time and low latency. In addition, it can be implemented as low overheads among multi-processor environments. Atropos is another scheduling policy that can be used in Xen 2.0. This originates from Earliest Deadline First, a real time scheduler developed in Cambridge University. This is used in multimedia OS [30].

XenFIT [74] checks the file system consistency dynamically. Not like previous works, it does not need to create or update any databases such as

file hashes. It is implemented in Xen, domain 0 checks system calls invoked by targeted guest OSes. Due to checking system calls, it can detect file updates, file modifications, and changing attributes of certain directory. In order to check system calls, it sets break points at the code of system call tables in the kernel memory, traps the invocation of system calls. It uses debug information of the kernel. However, it is a security software, does not schedule processes. In addition, it does not modify any memory of guest OSes. XenKIMONO [75] is a intrusion detection system for OS kernels. It can checks the consistency of kernel from outside. However, it is also a security software, does not schedule processes.

2.2 Security Issue in the Virtualized Environment

2.2.1 The Management VM-related Information Leakage

Server consolidation using VMM has been widely accepted. Particularly, the VM technology is promising for consolidating legacy systems. Multiple physical servers can be easily migrated to multiple VMs using physical-to-virtual conversion (P2V) tools. The administrators can continue to use legacy systems including OSes as is in VMs. In addition to VMs to run consolidated OSes, There is the privileged VM on a VMM, called the management VM, for the administrators to manage the OSes.

However, each VM is influenced from VMs coexisting in the same machine. For performance, processes in one VM can compete with processes in other VMs for CPUs. For security, sensitive information in the VM's memory can be stolen by the privileged VM for managing VMs if the management VM is compromised

Due to the privileges of the management VM, sensitive information in the user VMs may leak via the management VM. From users' point of view, the administrators in server consolidation are not always trustworthy [83, 48, 105]. Lazy administrators may allow outside attackers to intrude in the management VM. Worse, administrators themselves may be malicious and perform insider attacks. Such attackers can easily steal sensitive information from the user VMs. However, it is difficult to prohibit the access to VMs' memory for the management VM because the administrators have to manage the user VMs, e.g., VM migration, by accessing their memory. This inherently leads to information leakage from the user VMs.

In this situation, sensitive information in user VMs can leak via the man-

agement VM. Using the abilities of the management VM, the attackers inside it can steal the whole contents of the memory, disks, and networks of the user VMs on the same physical machine. For example, the attackers can dump the whole memory of the user VMs into their disks. They can mount the disks of the user VMs and read all the files. Network sniffers running in the management VM can easily capture packets to/from the user VMs. Fortunately, disks and networks could be encrypted by the guest OSes themselves in the user VMs to protect their contents. The user VMs can use encrypted file systems like Windows EFS and virtual private networks (VPNs).

Unlike disks and networks, memory is crucial because it is difficult for the guest OSes to encrypt their own memory. Without hardware support, neither OSes nor applications can run with encrypted memory. There are various types of sensitive information in memory [79]. Clear-text passwords and cryptographic keys can be extracted from the user VMs' memory. Such information resides in buffers in the kernel or processes temporarily. In addition, the memory usually contains the buffer cache, which is used for maintaining copies of the data of file systems in memory. Through the analysis of the buffer cache, the attackers can read the data blocks of specific files if the blocks are on the buffer cache. The encryption of the file systems in the user VMs is not sufficient because the buffer cache cannot be encrypted.

Therefore, the confidentiality of the user VMs' memory should be preserved while the administrators perform VM management. A simple solution to prevent information leakage is to disable the management functions of the management VM, but this would not be acceptable in server consolidation. At least, live migration is indispensable for the reasons of load balancing and power saving with negligible downtime. In particular, it is challenging to support para-virtualized guest OSes because the management VM has to inspect and modify the user VMs' memory. Although full virtualization is being used, para-virtualization is still useful in terms of efficiency. In addition, any modification to the existing management software should not be required because it is not realistic to modify various management software.

2.2.2 Related Work

There are several studies for preventing information leakage via the management VM. CloudVisor [105] runs a security monitor underneath the VMM using nested virtualization. It encrypts all the memory pages of the user VMs whenever the management VM accesses them. Therefore, it is difficult to support para-virtualization because CloudVisor does not allow the man-

agement VM to access pages necessary for VM management in user VMs, e.g., the P2M table in Xen. The security monitor cannot recognize such pages by hardware events such as VM exit. Although CloudVisor supports VM migration for full virtualization, there are no performance data. In addition, CloudVisor introduces extra overheads to user VMs due to an additional virtualization layer. It does not trust even the VMM, but we believe that we can also trust the VMM if we can trust the security monitor via remote attestation.

The secure runtime environment (SRE) [48, 49] preserves the confidentiality of user VMs for para-virtualized OSes. Like VMCrypt, SRE encrypts memory pages except several pages necessary for VM management when the management VM accesses them. SRE supports boot, suspend, and resume of user VMs. However, it cannot enable live migration because it provides an encrypted view to the management VM only when a user VM is not running. Moreover, SRE needs to modify the existing management software so that the VMM can identify all the unencrypted pages. For example, the resume program has to notify the VMM of the pages used for the P2M table because the VMM cannot identify them at the resume time.

Confidentiality Protection [89] preserves the confidentiality of users' information. Using multi cloud providers through an entity called service broker enables the identity anonymization that no cloud provider can know who uses the cloud. The service broker properly chooses the cloud. It also obfuscates users' to protect users' data. It is divided into software cloud and infrastructure cloud. The software cloud provides software service as program but does not execute it. The infrastructure cloud executes the program obfuscated by the software service attestation authority in order to the infrastructure cloud cannot know what the user does now.

In VMware vSphere Hypervisor (ESXi) [101], the VMM itself includes the management functions and the system administrators perform VM management by sending commands directly to the VMM. Unlike VMware ESX, the management VM called the service console is not used by default. Lacking the service console makes it difficult to steal information from the user VMs' memory. However, this architecture lowers the flexibility of the VM management. It is not easy for the administrators to use their own custom management software.

Domain disaggregation [60] moves management functions in the management VM to another VM called DomB. This architecture reduces the privileges of the management VM and prevents the attackers in the management VM from stealing sensitive information in the user VMs' memory.

For example, most of the code for building VMs is run in DomB. However, the administrators have to be still trusted because they also manage DomB as well as the management VM. In addition, domain disaggregation needs a large modification to the existing management software so as to use the functions provided by DomB.

Like VMCrypt, Overshadow [11] provides a dual memory view using the VMM. The difference is that Overshadow provides a normal view for applications and an encrypted view for the guest OS. The aim of Overshadow is to prevent information leakage from applications to the OS. In contrast, VMCrypt prevents information leakage from the user VMs to the management VM. To enable processes to issue system calls, Overshadow needs to inject small code into process address spaces. VMCrypt does not need to run any programs inside the user VMs.

SP^3 [104] is similar to Overshadow and provides different memory views for an OS and processes. Unlike Overshadow, SP^3 can also give different views for the memory shared between processes. However, SP^3 requires the modification to an OS because it extends page table entries to implement different memory views. VMCrypt needs no modification to the OSes in both the user VMs and the management VM. Like VMCrypt, SP^3 replicates memory pages for different views and synchronizes them as lazily as possible. In addition, it can set access permissions on a per-page basis. VMCrypt also allows the management VM to introspect user VMs' memory on a permission basis.

The trusted cloud computing platform (TCCP) [83] guarantees the confidentiality and integrity of the user VMs in IaaS clouds. It allows the user VMs to be launched on and migrated to only trusted nodes, using the trusted coordinator (TC). Before a user VM is run on a node, the TC attests the node to verify that the node is running the trusted VMM. After the user VM starts running, the VMM prevents information leakage from the VM to the management VM at runtime. The TCCP assumes the system proposed in the above domain disaggregation as an example of the trusted VMM. VMCrypt also uses the TCCP for node and key management, as an example of the trusted VMM.

For other resources, several systems have been proposed for preventing information leakage. BitVisor [84, 4] transparently encrypts disks and networks at the VMM level. The outside attackers cannot steal sensitive information from disks physically or from network packets. Since BitVisor does not provide the management VM, it is difficult for the administrators to steal information from the user VM under the trusted VMM. However,

	by the VMM	by the management VM	by user VMs
Secure	Yes	No	No
Accurate	Yes	No	No
Easy to Manage	Yes	Yes/No	No
Easy to Implement	No	Yes	No

Table 2.1. The comparison of approaches.

this architecture has two drawbacks. First, the VMM has to contain device drivers, which often have vulnerabilities. Second, the VMM has to provide mechanisms for VM management such as suspend and migration.

VPFS [10] enables trusted applications to use a secure file system constructed on untrusted storage. It runs untrusted legacy OS, which manages untrusted storage, in a VM. Trusted applications directly on the microkernel can access the storage via the untrusted VM. The confidentiality and integrity of the file system are preserved by encryption and hash functions. If VPFS is applied to a general VMM, the management VM is an untrusted VM and a user VM runs trusted applications. The user VM can securely access the file system on untrusted storage managed by the management VM, for example, domain 0 in Xen.

2.3 Hypervisor-based Approach and OS-based Approach

We discuss the hypervisor-based approach and the OS-based approach to implement the system-wide scheduler and the secure management for VMs. The Table 2.1 shows the summary of each approach. We can summarize that our approach trades the easiness to implement by the management VM to the security by the VMM. It is the acceptable trade-off because the security is more important than the difficulty to implement the system.

2.3.1 Cooperative Scheduling

One possible solution is cooperative scheduling among process schedulers in all VMs. In cooperative scheduling, each process scheduler shares process information with the others and controls the execution of its own processes

based on a system-wide scheduling policy. Technically, it is easier that a dedicated scheduling VM gathers information from all the other VMs and sends scheduling commands to them for prioritizing processes among VMs. For example, the front-end and back-end VMs send the CPU times used by the Apache, Tripwire, and Tomcat processes to the scheduling VM periodically. If the Tripwire process uses longer time than the others, the scheduling VM sends a scheduling command for suppressing the execution of Tripwire to the front-end VM. Then the process scheduler in the front-end VM suspends the Tripwire process for a while or lowers its priority.

A disadvantage of such cooperative scheduling is to need dedicated scheduler threads (or processes) in all VMs. A scheduler thread obtains process information, communicates with the scheduling VM, and controls the execution of processes. Such a thread requires special treatment in scheduling policies. If the priority of a scheduler thread itself is lowered or, worse, the thread is suspended due to a scheduling policy, cooperative scheduling does not work well. Similarly, some other processes may be also treated specially. For example, a VM may not seem to be idle due to `syslogd` [92, 80, 67, 91], which runs whenever a scheduling thread prints debug messages. Therefore, To prevent such a situation, it is necessary to develop scheduling policies that are aware of the scheduling threads. In addition, each guest OS has to be modified to run a scheduler thread. If a scheduler thread is implemented as a kernel thread, the kernel is modified. Even if it is implemented as a userland process, it may require several new interfaces for obtaining information from the kernel and controlling the execution of processes.

2.3.2 A Central Process Scheduler in the Management VM

Another form of cooperative scheduling is a scheduler running in the management VM as a process. The Central scheduler communicates with all VMs to monitor and manipulate their processes as described in Section 5.10. The Central scheduler can be also implemented in the management VM as a process, we call the process as the scheduling process. For idle-time scheduling, for example, the Central scheduler periodically obtains the CPU times used by all processes from all VMs. If all processes are idle, the Central scheduler sends a scheduling command for executing a background process to the corresponding VM. Then the process is resumed in the VM.

This scheduler has the same problem of the cooperative scheduling. It is also need dedicated scheduler threads or process in all VMs. This requires the special treatments of the scheduling processes in any scheduling policy. If

the scheduler process suspend the processes, the whole scheduling is stopped. Moreover, the management VM should be treated as a special VM. For the scheduling process to run correctly, the management VM is always allocated enough CPU times. The lack of CPU times for the management VM may cause the delay of scheduling due to the scheduling process may not run.

2.3.3 Encryption by the guest OSes

To prevent information leakage from guest OSes' memory, there are several candidates for who to encrypt guest OSes' memory. One is that letting the guest OS to encrypt themselves and the management VM can access it for migration. If the guest OS encrypts a memory page of itself accessed by the management VM, that can prevent the information leakage from memory. In this situation, guest OSes can easily decide unencrypted memory page, which pages should be encrypted or not since the guest OSes know uses of every memory. However, for the management VM to manage VMs, it should access whole memory of the VMs. In migration or suspension, it reads all memory of the VMs continuously and sends to other host or writes into disk. To protect secrecy information in memory, OSes themselves encrypt the memory accessed by the management VM. If the management VM accesses all memory of user VM at once, guest OSes must encrypt whole memory of theirs at once, however, it is impossible in the normal hardware such as PC because encrypted memory cannot be executed directly by the commodity CPU. Another problem is that if the management VM pauses the user VMs, guest OSes in the VMs cannot encrypt any memory of the guest OSes. This becomes an obstacle to maintain hardwares.

2.3.4 Managing VMs by themselves

Another is disable the hypervisor and the management VM to manage VMs. For example, Self-migration [35] migrates guest OSes entirely without hypervisor involvements or the management VM, letting guest OSes migrate themselves. Since the guest OS knows its state, such as the the state of the page table and process scheduler, it can track the memory to copy its memory to the destination host. The kernel thread of the guest OS copies its memory and transfers the memory with its network stack. It allows the guest OS to migrate while privileged access by the management VM is no longer needed. They can also use their own TCP/IP stack with ssl, so it

prevents information leakage from memory by the management VM. However, it is hard to implement because the last state of the memory can not be easily captured by itself because if it captures the last state, then the state may change due to the capturing itself. To solve this, self-migration waits for the working set to be identical. Once the state reaches the identical working set, the fixed point, it will deduce the every state of the guest OS. One drawback of self-migration is to modify guest OSes. To access memory state of guest OSes, it requires the special kernel thread or the same functional special devices. This architecture hardens other managements such as suspension, resumption, and booting. In the hypervisor-base approach, VMM or the management VM creates VMs, suspends them, and resumes them as well. However, the OS-based approach cannot conceded these managements, since there is a big difference between the VM migration and other VM managements. The VM migration guarantee that the one VM instance exist anywhere at least, so that instance continues self managements such as migration. Other managements do not guarantee that the one VM instance exist at least. If the VM is suspended, there is no VM instance anymore. There is no entity to decide when to resume or to execute resume the suspended VM.

Chapter 3

Monarch Scheduler

To solve the above problem of process scheduling in a VM environment, we propose the Monarch scheduler running in the VMM. The Monarch scheduler enables the administrators to adjust the execution of processes in the whole system when multiple servers are consolidated using VMs.

3.1 Overview

The Monarch scheduler is a system-wide process scheduler that can equally deal with processes in all the VMs. As illustrated in Figure 3.1, it monitors the execution of processes and changes the scheduling behavior of guest OSes to achieve its custom scheduling policies. The Monarch scheduler exploits the existing process schedulers inside guest OSes and the existing VM scheduler in the VMM as much as possible. Each process is basically scheduled by a process scheduler in the VM that it belongs to. The VM is scheduled by the VM scheduler. The role of the Monarch scheduler is to watch the whole system consisting of multiple VMs and slightly change scheduling policies in guest OSes so that a system-wide scheduling policy is achieved.

The Monarch scheduler provides a high-level API. To avoid the processes' competition for CPUs, the administrators can develop custom scheduling policies using the high-level API. For example, a policy for system-wide idle-

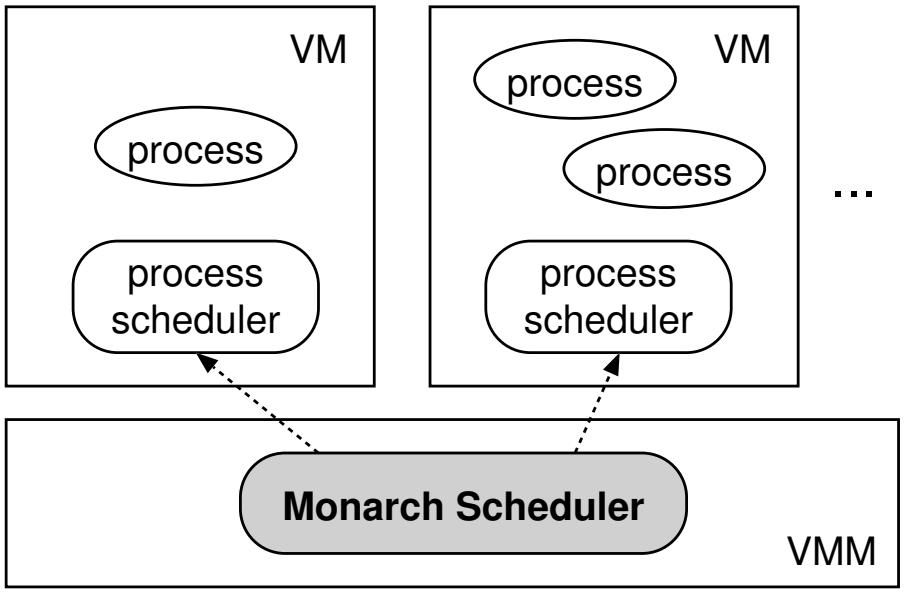


Figure 3.1. The Monarch scheduler running in the VMM.

time scheduling can be achieved by the Monarch scheduler. To allow the low priority task to run only when the whole system is idle, the administrators can write such policy for the Monarch scheduler. This policy improves the performance of the other high priority task even in the other VM. The high-level API is an interface for monitoring and controlling the execution of processes without deep knowledge of guest OSes. As long as the developers use the high-level API, they are almost unaware of the differences between guest OSes such as Linux and Windows and even the existence of multiple VMs.

To change scheduling policies in guest OSes, the Monarch scheduler uses a technique called direct kernel object manipulation (DKOM). DKOM is a technique that manipulates data in the OS kernel by directly modifying the kernel memory. This technique has been often used for attacks. For example, the attackers remove their malicious processes from a linked list for processes to hide their processes. We use this technique for changing scheduling behavior of guest OSes from the VMM. The Monarch scheduler recognizes kernel objects using type information obtained from the debug information of the OS kernel. It is challenging to change the behavior of guest OSes by using only the DKOM technique because this technique can change it only indirectly through the modification of kernel data.

To suspend and resume a process, the Monarch scheduler manipulates a run queue of a process scheduler in a guest OS or rewrites its state, using the DKOM technique. If a process is ready in a run queue, the Monarch scheduler removes it from the run queue to suspend it. Since a process scheduler in a guest OS allocates the CPU resource to one of the processes in its run queue in turn, this manipulation prevents a target process from being scheduled. The Monarch scheduler makes sure that a guest OS is not manipulating its run queue and keeps the consistency in the OS. For a process waiting for an event or the currently running process, the Monarch scheduler changes its state to suspend it. By rewriting the state, a waiting process is not scheduled when it is woken up. Also, the current process is removed from a run queue when it uses up its time slice. To resume a process, the Monarch scheduler inserts it into a run queue.

One advantage of using DKOM is to modify no source code of guest OSes. This enables the scheduler to control not only open-source OSes such as Linux but also closed-source OSes such as Windows. In fact, the Monarch scheduler supports Linux and Windows currently. Another advantage is to change no control flow inside guest OSes at runtime. This is important to coexist with integrity checkers of kernel code. Without using DKOM, code instrumentation is another useful technique for changing the behavior of guest OSes [93]. For example, there is another possible implementation to replace some instructions in the OS kernel by ones for jumping to instrumented scheduling code. Using code instrumentation enables executing arbitrary code at arbitrary points. However, some integrity checkers fail to check kernel code because code instrumentation modifies kernel code. The Monarch scheduler can coexist with integrity checkers because it uses only DKOM.

Running a system-wide scheduler using only DKOM in the VMM does not require modifying source code of guest OSes in advance. This enables the scheduler to control not only open-source OSes such as Linux but also closed-source OSes such as Windows. In fact, the Monarch scheduler supports Linux and Windows currently. In addition, a system-wide scheduler makes it easier to change scheduling policies than in cooperative scheduling described in Section 2.1. Cooperative scheduling requires to modify all guest OSes to cooperate with each other. Since the Monarch scheduler runs only in the VMM, it requires no special processes or threads inside guest OSes for cooperation. The Monarch scheduler itself collects information from all guest OSes directly and achieves cooperation among guest OSes.

The Monarch scheduler records the execution time of processes in the VMM. The VMM cannot understand processes in guest OSes directly, but

it can recognize them via their virtual address spaces [37]. The Monarch scheduler monitors the switches of virtual address spaces and measures the CPU time used by processes. Although the process time is also recorded by guest OSes, it may be inaccurate when processes are running in VMs. As far as an OS kernel is not modified to be aware of the VM, it cannot recognize the context switches between VMs. As a result, it may incorrectly account the CPU time to a process even while the VM for it is not scheduled. The measurement by the VMM does not depend on the accounting mechanisms in guest OSes.

3.1.1 Scheduling API

The Monarch scheduler provides a high-level API. The administrators can develop custom scheduling policies using the high-level API. Its high-level API allows the administrators to be almost unaware of the existence of multiple VMs and the differences between guest OSes. The high-level API is an interface for monitoring and controlling the execution of processes without deep knowledge of guest OSes. As long as the developers use the high-level API, they are almost unaware of the differences between guest OSes such as Linux and Windows and even the existence of multiple VMs.

Domain Object The domain object represents a group of VMs. The `get_domain_by_name` function takes a VM name as its argument and returns a domain object. For a VM name, the developers can specify a regular expression. If it matches multiple VMs, the domain object represents all of them. This is useful when scheduling policies deal with multiple VMs transparently. If the developers specify an exact name including no wild cards, the created domain object represents only one VM. The other function for creating a domain object is `get_domain_by_domid`. Using this function, the developers can specify one VM by its ID.

Task Object The task object represents a group of processes. The `get_task_by_name` function takes a domain object and a process name as its arguments and returns a task object. If there are multiple processes with the same name, the created task object represents all of them. The developers can also specify a regular expression for a process name. In addition, if the specified domain object represents multiple VMs, the created task object may represents multiple processes among several VMs. The `add_tasks` function takes two task objects and returns a new task object representing

all the processes included in the two task objects. Similarly, the `sub_tasks` function returns a task object representing processes that are included in the first task object but are not included in the second one.

If the developers want to distinguish each process, they can use the `get_task_by_pid` function. This function takes a process ID instead of a process name. It creates a task object representing a process with the specified ID. Note that the specified domain object has to represent only one VM.

Scheduling Function The `sched_loop` function is used to register a custom scheduling function to the Monarch scheduler. This function takes a function as its argument and invokes the registered function at regular intervals. The Monarch scheduler regards the interval as a *quantum*, which is a scheduling unit. The developers can configure the duration of a quantum by the `set_quantum` function.

Controlling Tasks The `suspend` function executes the suspend operation for a specified task. If the task object represents a group of processes, the function suspends all the processes together. If some of the processes are already suspended, this function does not change their states. Likewise, the `resume` function executes the resume operation for a specified task.

Monitoring Tasks The `set_period` function notifies the Monarch scheduler to record the CPU time used by each process for a specified period. The period is specified by the amount of quanta. The `get_time` function returns the CPU time used by a specified task for a specified period. If the specified task represents a group of processes, this function returns the sum of the CPU time used by these processes. If the specified period is less than or equal to the value configured by `set_period`, the function returns the CPU time used only for that period. For example, when the configured period is 10, the developers can obtain the CPU times used for one quantum and 10 quanta. This is useful for considering the CPU utilization for both short and long periods.

3.1.2 Example Scheduling

We show two example scheduling using the high-level API of the Monarch scheduler.

Proportional-share Scheduling the Monarch scheduler can achieve proportional-share scheduling like Figure 3.2 to solve the problem that we described in Section 2.1. The `init` function creates two domain objects for the front-end VM and the back-end VM. The front-end VM runs the Apache web server and the back-end VM runs the Tomcat application server. The front-end VM also runs Tripwire , which is an intrusion detection system for examining file systems, because it is subject to attacks from the outside. Tripwire should run without largely affecting the whole system, but it can degrade the performance of not only Apache in the same VM but also Tomcat in the other VM.

Then it creates three task objects for the Apache and Tripwire processes in the front-end VM and the Tomcat processes in the back-end VM. The task objects for Apache and Tomcat are concatenated to one for dealing with these processes together. The `schedule` function is periodically invoked by the Monarch scheduler. It obtains the CPU time used by these processes since the scheduler function was invoked last. To assign a lower priority to Tripwire, the function attempts to allocate the CPU time so that the ratio of Tripwire to a set of Apache and Tomcat is $1 : 4$. If Tripwire used more CPU time than that ratio, it is suspended. Although the function needs much more control to achieve practical proportional-share scheduling.

Idletime Scheduling The indexing service is often executed at a idle time [22, 54, 56]. To allow the indexing service in Windows to run only when the whole system is idle, the Monarch scheduler can achieve simple idle time scheduling [16]. The indexing service maintains an index of most of the files to improve the performance of the file search. A problem in a VM environment is that the service starts running whenever one of the VMs becomes idle. In Figure 3.3, the `init` function creates one domain object for all VMs. Then it creates two task objects for all the processes in all VMs and for processes named `SearchIndexer.exe` in all VMs. The `schedule` function periodically obtains the sum of the CPU time used by all processes and that used by the indexing service. If any processes except the index service use CPU time, the indexing service is suspended. The parameter P specifies a *preemption interval* [16], which is the period until the indexing service starts running after the whole system becomes idle.

Possible Scheduling The Monarch scheduler enables controlling the execution of processes on average in a longer period than process schedulers in guest OSes. It may violate a scheduling policy in a short period, but it can achieve

```
#define P 10
task_t p_tw, p_ap, p_tc, p_web;

void init() {
  dom_t d1 = get_domain_by_name("front");
  dom_t d2 = get_domain_by_name("back");

  p_tw = get_task_by_name(d1, "tripwire");
  p_ap = get_task_by_name(d1, "httpd");
  p_tc = get_task_by_name(d2, "tomcat");
  p_web = add_task(p_ap, p_tc);

  set_period(P);
}

void schedule() {
  time_t t_tw = get_time(p_tw, 5);
  time_t t_web = get_time(p_web, 5);

  if (t_tw * 4 > t_web)
    suspend(p_tw);
  else
    resume(p_tw);
}
```

Figure 3.2. Proportional-share scheduling for allocating the CPU resource in a ratio of 1 : 4.

scheduling policies in total. This is because the aim of the Monarch scheduler is to slightly change the execution of several processes and most of process scheduling is left for guest OSes. Therefore, it is unsuitable for scheduling policies that need exact control of the execution, such as real-time scheduling.

3.1.3 Hybrid Scheduling

In addition, system-wide process scheduling has a security issue to be considered. It is inherently vulnerable to a new type of DoS attacks. Let us consider that the attackers compromise one VM and intrude it. The attackers may be able to perform DoS attacks against processes in other VMs only by running specific processes. For example, they can prevent the execution of file

```
#define P 10
task_t p_all, p_si;

void init() {
  dom_t d_all = get_domain_by_name(".*");

  p_all = get_task_by_name(d_all, ".*");
  p_si = get_task_by_name(d_all, "SearchIndexer.exe");

  set_period(P);
}

void schedule() {
  time_t t_all = get_time(p_all, P);
  time_t t_si = get_time(p_si, P);

  if (t_all - t_si > 0)
    suspend(p_si);
  else
    resume(p_si);
}
```

Figure 3.3. Idletime scheduling for running the indexing service when the whole system is idle.

indexing imposed on idletime scheduling by running one process with a busy loop. A system-wide scheduler stops file indexing due to the process running in the compromised VM. Although the administrators should elaborate scheduling policies that can tolerate this type of DoS attacks, such policies become complicated and error-prone. Traditionally, performance isolation provided by VMs could prevent such attacks. If there are processes to be ready to run in a VM, the VM can receive a certain CPU time. System-wide process scheduling breaks such performance isolation.

To mitigate inherent DoS attacks introduced by system-wide process scheduling, the Monarch scheduler provides not only system-wide process scheduling but also *hybrid scheduling*. The hybrid scheduling periodically switches two modes: controlled and autonomous. In the *controlled* mode, the Monarch scheduler performs system-wide process scheduling. In the *autonomous* mode, it stops its own scheduling and allows the VMM and guest OSes to perform their own original scheduling. Even if the attackers in

compromised VMs run malicious processes so that victim processes in other VMs are suspended by the Monarch scheduler, such DoS attacks are mitigated thanks to the autonomous mode. The victim processes can run for a certain period at least.

The hybrid scheduling can achieve both system-wide scheduling and performance isolation among VMs. The controlled mode allows process execution that is not aware of the isolation enforced by VMs. The autonomous mode mitigates DoS attacks across VMs by the enforcement of performance isolation. To take trade-off between two modes, the Monarch scheduler allows the administrators to adjust the ratio of scheduling between these two modes. As scheduling time in the controlled mode becomes longer, the Monarch scheduler can control process execution more accurately. As that in the autonomous mode becomes longer, performance isolation among VMs is left more largely.

Our threat model is as follows. We assume that the attackers intrude some of the VMs and perform DoS attacks specific to system-wide scheduling. We do not assume that the kernels of guest OSes and the VMM have been compromised because the VMM and security hardware can guarantee their integrity. [21, 61, 73, 100]

3.1.4 Security

The attackers that intrude into VMs may be able to affect the Monarch scheduler. If they elaborately alter pointers or page table entries that the Monarch scheduler refers to, the scheduler can crash. Fortunately, the crash affects only the Monarch scheduler itself because the scheduler is implemented as a userland process. When the Monarch scheduler crashes, it is just restarted. At that time, it resumes the processes that it has suspended in guest OSes and schedules processes from scratch. The processes to be resumed can be determined by an additional flag of the data structure in guest OSes. To prevent crashing again, the Monarch scheduler can stop the VM that makes it crash and notify the administrators of such an event.

The attackers in VMs might be able to exploit vulnerabilities of the Monarch scheduler. If they could overflow a buffer in the Monarch scheduler, they might take over the control of the scheduler. The intruders are prevented from controlling the whole domain 0 by confining the Monarch scheduler to a dedicated VM or container like Jail [39]. However, they can still steal information from the other VMs or modify them because the Monarch scheduler has to have such privileges. Restricting the interface for accessing VMs could

mitigate the damages, but it is future work.

The attackers in VMs can deceive the Monarch scheduler by modifying the kernel. For example, if they create another process scheduler in a VM, the Monarch scheduler could not control the processes in the VM. Since this needs kernel modification such as the timer-interrupt handler, kernel integrity checkers can easily detect the attack. Besides, the attackers could alter the value of used CPU time in process information. If monitored process information is not correct, the Monarch scheduler could not achieve expected scheduling policies. Such alteration can be detected by comparing the CPU time recorded in the guest OS with that in the VMM using the technique of Antfarm [37].

3.2 Implementation

We have implemented the Monarch scheduler in Xen 3.4.2 [3]. In Xen, a regular VM is called *domain U*. In the current implementation, the Monarch scheduler supports the Linux 2.6 guest OS and Windows Vista for the x86-64 architecture.

3.2.1 Scheduler Overview

The Monarch scheduler is invoked by timer interrupts in the VMM. The default interval is 10 ms in the current implementation. First, the Monarch scheduler pauses all virtual CPUs to stop the execution of all domain Us. This prevents the conflicts of process scheduling between the Monarch scheduler and guest OSes. During the controlled mode in hybrid scheduling, the Monarch scheduler traverses the process lists in guest OSes and finds target processes. Based on the execution time of the processes, it changes process scheduling in guest OSes by suspending or resuming some of the processes. Finally, it continues all domain Us again. During the autonomous mode, the Monarch scheduler does nothing.

When the Monarch scheduler switches between the controlled and autonomous modes, it largely changes process scheduling in guest OSes. When it enters the autonomous mode, it resumes all the processes that have been suspended by it. All processes are scheduled by process schedulers in guest OSes as if the Monarch scheduler is not used. When the Monarch scheduler comes back to the controlled mode, it re-schedules all processes again

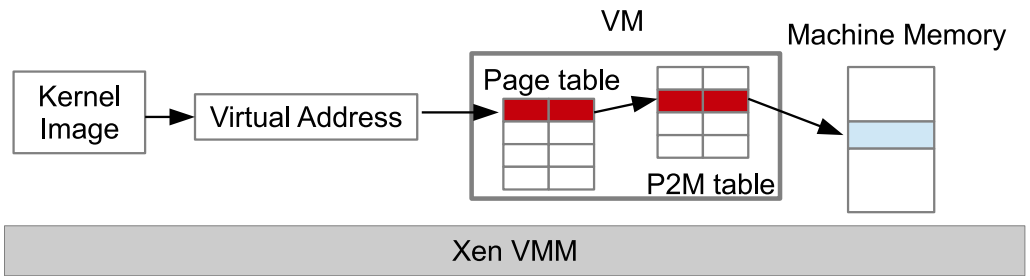


Figure 3.4. Accessing VMs' memory.

based on its scheduling policy and suspends processes if necessary. The periods allocated to these two modes can be configured by the administrators. The default periods are 500 ms for the controlled and autonomous modes, respectively.

3.2.2 Accessing Kernel Data

The Monarch scheduler obtains information on data types from the debug information of OS kernels. An example of such type information is the task_struct structure for representing the process in Linux. The debug information is generated by compiling the kernel with the debug option and stored in the DWARF [15] format. Such type information can also be obtained from the source code of the kernel, but obtaining type information from the source code is more complicated than doing from the debug information because the Linux kernel contains various macros for configuration and enables specific fields in data structures by defining macros.

To access data in a guest OS from the VMM, the Monarch scheduler have to translate their virtual addresses in domain U into *machine addresses* as illustrated in Figure 3.4. In Xen, the VMM uses the machine address to access the entire memory. Domain U is given *pseudo-physical memory* for the illusion of its own physical memory. First, the Monarch scheduler looks up the page table in domain U and translates a virtual address into a pseudo-physical address in the domain U. Next, it looks up the *P2M table* in the VMM and translates the pseudo-physical address to a machine address. The Monarch scheduler maintains the result of this translation as a cache. When it accesses virtual addresses in the same page, it can obtain machine addresses from the cache directly. The cache is invalidated before the Monarch scheduler continues the domain U because the page table and the P2M table can be changed while the domain U is running.

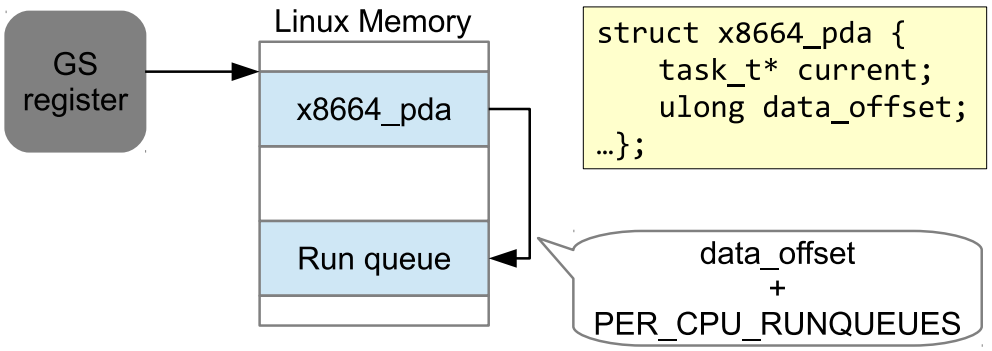


Figure 3.5. The Monarch scheduler finds the run queue dynamically.

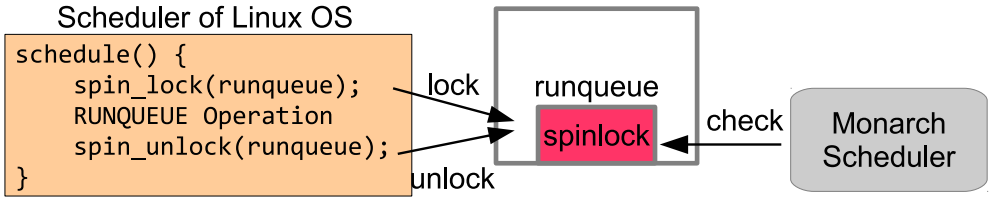


Figure 3.6. The Monarch scheduler checks the spin lock of the run queue.

When the Monarch scheduler examines processes in a guest OS, it traverses a circular list including all the processes. The staring point of the process list is the init_task symbol, which is invariant in each kernel image. The virtual address of this symbol is also obtained from the symbol table in the debug information of the kernel. On the other hand, it is not so straightforward to find all the run queues in a guest OS. In Linux, a run queue is created for each virtual CPU. Since the number of virtual CPUs is not determined until a VM is created, the address of each run queue changes according to the number. Therefore, the Monarch scheduler obtains the address of a run queue by starting from the GS base register of a virtual CPU in Figure 3.5. The register points to per-CPU specific data structure named x8664_pda. This data structure contains a pointer to a run queue.

The Monarch scheduler guarantees to consistently access kernel data in guest OSes. If it inspects a run queue while a guest OS is modifying it at the same time, it and/or the guest OS may crash. To prevent this situation, the Monarch scheduler checks locks for kernel data as depicted in Figure 3.6. Linux uses spin locks for mutual exclusion of accessing run queues and the process list, respectively. Before the Monarch scheduler access such kernel data, it checks whether the corresponding spin lock is acquired by a guest

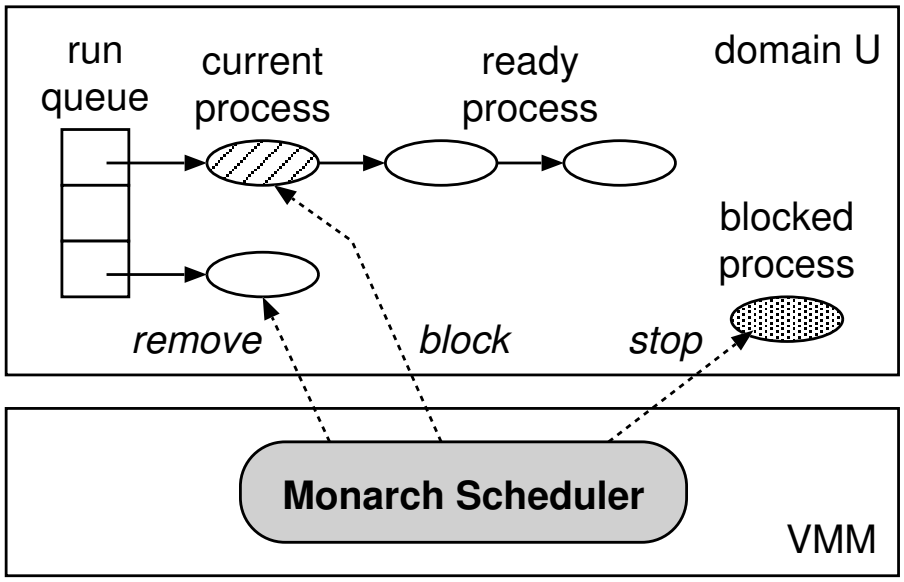


Figure 3.7. Suspending processes in various states.

OS or not. If the spin lock is not acquired, the Monarch scheduler can safely manipulate kernel data. It does not need acquire the spin lock because it pauses the domain U in which the guest OS runs. If the spin lock is already acquired by a guest OS, the Monarch scheduler skips scheduling at that time. Since a guest OS should release such spin locks in a short period, the Monarch scheduler can perform scheduling shortly.

3.2.3 Suspending/Resuming Processes

The Monarch scheduler uses several techniques for suspending processes according to their state. In Linux, a process has three main states: ready, running, and blocked. For a process in the ready state, the Monarch scheduler manipulates a run queue to suspend it. A process in this state is waiting for being scheduled in a run queue. In Linux, a run queue is an array of lists, each of which accommodates runnable processes with the same priority, as shown in Figure 3.7. To suspend a process in this state, the Monarch scheduler removes the process from one of the lists in a run queue. Together with this manipulation, it also updates the counter that maintains the number of processes in a run queue. Since the processes in run queues do not hold any locks in the kernels, suspending them does not cause deadlocks.

For a process in the running or blocked state, the Monarch scheduler

rewrites the process state to suspend it. A process in the running state is the currently running process and a process in the blocked state is waiting for I/O completion or lock acquisition. To suspend a process in the running state, the Monarch scheduler changes the process state to blocked. When a process scheduler in a guest OS is invoked for a context switch, the current process is removed from a run queue. If the state is not running, the process is not inserted into the run queue again. At this time, the process does not hold any locks in the kernel. Although the current process is also in a run queue in Linux, the Monarch scheduler cannot suspend it by directly removing it from the run queue. Even if the Monarch scheduler does so, the process scheduler in a guest OS inserts the current process into the end of the run queue and schedules it again.

On the other hand, the Monarch scheduler changes the process state to stopped for a process in the blocked state. The state of stopped is identical to that of a process suspended by the SIGSTOP signal. Even when a process is woken up by an event such as I/O completion, the wake-up function does not insert the process into a run queue if the process state is stopped. When such a process is stopped by a guest OS, it is guaranteed that it does not hold any locks in the kernel. Although the process can be resumed by sending the SIGCONT signal, the Monarch scheduler suspends the process again immediately. Since a process in this state is not in a run queue, the Monarch scheduler cannot remove the process from a run queue. Conversely, this process rewriting is ineffective for a process in the ready state. Even if the Monarch scheduler rewrites the process state, a process scheduler in a guest OS schedules the process without checking its state.

To resume a process that has been removed from a run queue, the Monarch scheduler inserts it into the run queue from which it has been removed. The inserted list in the run queue is selected according to its priority. For a process whose state has been rewritten, the Monarch scheduler rewrites its state to the ready state before inserting it into a run queue.

3.2.4 Monitoring Accurate Process Time

To record the execution time of each process, the Monarch scheduler measures the CPU time used for the execution in the context of the corresponding virtual address space. A virtual address space is uniquely identified by the machine address of the page directory in a page table. When a guest OS sets the address to the CR3 register in a virtual CPU for the context switch between processes, the Monarch scheduler can check the address. The in-

struction for changing CR3 is privileged and trapped by the VMM. Similarly, the Monarch scheduler can check the address when a context switch occurs between VMs and the current virtual address space is changed. The CPU time from when the specific value is set to CR3 until the value of CR3 is changed is accumulated as the corresponding process time.

The Monarch scheduler binds virtual address spaces to real processes by using process information in guest OSes. In Linux, the address of the page directory is stored in the mm_struct structure, which is followed from task_struct. By traversing the process lists in guest OSes, the Monarch scheduler can bind the accurate execution time to each process.

3.2.5 Support for the Windows Guest OS

3.2.5.1 Obtaining Type Information

The Monarch scheduler obtains information on data types from the debug information of the kernel of a guest OS. Examples of such type information are the `task_struct` structure in Linux and the `EPROCESS` structure in Windows for representing a process. In the current implementation, the developers manually obtain necessary type information in advance and then compile the Monarch scheduler using that specific type information. For example, a specific version of the `task_struct` structure has to be used.

For Linux, the Monarch scheduler obtains type information from its kernel image compiled with the debug option as illustrated in Figure 3.8linuxdbg. Such a kernel image contains debug information in the DWARF [15] format. In some Linux distributions, a debuginfo package for the kernel including debug information is provided. For Windows, on the other hand, type information has to be obtained via the WinDbg kernel debugger [53], as shown in Figure 3.9. WinDbg requires the Windows kernel to be booted in the debug mode, but it is not desirable to run the Windows kernel in the debug mode at production run. In addition, we could not run the Windows kernel in the debug mode on a VM in Xen 3.3.0. We guess that Xen does not support to run the Windows kernel in the debug mode because that mode is not used usually. Therefore, the developers need to run the Windows kernel in the debug mode without a VM and obtain necessary type information.

Such type information can be also obtained from the source code of guest OSes, but the Monarch scheduler uses debug information for two reasons. One is the availability of source code. The source code of some guest OSes is not open. The other reason is that the source code of kernels contains com-

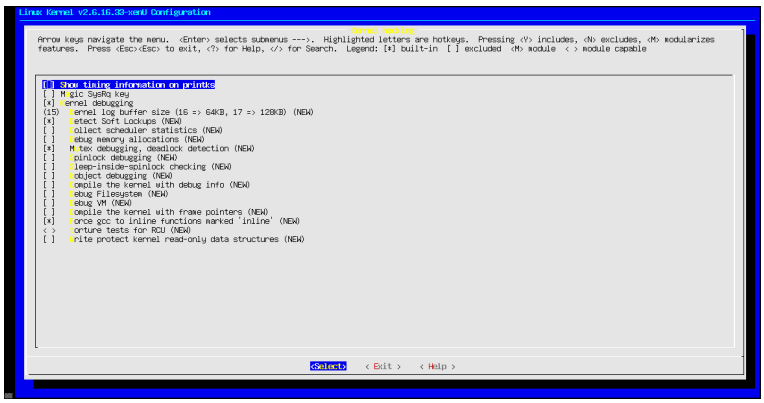


Figure 3.8. Compiling Linux with the debug option.

plex macros for configuration. The Linux kernel enables various functions by defining macros. According to defined macros, some fields in data structures are also enabled. For example, the definition of the `rq` structure for a run queue changes when the `CONFIG_SMP` macro is defined.

3.2.5.2 Finding Prochead of Windows

For Windows, however, finding such a starting point is not easy. As well as in Linux, all the processes are traversed from the `PsActiveProcessHead` global variable [82], as illustrated in Figure 3.10. Although the address of the variable can be obtained by using WinDbg, it changes whenever the OS is booted, at least, in case of Windows Vista. We guess that this is caused by the address space layout randomization (ASLR), introduced in Windows Vista [55]. ASLR changes the addresses where DLLs are loaded every time. Therefore, even if we can obtain the address for the Windows kernel running in the debug mode, the address is not valid after the kernel is rebooted in the non-debug mode.

To overcome this problem, the Monarch scheduler finds the `PsActiveProcessHead` global variable from a set of processes. First, the Monarch scheduler finds the candidates of all processes in a guest OS. Since it is difficult to directly find processes, the Monarch scheduler scans the whole memory and searches a bit sequence that represents a process. In Windows, a process is expressed as an object inside the kernel as shown in Figure 3.10. Each object contains a header that points to a type object corresponding to its type [82]. The address of each type object is unique. The Monarch scheduler searches that value from the whole memory, as described in the

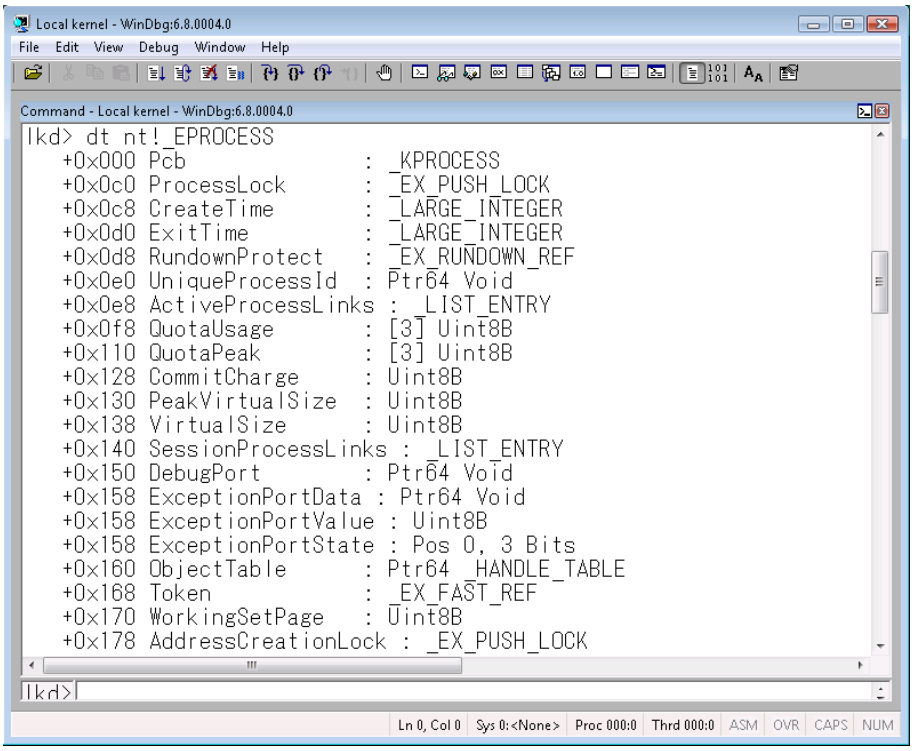


Figure 3.9. Obtaining the type information of the Windows kernel with WinDbg.

literature [8], such as Table 3.11. Since this memory scan takes time, the Monarch scheduler dumps the memory of a target domain into a file and analyzes the file. The target domain stops only during the memory dump and can continue to run after that.

From the found candidates of process objects, the Monarch scheduler selects likely process objects. The unique value for a process object does not always appear only in the header of process objects, but it can appear in the other area randomly. The Monarch scheduler examines all the candidates using the following knowledge: (1) a process name is an ASCII string in many cases, (2) a process ID is a multiple of four [68], and (3) all process objects are connected to one circular list.

Then the Monarch scheduler finds the `PsActiveProcessHead` global variable from the found process objects. `PsActiveProcessHead` is distinguishable from the other process objects by the difference of the memory location. In the x86-64 architecture, the high 32 bit of the address of `PsActiveProcessHead` is `0xfffff800` while that of the other process ob-

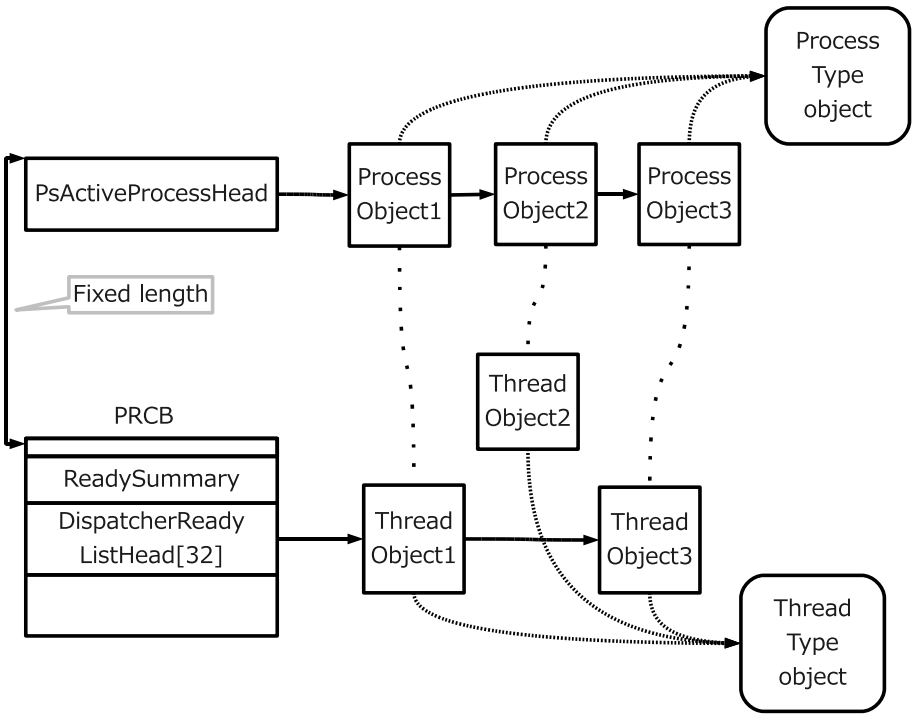


Figure 3.10. The data structures inside the Windows kernel.

jects is `0xfffffa80`.

3.2.6 Finding Run Queues of Windows

The Monarch scheduler finds all the run queues in guest OSes at runtime. In popular OSes supporting SMP, there exists a run queue for each virtual CPU. Since the number of virtual CPUs is not determined until a VM is created and a system is booted, the address of each run queue changes according to the number.

For Windows, the Monarch scheduler finds a run queue from the `PRCB` structure, which is per-CPU data structure [82]. The address of `PRCB` is unknown, but we found that it is located at a fixed distance from the address that `PsActiveProcessHead` points to.

Together with manipulating a run queue, the Monarch scheduler also updates related data structures. In Windows, a bitmap called `ReadySummary` has to be updated. The bitmap maintains whether there are threads in a specific priority or not. When the Monarch scheduler removes a thread from

```
    0     Idle           priority: 0 next: 0xffffffffffffff18,
    4     System         priority: 8 next: 0xfffffa8002c97c10,
 2212     WmiPrvSE.exe   priority: 8 next: 0xfffffa8001659c10,
 2424     explorer.exe   priority: 8 next: 0xfffffa8002f1ab50,
 2200     taskeng.exe    priority: 8 next: 0xfffffa8003696a40,
 2248     rdpclip.exe    priority: 8 next: 0xfffffa80036c5040,
 2316     dwm.exe        priority: 8 next: 0xfffffa80036d66c0,
    4                    priority: 0 next: 0xa800000009ff18,
...
 1112     svchost.exe    priority: 8 next: 0xfffffa800307dc10,
  680     logon.scr      priority: 4 next: 0xfffff800017ad338,
```

Figure 3.11. Finding processes from Windows Guest OS's memory.

a run queue, it clears the bit corresponding to that priority if there is no other thread in the same priority. When the Monarch scheduler inserts a thread into a run queue, it sets the bit corresponding to that priority. We found the information on which bit corresponds to which priority from the source code of the React OS [77].

Chapter 4

VMCrypt

VMCrypt prevents the sensitive information of the user VMs' memory from leaking via the management VM, using the trusted VMM. Nevertheless, the the management VM administrators can manage the user VMs with the management VM, including live migration, as before VMCrypt is introduced. VMCrypt supports para-virtualized guest OSes and allows the the management VM administrators to use the existing management software.

4.1 Dual Memory View

VMCrypt provides a dual memory view for each user VM: a normal view and an encrypted view, as illustrated in Figure 4.1. A *normal* view is a view of unencrypted memory and is used by a user VM. This view enables software running inside a user VM to access its memory as usual. In contrast, an *encrypted* view is a view of encrypted memory and provided to the management VM. Management software running in the management VM can access only encrypted data via this view, so that the attackers in the management VM cannot steal any useful information inside the user VMs. Nevertheless, the server administrators in the management VM can manage the user VMs through the encrypted views. For example, migrating a user VM is achieved by transferring the encrypted view of the VM's memory to the destination.

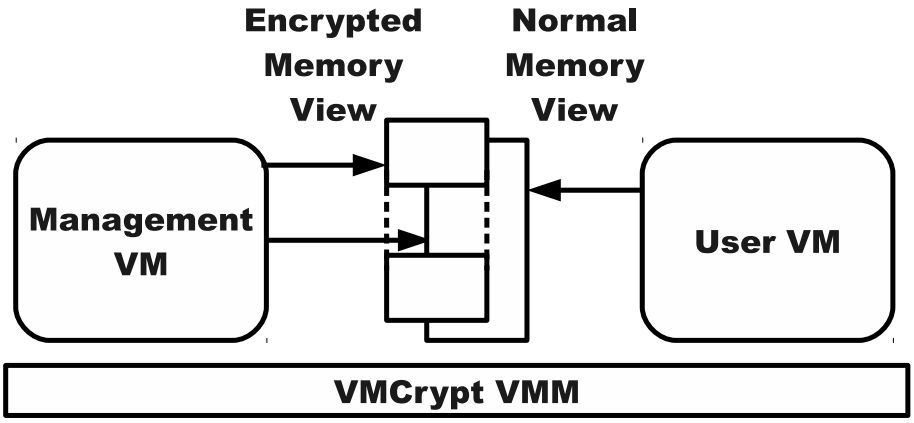


Figure 4.1. A dual memory view provided by VMCrypt.

To enable the server administrators to manage the user VMs with para-virtualization, VMCrypt exceptionally provides a normal view to the management VM only for several memory regions. For such regions, the management VM can directly access unencrypted data in a user VM. Examples of such regions are memory shared between VMs and the page tables inside a user VM. For para-virtualized guest OSes, the management VM needs to read and write data in the shared memory to exchange information with a user VM. The page tables have to be modified by the management VM during VM migration. The details of such unencrypted memory regions are listed in Section 4.3.4.

VMCrypt automatically identifies such unencrypted memory regions by monitoring the interaction between the management VM, the user VMs, and the VMM. For example, a shared memory region is passed from the management VM to the user VM via the VMM. The memory regions for the page tables are registered to the VMM. As such, the VMM can recognize all the unencrypted memory regions. The information on the identified memory regions is cached during the life cycle of a user VM, including VM migration, to preserve the compatibility with the existing management software. With the cache, the VMM can restore the user VM's memory correctly even at another host.

The two views in a dual memory view are provided *concurrently* to enable live migration. In other words, an encrypted view coexists with a normal one for each user VM. Therefore, the management VM can access the memory of a running user VM. Live migration requires that the management VM transfers the memory of a user VM while the user VM accesses its memory.

If VMCrypt directly encrypted the memory of the user VM by overwriting it, the running software in the user VM would fail when reading encrypted data.

VMM is trusted in VMCrypt, the attackers in the management VM cannot trick VMCrypt so that memory regions including sensitive information are provided to the management VM as a normal view. Even if the attackers attempt to register the whole memory as unencrypted regions, the VMM checks the validity of such special memory regions. We describes how such memory validation is performed in Section 4.3.4.

Let us briefly demonstrate how VM migration works well with VMCrypt for para-virtualized guest OSes. The management VM reads the encrypted memory contents through the encrypted view of a user VM and transfers them to a destination host. During this, the management VM modifies the page tables in the user VM so that the page tables can be migratable. The page tables are accessed through the normal view and transferred without encryption. At the destination host, the management VM writes received data to the encrypted view of a newly-created user VM as is. At the same time, the corresponding normal view is constructed from the encrypted view. The received page tables are written directly into the normal view and modified to be adapted to the destination host. The detailed behavior of VM migration with VMCrypt are described in Section 4.3.5.

4.2 Threat Model and Assumptions

We assume that the management VM can be compromised by outside attackers or abused by administrators. Such attackers could take the root privilege of the OS in the management VM and even alter the OS kernel. This means that the management VM is removed from the trusted computing base in terms of confidentiality.

In this paper, we focus on the attempts to steal sensitive information from the user VMs' memory. Information leakage from other resources such as CPU registers, storage, networks, and covert channels is out of scope. In real server consolidation, these resources should be protected against the management VM as well. Since there are several studies for protecting them, we can incorporate those with VMCrypt. For example, the protection of CPU registers has been proposed in SRE [48, 49]. Storage and networks can be encrypted by the guest OSes themselves, as described in Section 2, or by the VMM [84]. Mitigating the risk of covert channels is discussed

in the literature [78]. Also, we do not consider the other types of attacks against the user VMs from the management VM, such as integrity attacks and denial-of-service attacks.

We assume that server providers themselves are trusted, as widely accepted [83, 105]. In server consolidation, trusted senior administrators should be responsible for the maintenance of VMMs and the hardware. They would not be lazy or malicious, unlike average the management VM administrators that manage the user VMs in the management VM. Average administrators may manage VMMs and the hardware as well, but senior administrators should finally examine the correctness. Therefore, we assume that VMMs are well maintained and have no vulnerabilities that are compromised by the attackers. Also, we do not consider physical attacks such as the cold boot attack [23] because server rooms should be strictly protected in data centers.

We assume that the systems inside the user VMs are maintained sufficiently by the users and that they are not compromised directly from the attackers. If there were several software vulnerabilities or if weak passwords were used, the attackers could intrude into the users' systems. Once they were inside the user VMs, they could easily steal sensitive information. This paper excludes this possibility to focus on information leakage via the management VM.

We assume that the VMM is trusted while the management VM is not trustworthy. The VMM should not be compromised by the management VM, and the memory of the VMM is protected from the management VM. This assumption depends on the architecture of the VMM while there are several sort of implementations of the VMMs. VMMs that are suitable for VMCrypt are Xen [3], VMware ESX [101] On the other hand, VMMs that are not suitable for VMCrypt are Virtual PC [52], KVM [46], and VMware workstation [101].

4.3 Implementation

We have implemented VMCrypt in Xen 4.0.1 [70]. In Xen, a user VM is domain U and the management VM is domain 0. In the current implementation, VMCrypt supports para-virtualized Linux for the x86-64 architecture as guest OSes. VMCrypt mainly depends on Xen in how the VMM identifies unencrypted memory regions. If we implement this method in the VMM, VMCrypt can be applied to full-virtualization in Xen and the other virtualized systems such as VMware ESX.

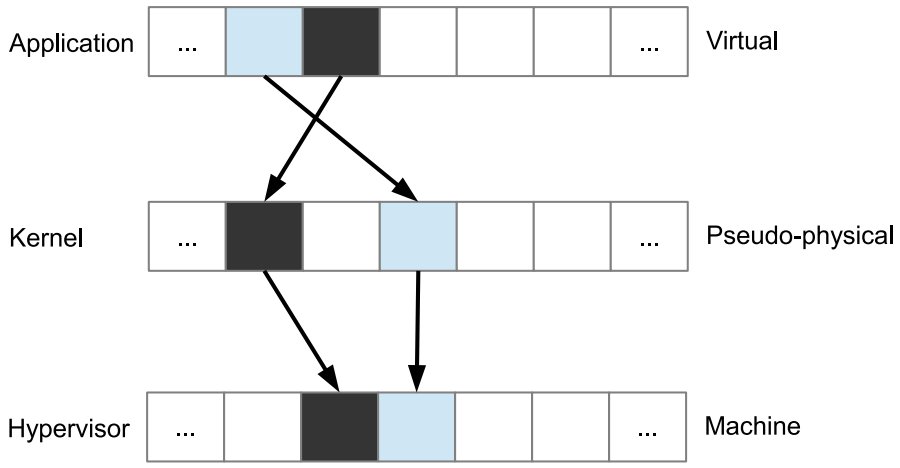


Figure 4.2. The three layers of Xen memory.

4.3.1 Memory Model in Xen

As illustrated in Figure 4.2, Xen distinguishes machine memory and pseudo-physical memory to virtualized memory. *Machine memory* is the entire memory installed in the machine and consists of a set of machine page frames. It is reserved for the VMM, allocated to domains, or unallocated. Each machine page frame has a number called the *machine frame number (MFN)*, which is consecutively numbered from 0. *Pseudo-physical memory* is a per-domain abstraction and allows a guest OS to consider the allocated physical pages as contiguous. For each machine page frame, a *pseudo-physical frame number (PFN)* is consecutively numbered from 0. The VMM maintains a machine-to-physical (M2P) table for the mapping from MFNs to PFNs. For the inverse mapping, a physical-to-machine (P2M) table is maintained by each domain.

4.3.2 Constructing an Encrypted View

To construct an encrypted view from a normal view of domain U's memory, the VMM encrypts the contents of memory pages of domain U when domain 0 maps those pages. Domain 0 has to map memory pages on its address space to access domain U's memory. To make two memory views coexist, the VMM replicates pages of domain U and maps them on domain 0, as illustrated in Figure 4.3 (a). For this *encrypted replication*, the VMM allocates pages in domain 0 and copies encrypted contents to them. When domain 0 unmaps

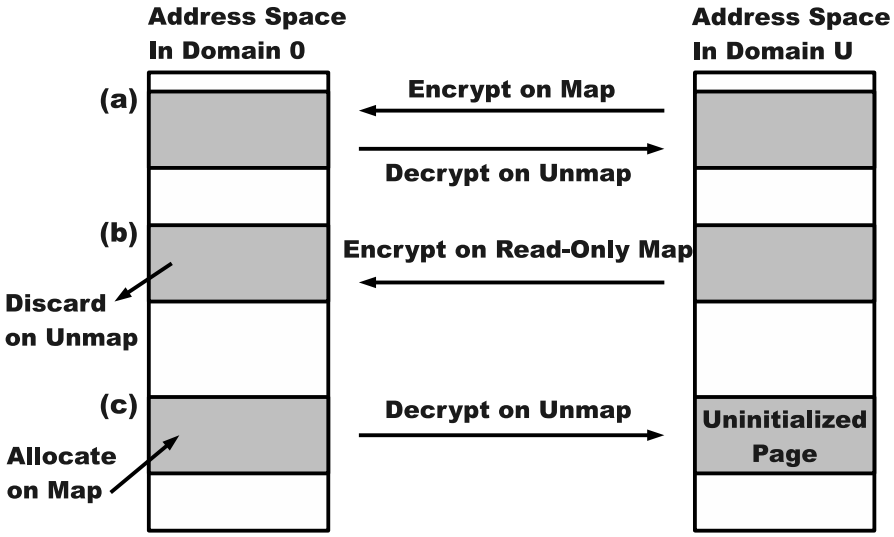


Figure 4.3. Synchronization between an encrypted view and a normal view.

domain U's pages, the VMM decrypts the contents, writes them back to the original pages in domain U, and releases the pages allocated in domain 0.

The VMM detects memory mapping and unmapping of domain U by monitoring the modification to the page tables of domain 0. The page tables are maintained by the VMM and protected to prevent illegal memory accesses. To modify a page table entry (PTE), domain 0 has to issue a hypercall to the VMM. If domain 0 attempts to modify its PTE directly, a page fault occurs and the VMM emulates the modification. In either case, the VMM can check the modification to PTEs. When a new PTE includes an MFN allocated to domain U, the VMM can notice that the modification is for memory mapping. When an old PTE includes an MFN belonging to domain U, the modification is for memory unmapping.

As an optimization for memory decryption, the VMM does not decrypt replicated pages when domain U's pages are mapped on domain 0 in a read-only manner (Figure 4.3 (b)). Since domain 0 cannot modify read-only pages, the contents do not need to be written back to the original pages in domain U. When domain 0 unmaps such pages, the VMM simply releases them. This optimization is enabled by a dual memory view that concurrently exists because unencrypted contents are still preserved in the original pages. This reduces the overhead of memory decryption. For example, all of the domain U's pages are just read on suspending domain U.

As an optimization for memory encryption, the VMM does not encrypt

domain U's uninitialized pages when those pages are mapped on domain 0 (Figure 4.3 (c)). We define as *uninitialized* a page in which any data has not been written since domain U is created. Such uninitialized pages do not need to be encrypted because they contain no valid data. This reduces the overhead of memory encryption. For example, most of the domain U's pages are initialized by domain 0 when domain U is booted and resumed.

In this implementation, normal and encrypted views are synchronized only on memory mapping and unmapping by domain 0 for efficiency. We call this *lazy synchronization*. If domain U is stopped, the encrypted view is the latest because domain U does not modify the normal view. However, the encrypted view may become obsolete if domain U is running. For example, live migration is performed without stopping domain U. This inconsistency between memory views is usually acceptable because domain 0 cannot access domain U's memory consistently even when the memory pages are shared between domain 0 and domain U, as traditionally performed. Management software in domain 0 should already consider this.

4.3.3 Dealing with Unencrypted Pages

The VMM does not encrypt the contents of unencrypted pages, which domain 0 can access for the VM management. When domain 0 maps such pages, the VMM makes domain 0 share the pages with domain U. Domain 0 can read information from the shared pages at any time, but it can update the pages only before domain U starts running. This prevents domain 0 from interfering with running domain U, e.g., by altering the page tables. This limitation does not disable most of VM management because domain 0 only sets up the shared pages at the creation time of domain U. When domain 0 unmaps the pages, the VMM simply ceases to share them. This mechanism is the same as the traditional one for memory mapping between domain 0 and domain U.

To cache unencrypted pages that have been identified once, the VMM maintains the *encryption bitmap*, as shown in Figure 4.4. Each bit corresponds to each machine page frame consecutively and all the bits are set at first. The bit is cleared if the corresponding page is identified as unencrypted. For the attempt to map domain U's pages, the VMM determines which view it provides to domain 0 by referring to this bitmap. The encryption bitmap is necessary because the VMM cannot always determine whether a specified page should be encrypted or not when the page is mapped on domain 0. The VMM identifies unencrypted pages at the appropriate times as described in

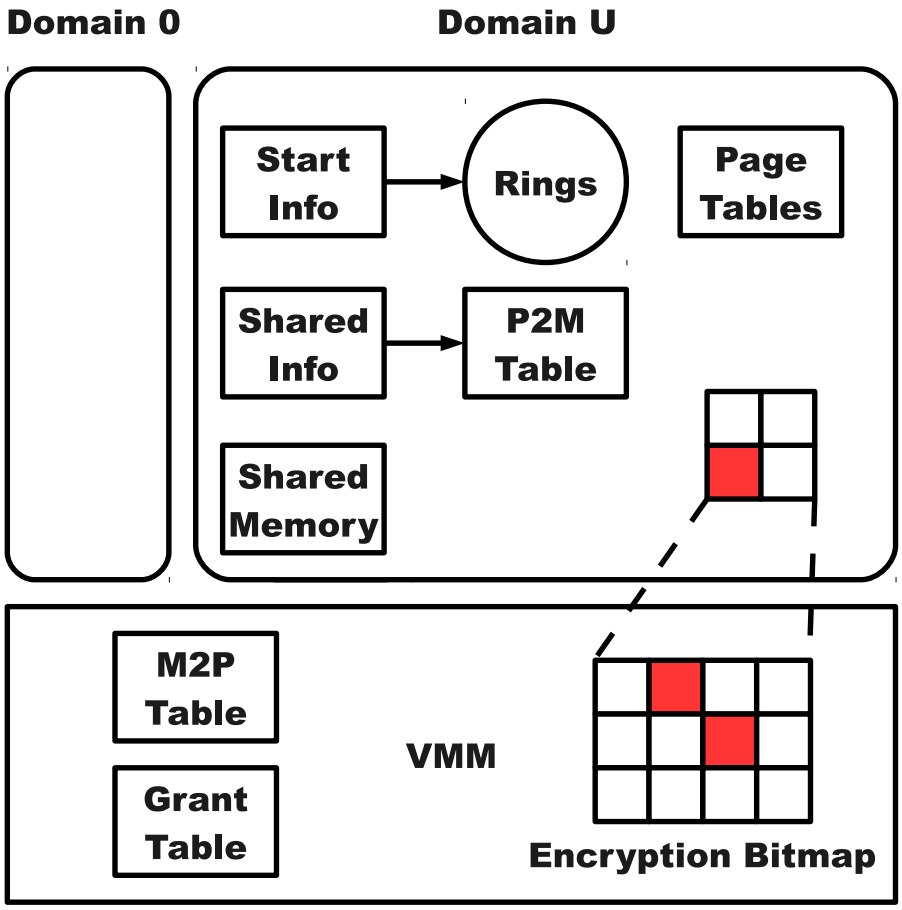


Figure 4.4. Unencrypted pages and the encryption bitmap.

Section 4.3.4 and constructs the encryption bitmap.

The encryption bitmap in the VMM is also embedded into domain U's memory. When domain U is migrated, the *embedded bitmap* is automatically transferred to the destination as well as the domain U's memory itself. This allows domain 0 to use the existing management software as is. Through the embedded bitmap, the VMM at the destination can extract information on unencrypted pages for the migrated domain U. Without the encryption bitmap, the VMM could not identify unencrypted pages when domain U is restored.

To allocate the bitmap inside domain U's memory, the VMM reserves a memory region using the e820 facility of BIOS. Since e820 is used to report the memory map to guest OSes, the VMM inserts a reserved area into that

memory map when it creates domain U. The encryption bitmap is copied to the reserved area in domain U's memory when those pages are mapped to domain 0. To preserve the integrity, the embedded bitmap is copied back on unmapping only while domain U is constructing, e.g., for VM migration. Its details are described in Section 4.3.5.

To prevent the attackers from tampering with the embedded bitmap, the VMM checks the integrity of the bitmap. Since the bitmap is also encrypted, the attackers cannot modify the bitmap as they intend. However, they can corrupt the bitmap by overwriting it. Even if the bitmap is changed randomly, several pages may be considered unencrypted illegally. As a result, the attackers may steal sensitive information from such pages. To detect such attacks, the VMM embeds versions and hashes into all the pages used for the embedded bitmap. Whenever one of these pages is mapped, the version is increased and the hash of the entire page is re-calculated. When the VMM decrypts the bitmap on memory unmapping, it checks that the version is the latest and the hash is correct. Also, the attackers cannot perform replay attacks, which attempt to use an old bitmap, because the version is obsolete.

4.3.4 Identifying Unencrypted Pages

The VMM automatically identifies the unencrypted pages to exceptionally deal with them. Figure 4.4 shows the data that domain 0 needs to access. This identification is specific to the para-virtualization in Xen.

4.3.4.1 Start Info

The start info page is used for passing information from domain 0 to domain U when domain U starts running. This page contains information necessary for booting a guest OS such as the allocated memory size. Domain 0 needs to set up this page when domain U is booted or restored.

The VMM identifies the start info page by monitoring virtual CPU registers of domain U. This identification is done when domain 0 issues the first unpause hypercall for the domain U. This hypercall is used to start running domain U. To pass the start info page to domain U, domain 0 sets the virtual address of the start info page in domain U to the RSI register at boot time. The VMM translates the virtual address to the corresponding MFN. Since the virtual address is straightly mapped to the pseudo-physical address at boot time, the VMM can obtain its PFN at first. Usually, the PFN can be translated to the MFN by the P2M table in domain U, but the P2M table has

not been constructed yet before domain U is booted. Therefore, the VMM examines all the entries in its M2P table one by one and finds the MFN. At resume time, domain 0 directly sets the MFN for the start info page to the EDX register. The VMM can obtain the start info page from it. Even if the attackers alter that register, they cannot steal useful information. They could make the VMM recognize an arbitrary page as the start info page. At this time, however, domain U will not be able to boot because an arbitrary page does not contain the correct information for booting. In addition, domain U has no sensitive information yet at the boot time.

4.3.4.2 Console/XenStore Rings

One pair of ring buffers is used to achieve the console of domain U. Domain 0 sets up four ring buffers to communicate with domain U at boot and resume times of domain U. Two ring buffers are used for achieving the console of domain U. One is for text outputs and the other is for key inputs. Domain 0 needs to read text written to the console of domain U and display it to the virtual console in domain 0. It also sends key inputs to the virtual console to the console of domain U. The other two ring buffers are used for domain U to access a storage system called XenStore in domain 0. Domain 0 and domain U exchange information on VM configurations through XenStore.

Since the information on these ring buffers is stored in the start info page, the VMM can identify it easily. Even if the attackers specify arbitrary pages as the ring buffers, domain U uses them only for console and XenStore, not for the other purposes. Domain U may write sensitive information to the console, and so the contents should be encrypted between domain U and its user. Such a mechanism for secure console is beyond the scope of this paper.

4.3.4.3 Shared Info

The shared info page is used for sharing information between the VMM and domain U. Through this page, the VMM notifies domain U of virtual CPU interrupts, wall-clock time, and so on. Domain U sets information on the P2M table to this page. Domain 0 needs to obtain the P2M table when it suspends and migrates domain U. As such, this page contains only information for enabling VMs, which is not sensitive.

The VMM can identify the page easily because the VMM allocates the shared info page when it creates domain U. The attackers in domain 0 cannot make the VMM and domain U use an arbitrary page as the shared info page.

4.3.4.4 P2M Table

The P2M table is a mapping table from PFNs to MFNs. Domain 0 needs to access this table to obtain all the MFNs allocated to domain U when it suspends and migrates domain U. The P2M table reveals which machine page frames are allocated to domain U, but it does not contain any sensitive information.

Since the MFN of the top page of the P2M table is stored in the shared info page, the VMM can easily identify this table. The P2M table has three-level hierarchical, tree structure to allow machine page frames to be sparsely allocated to domain U. The top node of the tree consists of one page, which contains an array of MFNs for mid-level nodes. A mid-level node contains an array of MFNs for leaf nodes. A leaf node contains the actual mapping from PFNs to MFNs. The VMM traverses the P2M table from the top node and obtains all the pages used for the table. The VMM performs this traversal whenever domain 0 maps the shared info page of domain U. The P2M table is partially constructed by domain 0 when domain U is created. Then it is reconstructed by the guest OS of domain U during its boot and resume processes. If the mapping between PFNs and MFNs is changed at runtime, the guest OS reconstructs the table again. The VMM needs to know the up-to-date P2M table so that domain 0 can access it.

The attackers in domain 0 cannot make arbitrary pages a part of the P2M table by modifying the P2M table. Since each entry in the P2M table is an MFN, the VMM validates pages used for the P2M table while it traverses them. It checks that the MFNs included in the pages are allocated to domain U. Since the possible range of an MFN is not broad, most of arbitrary pages cannot be the P2M table. Even if the attackers could make several pages be included in the P2M table, the VMM would consider them as invalid after domain U writes sensitive information to them.

4.3.4.5 Page Tables

In para-virtualized OSes, a page table is a table for translating virtual addresses into MFNs. When domain 0 suspends domain U, it needs to rewrite all the PTEs and PDEs so that the table maintains the translation from virtual addresses into PFNs. The rewritten tables are independent of machine page frames. When domain 0 resumes domain U, it rewrites all the PTEs and PDEs using newly-allocated MFNs inversely. Although the page tables contain information on the memory structure in domain U, such information is not sensitive.

The VMM can easily identify the pages used for the page tables in domain U because such pages are registered to the VMM using the mmuext_op hypercall. The pages are typed as PGT_l[1-4]_page_table in the VMM. If a page is no longer used for a page table, it is unregistered from the VMM. The VMM does not allow domain U to use unregistered pages as page tables.

The attackers in domain 0 cannot register arbitrary pages as page tables in domain U. If they could do that, they could map such pages with a normal view and steal information. When they register a page as a page table by issuing the hypercall, the VMM validates PTEs or PDEs included in the page by checking that MFNs included in PTEs and PDEs are allocated to domain U. Most of arbitrary pages in domain U cannot pass this validation. Even if the attackers could succeed in validating several pages such as empty pages, the VMM could detect the writes of sensitive information by domain U as invalid modification to page tables.

4.3.4.6 Introspected Data

VM introspection is a trade-off between detectability and confidentiality. For example, the user VMs want domain 0 to detect hidden processes while they may not want domain 0 to know the list of running processes. To enable VM introspection, domain 0 has to access the unencrypted memory of domain U. However, exposing the whole memory causes the leakage of sensitive information obviously. Therefore, VMCrypt allows the users to grant permissions for accessing specified data in domain U to domain 0. If the users wish, they can register a permission list to the VMM. The list consists of type (structure) names and symbol names that domain U allows domain 0 to introspect. Domain 0 can access memory pages for all the kernel objects of specified types and for data pointed by specified kernel symbols. The permission list is embedded into domain U's memory during migration as well as the encryption bitmap.

For dynamically-allocated kernel objects, the VMM identifies memory pages to be unencrypted, introspecting a slab allocator [5] in domain U. A slab allocator allocates one page only for kernel objects of the same type. When domain 0 attempts to map a page of domain U, the VMM checks whether the page is allocated for one of registered types by the slab allocator. If so, the page is not encrypted.

For kernel symbols, the VMM identifies pages including the data pointed by registered symbols as unencrypted. When domain 0 attempts to map a page of domain U, the VMM looks up the address of each registered symbol

and its data size from the symbol table in a guest OS. If the data is included in the page, the VMM does not encrypt the page. To minimize the data exposure, the VMM can encrypt the region other than the permitted data. Note that an unencrypted region is aligned by the block size of an encryption algorithm, for example, 16 bytes for AES. Therefore, the data around unencrypted regions may be still exposed to domain 0, but the size of such data is small.

4.3.4.7 Shared Memory with the Grant Table

The grant table is a mechanism for sharing memory pages between domain U and domain 0. For example, I/O ring buffers are shared between the front end drivers in domain U and the back end drivers in domain 0. With the grant table, domain 0 has to read and write the memory pages permitted by domain U. The VMM can identify all the shared pages by checking the grant table. Domain 0 can map only the pages that domain U explicitly permits to access. Such pages may include sensitive information, but domain U should encrypt it with encrypted file systems and VPN.

4.3.5 Live Migration with VMCrypt

In live migration, domain 0 transfers the memory image of running domain U from a source host to a destination host. VMCrypt encrypts the memory contents while it allows domain 0 to access necessary information in domain U's memory.

4.3.5.1 Source Host

Domain 0 first maps the shared info page of domain U to obtain information on the VM and the P2M table. Then it transfers the P2M table to the destination host. At this time, it canonicalizes the entries so that the table does not depend on host-specific memory allocation. Specifically, it replaces MFNs in the entries with the corresponding PFNs. Next, domain 0 maps all the pages of domain U and transfers their contents to the destination in turn. When domain 0 transfers pages used for page tables, it canonicalizes the PTEs in the tables. In live migration, domain 0 repeatedly transfers dirty pages, which are modified by domain U during migration. Finally, it stops domain U and transfers the remaining dirty pages and other states.

VMCrypt allows domain 0 to inspect domain U's memory pages necessary for migration, such as the shared info page, the P2M table, and the page tables. Such shared pages are not encrypted while the other pages are encrypted by the VMM when domain 0 maps them. Thanks to a dual memory view provided by VMCrypt, domain 0 can concurrently access encrypted pages while domain U accesses the original pages. When domain 0 unmaps pages, the VMM does not decrypt them for performance because any pages are not modified by domain 0.

When domain 0 maps the pages used for the embedded bitmap, the VMM copies the encryption bitmap in the VMM to the pages and encrypts them. If the encryption bitmap changes during live migration, the embedded bitmap has to be re-transferred. The VMM makes the pages for the embedded bitmap dirty so that the migration software in domain 0 re-transfers them automatically. To detect the corruption of the bitmap during migration, the VMM embeds the hash value of the bitmap into the domain U's memory. The bitmap is critical because domain 0 can map random pages as unencrypted by corrupting the bitmap.

Also, the VMM embeds the hash value of the page tables into domain U's memory to detect the alteration during migration. For example, if restored domain U uses altered page tables, the domain U might store sensitive information to unencrypted pages. When domain U is finally stopped at the source host, the VMM canonicalizes all the entries and calculates its hash value.

4.3.5.2 Destination Host

At the destination host, domain 0 creates a new domain U and reconstructs its memory using the received memory image. When it receives a memory page, it allocates a new page for domain U, maps it, and writes the contents to it. When domain 0 receives pages used for page tables, it uncanonicalizes the PTEs of the tables, according to the memory allocation at the destination host. Specifically, it replaces PFNs in the PTEs with the corresponding MFNs. Domain 0 repeats this as long as memory pages are transferred from the source host. When domain 0 has received all data from the source, it sets up the start info and shared info pages. Then it uncanonicalizes the P2M table and finally starts domain U.

The VMM does not decrypt the received memory pages on memory unmapping, but just before domain U starts running. Until the encryption bitmap is restored, the VMM cannot determine whether each page should

be decrypted or not. This delay of decryption does not cause any problems. During live migration, domain U at the destination host neither runs nor accesses any pages. Domain 0 can access shared pages such as the page tables because those pages are transferred without encryption from the source host. When domain 0 issues the first unpause hypercall to start domain U, the VMM extracts the embedded bitmap from domain U's memory. Then it copies back the bitmap to the encryption bitmap in the VMM. After the bitmap extraction, the VMM decrypts memory pages of domain U on the basis of the encryption bitmap.

As an optimization, the VMM decrypts memory pages as early as possible to reduce the downtime of live migration. If all pages are decrypted at the final stage, the downtime becomes long because domain U at the source host is stopped at this stage. The VMM extracts the embedded bitmap just after domain 0 receives all the pages used for the bitmap. After that, the VMM can decrypt memory pages on the basis of the restored encrypted bitmap. However, the encryption bitmap may not be consistent because it can be updated during live migration. For example, pages to be encrypted may be mapped without encryption before the encryption bitmap is correctly updated. This may lead to information leakage from domain U.

To prevent this inconsistency, the VMM maintains the *decryption record*, which is used to record whether each page of domain U has been decrypted or not, as shown in Figure 4.5. Its bit is set when the corresponding page is decrypted. When domain 0 unmaps a page of domain U, the VMM sets the corresponding bit of the decryption record if the VMM decrypts it on the basis of the encryption bitmap. When domain 0 maps the page later, the VMM determines whether the page should be encrypted or not, using the decryption record instead of the encryption bitmap. With the decryption record, the VMM can give a consistent memory view to domain 0. It is guaranteed that decrypted pages are necessarily encrypted when domain 0 maps them. At the final stage, the VMM adjusts the encryption of all the pages on the basis of the consistent encryption bitmap. Note that unencrypted pages may be incorrectly decrypted but they can be restored by encryption in case of AES-XTS [31] at least.

The attacks against the embedded bitmap cannot succeed in information leakage. Corruption of the bitmap is detected by the hash value embedded into domain U's memory. Replay attacks are useless because domain 0 is always given the same memory view as that at the source host. Even if a page is decrypted using an old encryption bitmap, it is necessarily encrypted again on the basis of the same bitmap when domain 0 maps it.

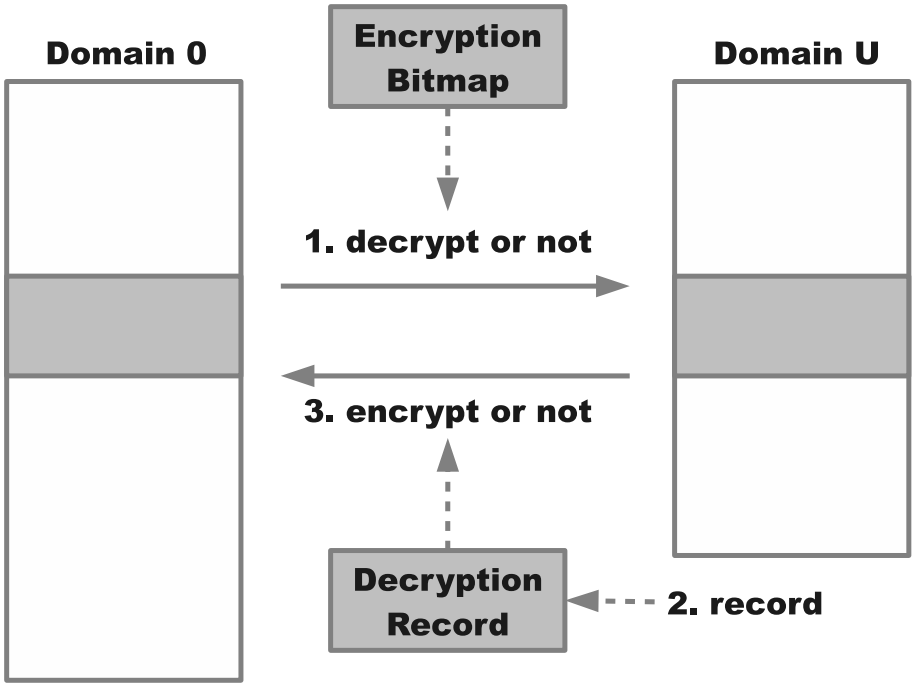


Figure 4.5. Encryption based on the decryption record.

Alteration of the page tables is also detected by the embedded hash value. Just before domain U starts running, the VMM canonicalizes the PTEs, calculates the hash value, and compares it with the embedded one. Replay attacks are not impossible, but it is difficult to steal useful information with only old page tables. In addition, the pages used for replayed page tables have to be marked as unencrypted in the encryption bitmap, which cannot be compromised.

4.3.6 Other VM Management with VMCrypt

4.3.6.1 Booting with VMCrypt

To boot domain U, domain 0 sets up the memory and devices of domain U. When it creates domain U, the VMM generates a session key used for encryption and decryption of the memory of the domain U. It also reserves a memory region in domain U using e820 for embedding the encryption bitmap. Then it registers the shared info page to the encryption bitmap. Note that the VMM does not refer to the encryption bitmap and provides a normal

view to domain 0 until the unpause hypercall is issued after domain U is created. This is because domain 0 itself has to construct the memory image of domain U from scratch. If the encrypted view of domain U were provided, domain 0 could not initialize the memory of domain U. Usually, the initial memory image of domain U does not contain sensitive information.

Domain 0 maps pages of domain U and writes its kernel image into them. It also constructs only leaf nodes in the P2M table of domain U. This P2M table is not yet registered to the encryption bitmap. It is done when the shared info page is mapped on domain 0 at the next time. Then it sets up the initial page tables of domain U. At this time, the VMM registers pages used for the page tables to the encryption bitmap. It sets up the start info page and the shared info page. Finally, domain 0 unmaps all the mapped pages but the VMM does not decrypt these pages. After domain 0 issues the unpause hypercall, the guest OS boots.

4.3.6.2 Suspension with VMCrypt

To suspend domain U, domain 0 saves its memory into a file. First, domain 0 maps the shared info page of domain U to obtain information on the VM and the P2M table. At this time, the VMM traverses the P2M table and registers the pages used for the table to the encryption bitmap. Domain 0 maps the P2M table, translates MFNs in its entries into the corresponding PFNs, and saves them into the file. Next, domain 0 maps all the pages of domain U and saves their contents into the file in turn. If a mapped page is to be encrypted, the VMM encrypts its contents. Otherwise, it maps the page as is. When domain 0 maps pages used for page tables, it translates MFNs in the page tables into the corresponding PFNs. When domain 0 maps the pages for embedding the encryption bitmap in the VMM, the VMM copies the bitmap to the pages and encrypts them. After domain 0 maps and saves a set of pages, it unmaps them but the VMM does not decrypt these pages because of read-only mapping.

To keep the session key of suspended domain U, domain 0 obtains the key from the VMM using the vmcrypt_key_op hypercall before it suspends domain U. The session key is encrypted with the public key in the VMM. Domain 0 saves the encrypted session key into a file.

4.3.6.3 Resumption with VMCrypt

To resume domain U, domain 0 puts a saved memory image back to the memory of domain U. This operation is similar to booting domain U from the VMM's point of view. First, when domain 0 creates domain U, the VMM generates a new session key. Unlike boot, domain 0 passes the saved session key for resuming domain U to the VMM using the vmcrypt_key_op hypercall. The VMM decrypts the encrypted key using its private key and uses it for decrypting the memory image. After domain U starts running, the VMM discards the old session key and uses the new one. During resume, domain 0 maps pages of domain U and writes the saved memory image into the pages in turn. Like boot, the VMM does not refer to the encryption bitmap at first, but it starts to refer to the bitmap after the VMM extracts the bitmap embedded into domain U's memory. It is earlier than the time of the first issue of the unpause hypercall. Therefore, the VMM does not encrypt or decrypt the pages of domain U until the bitmap extraction.

The VMM extracts the embedded bitmap from the restored memory of domain U as early as possible. Once all the pages where the bitmap is embedded are mapped and then unmapped, the VMM considers that the bitmap is restored. This validity can be checked by the hashes embedded in the pages. At this time, the VMM copies the embedded bitmap back to the encryption bitmap in the VMM. Since the embedded bitmap is sorted by the PFN but the encryption bitmap is by the MFN, the VMM translates the embedded bitmap with the M2P table in the VMM. Using the P2M table is natural for this translation, but the P2M table of domain U is not yet restored correctly. When the VMM extracts the embedded bitmap, it decrypts all the already restored pages on the basis of the encryption bitmap. To prevent pages still mapped on domain 0 from being decrypted, the VMM maintains a map count, which is how many times each page is mapped on domain 0 without being unmapped. When the map count becomes zero, the VMM can securely decrypt the contents of the page without exposing them to domain 0. After that, the VMM refers to the encryption bitmap whenever pages of domain U are mapped on domain 0. After domain 0 issues the unpause hypercall, domain U starts running and the guest OS performs resume operations.

4.3.7 Introspection with VMCrypt

When domain 0 introspects certain data in domain U, it first translates its virtual address to an MFN by traversing the page table in domain U. Since the page table is not encrypted, domain 0 can perform this translation as usual. Then, domain 0 maps the page of the MFN. At this time, the VMM does not encrypt it if the page is used for permitted types of kernel objects. If the page includes data pointed by permitted symbols, the VMM does not encrypt it. Otherwise, the page is encrypted. For example, domain 0 can traverse the process list in domain U if the task_struct structure and the init_task symbol are permitted by the users of the domain U.

4.3.8 Security Consideration

In VMCrypt, the management VM could interfere with the VMM through the alteration of unencrypted pages of the user VMs. For example, when the VMM traverses the P2M table, it might crash if the table was altered by the attackers so that it includes non-existing memory pages. To prevent this, our VMM carefully checks that the accessed pages belong to the target VM.

VMCrypt increases the size of the VMM, resulting in a larger trusted computing base. This may make the whole system more vulnerable, but the increased code size is 12500 lines, including 6500 lines of code for AES. This size is only 5 % of the original VMM.

Chapter 5

Experiments for the Monarch Scheduler

We performed experiments to examine the overheads and the scheduling abilities of the Monarch scheduler. For a server machine, we used a PC with one Intel Core 2 Duo processor E6600, 6 GB of memory, and a Gigabit Ethernet NIC. We ran Xen 3.4.2 for the x86-64 architecture on this PC. For domain 0, we allocated two virtual CPUs and 1 GB of memory and we ran Linux 2.6.18. For domain U, we allocated one virtual CPU and 1 GB of memory and we ran Linux 2.6.18 as a guest OS. For a client machine, we used a PC with one Intel Core 2 Quad processor Q9550S, 8 GB of memory, and a Gigabit Ethernet NIC. These two machines were connected with a Gigabit Ethernet switch.

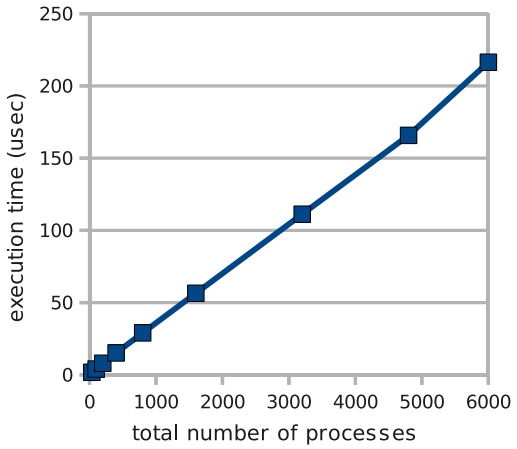
5.1 Scheduling Overheads

To examine the overheads of running the Monarch scheduler, we measured the time needed for traversing the process lists in guest OSes. Each VM is paused during this traversal. In this experiment, the Monarch scheduler searched target processes from the process lists by comparing process names and did not suspend or resume any processes. On each guest OS, 36 processes

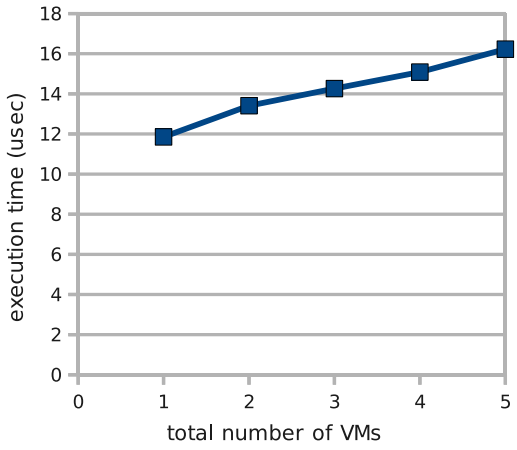
were running originally. We performed the traversal of the process lists 1000 times and obtained the average time.

First, we changed the number of processes in one VM between 36 and 6000 to examine the impact of the length of the process list. Spawning more than 6000 processes caused an out-of-memory error. To adjust the number of processes, we ran dummy processes that always slept. As in Figure 5.1(a), the traversal time is proportional to the number of processes and 36 ns for one process. For traversing 6000 processes, it takes 220 μs and the overhead is 2.2% when the scheduling interval is 10 ms. However, running 6000 processes in one VM is not realistic. For 400 processes, it takes 15 μs and the overhead is 0.15%.

Next, we changed the number of VMs between one and five. The purpose of this experiment is to clarify the overheads of inspecting multiple VMs. Therefore, we fixed the total number of processes in the whole system to 300. Figure 5.1(b) shows that the time for traversing 300 processes depends on the number of VMs but increases only 0.88 μs per VM. This overhead comes from increasing the number of pausing virtual CPUs and checking locks for kernel data. From this result, the scheduling overheads mainly come from the number of processes.



(a) For processes



(b) For VMs

Figure 5.1. The time for traversing process lists.

5.2 Monitoring Overheads

To examine the overheads of monitoring process execution in the VMM, we measured the time needed for recording the execution time of processes with CR3 and the number of context switches per second. We performed this experiment at the VM start-up time and in a steady state. We regarded 15 seconds after booting a VM as the VM start-up time. While many processes were created at the VM start-up time, only a few processes ran in a steady

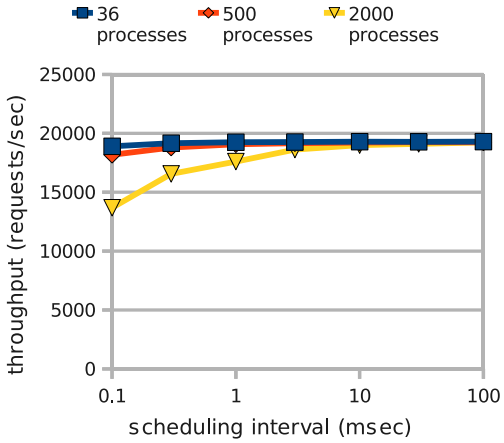
state. We used between one and five VMs.

At the VM start-up time, it took 0.26 μs per context switch and context switches occurred 1467 times per second on average. From these results, the overhead of process monitoring is 0.04%. In a steady state, on the other hand, it took 0.20 μs per context switch. Since context switches occurred 129 times per second, the overhead of process monitoring is 0.003%. The reason why it takes more time at the start-up time is that newly created processes need to allocate new recording area. In any cases, this overhead is negligible.

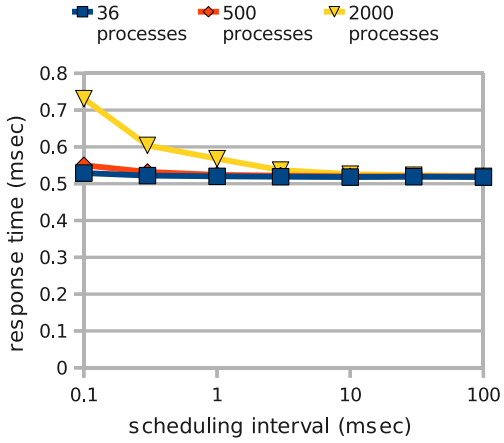
5.3 Performance Degradation

To examine the performance degradation due to the above scheduling and monitoring overheads, we measured the throughput and response time of the lighttpd web server [34]. In this experiment, we created one VM and ran the lighttpd process and dummy processes. The Monarch scheduler traversed the process list to find target processes and did not change the process scheduling. We used the ApacheBench benchmarking tool [95] and sent ten requests concurrently.

We changed the scheduling interval at which the Monarch scheduler was invoked between 0.1 and 100 ms. We measured the throughput and response time when the number of processes was 36, 500, and 2000. We chose the maximum number of processes so that the time for traversing the process list was less than 0.1 ms. Figure 5.2 shows the results. The performance was degraded largely when the interval was 0.1 ms and the number of processes was 2000. However, this interval is too short realistically. When the interval was 10 ms, which is the default in the Monarch scheduler, the throughput was degraded by 1.5% and the response time became 1.3% longer even for 2000 processes. For 500 processes, the performance was degraded by less than 0.3%.



(a) Throughput



(b) Response time

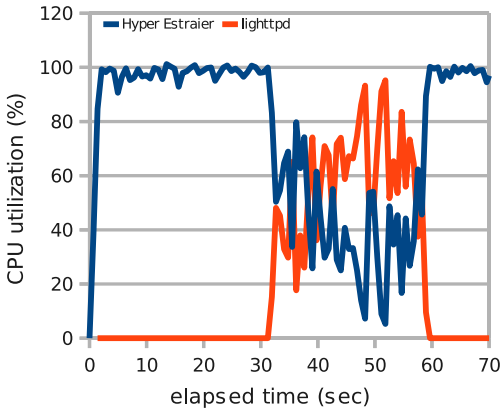
Figure 5.2. The performance degradation of a web server.

5.4 System-wide Idle-time Scheduling

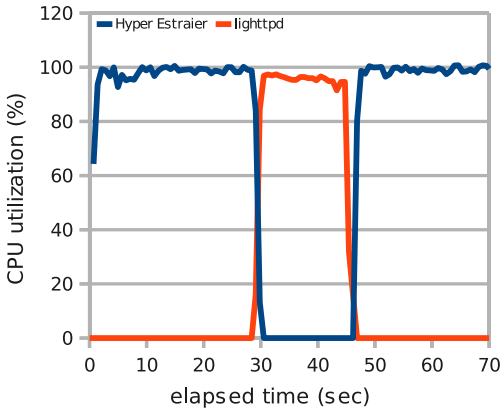
We examined the effectiveness of idle-time scheduling achieved by the Monarch scheduler. We ran lighttpd in VM 1 and the file indexing in Hyper Estraier [28] in VM 2. Hyper Estraier is a high-performance text search engine. First, we monitored the activities of these two processes when we did not use the Monarch scheduler. Figure 5.3(a) shows the changes of the CPU utilization of these two processes. While lighttpd was running in VM

1, the file indexing was also running because it was only an active process in VM 2. Therefore, the file indexing largely affected the execution of lighttpd although it should stop. The throughput of lighttpd was degraded by 24% and the response time was 32% longer.

Second, we used the Monarch scheduler to execute the file indexing only at idle time in the whole system. To examine the accuracy of scheduling, we disabled hybrid scheduling in this experiment. Figure 5.3(b) shows the results of this system-wide process scheduling. When lighttpd started running in VM 1, the file indexing stopped immediately in VM 2. The throughput of lighttpd was degraded by 2.4% and the response time was 2.5% longer.



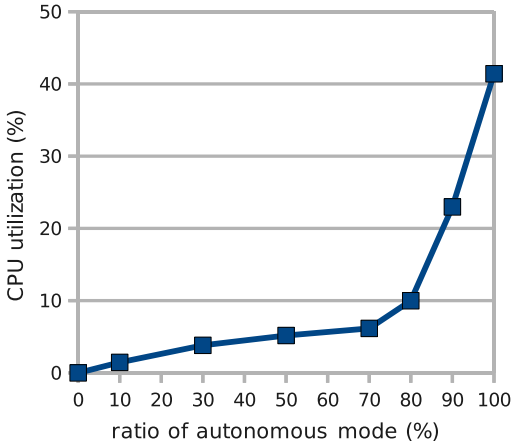
(a) Default scheduling



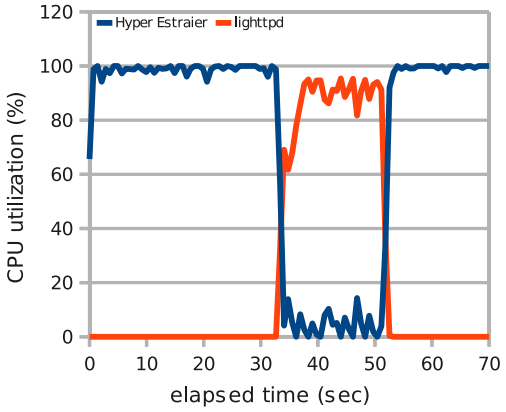
(b) Idle-time scheduling

Figure 5.3. System-wide idle-time scheduling for Hyper Estraier.

Third, we enabled hybrid scheduling of the Monarch scheduler. Without hybrid scheduling, the attackers can completely prevent the execution of the file indexing by making lighttpd always busy. We changed the ratio of the controlled and autonomous modes. Figure 5.4(a) shows the CPU utilization of the file indexing for the various ratios of the autonomous mode. As the ratio becomes large, the file indexing runs more. Figure 5.4(b) shows the changes of the CPU utilization of two processes when the ratio is 50%. Strictly speaking, hybrid scheduling violates idle-time scheduling but prevents DoS attacks from the VM that executes lighttpd. When the ratio of the autonomous mode is more than 80%, the CPU utilization increases steeply. This is because the Monarch scheduler switches from the controlled mode to the autonomous one before the manipulation of guest OSes works effectively.



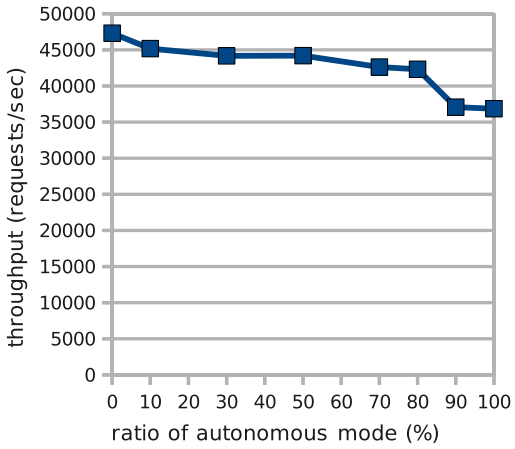
(a) CPU utilization



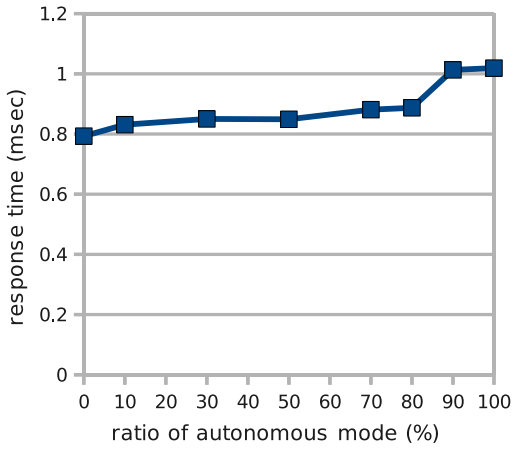
(b) Hybrid scheduling

Figure 5.4. The effects of hybrid scheduling with idle-time scheduling.

Hybrid scheduling degrades the performance of lighttpd even when lighttpd runs normally. The performance degradation is shown in Figure 5.5. As the ratio of the autonomous mode is increasing, the throughput is decreasing and the response time becomes longer. When the ratio is 50%, the throughput degradation is 9.7% and the response time is 8.8% longer.



(a) Throughput



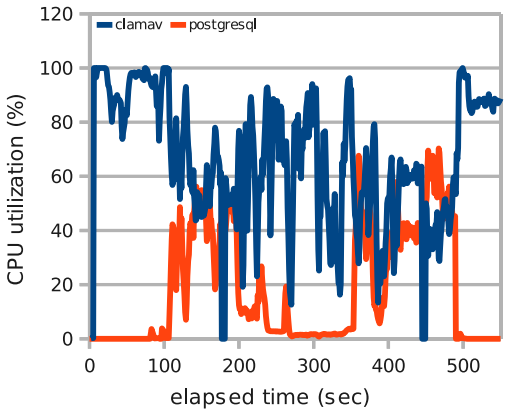
(b) Response time

Figure 5.5. The performance degradation by hybrid scheduling.

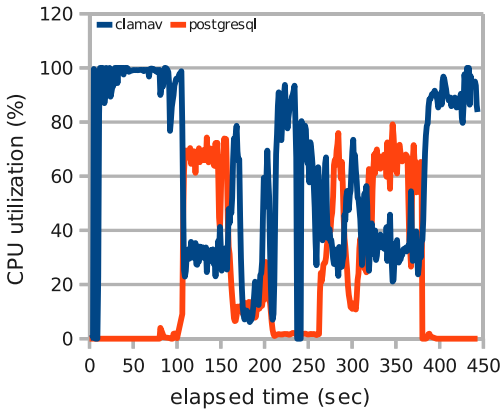
5.5 System-wide Priority Scheduling

We examined the effectiveness of priority scheduling by the Monarch scheduler. We ran the power test of the DBT-3 benchmark [106] in VM 1 and the virus scanner of ClamAV [88] in VM 2, a virus scanner is frequently run at a lower priority [12]. DBT-3 tested the performance of PostgreSQL in a decision support system. Without system-wide priority scheduling, the virus scanner interfered with PostgreSQL across VMs as shown in Figure 5.6(a).

When we ran only DBT-3, the power test took 221 seconds. On the other hand, it took 384 seconds when we ran the power test with the virus scanner.



(a) Default scheduling



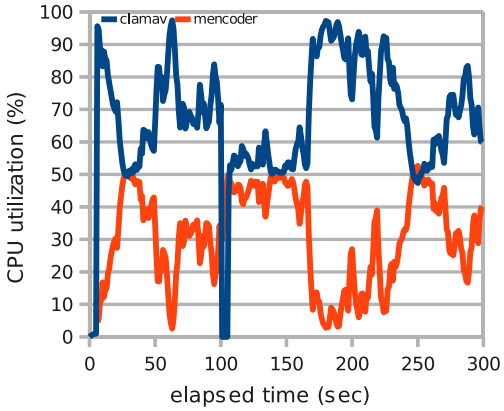
(b) Priority scheduling

Figure 5.6. System-wide priority scheduling for DBT-3 and ClamAV.

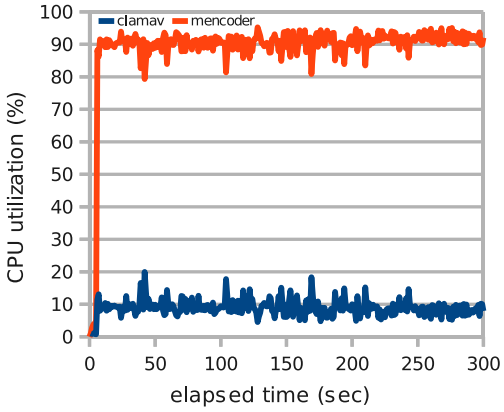
To execute the virus scanner in a lower priority than PostgreSQL, we configured the priorities of PostgreSQL and the virus scanner so that their CPU shares were 2 and 1, respectively. Figure 5.6(b) shows the results. When PostgreSQL needs much CPU time, for example, between 100 and 150 seconds and between 320 and 380 seconds, the CPU utilization of PostgreSQL is approximately double that of the virus scanner. When PostgreSQL is idle, the virus scanner uses most of the CPU time. Under this scheduling, the

power test took 275 seconds, which was 28% faster than when we did not use the Monarch scheduler.

Next, we performed other experiments to demonstrate DoS attacks to ClamAV and show the effectiveness of hybrid scheduling. We ran four processes for MEncoder [58] in VM 1 and the virus scanner of ClamAV [88] in VM 2. MEncoder encoded MPEG-4 [57] video data of 800MB to the SONY PSP [87] format. We assigned a high priority to MEncoder and a low priority to the virus scanner so that their shares were 2 and 1, respectively. Figure 5.7(a) shows the total CPU utilization of all MEncoder processes and that of the virus scanner when we did not use the Monarch scheduler. The virus scanner could use more than 50% of the CPU time. However, when we used the Monarch scheduler, the CPU utilization of the virus scanner was reduced to 12%, as in Figure 5.7(b). As such, the attackers can perform DoS attacks using system-wide scheduling.



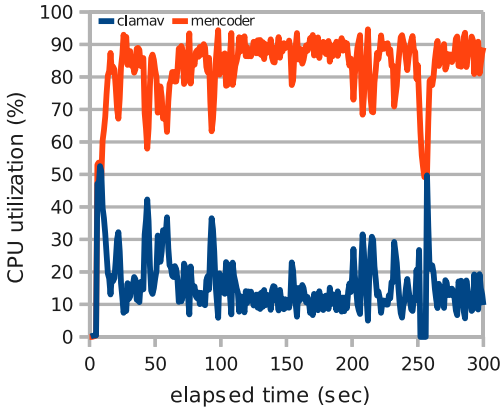
(a) Default scheduling



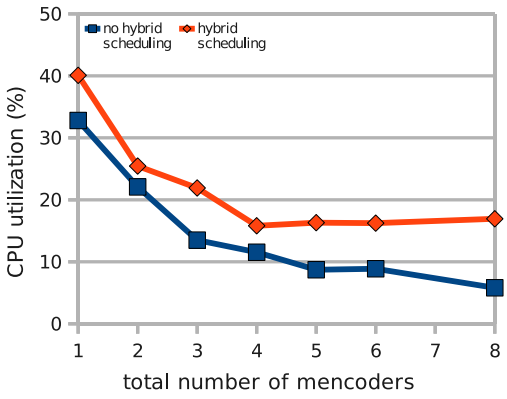
(b) Priority scheduling

Figure 5.7. System-wide priority scheduling for MEncoder and ClamAV.

Figure 5.8(a) shows the CPU utilization of these two when we enabled hybrid scheduling. The virus scanner can obtain more CPU time. Figure 5.8(b) shows the relationship between the number of MEncoder processes and the CPU utilization of the virus scanner. Without hybrid scheduling, as the attackers ran more MEncoder processes in VM 1, the CPU utilization of the virus scanner in VM 2 was decreasing. When we configured the ratio of the autonomous mode to 50% in hybrid scheduling, the virus scanner can obtain more than 16% of CPU time even for more than four MEncoder processes.



(a) Hybrid scheduling



(b) Impact of MEncoder

Figure 5.8. The effects of hybrid scheduling with priority scheduling.

5.6 Proportional-share Scheduling in One VM

We ran four π calculator processes [17], PI_1, PI_2, PI_3, and PI_4 in a VM. We assigned k shares to PI_k, respectively, so that the ratio of the allocated CPU times is $1 : 2 : 3 : 4$. For stable scheduling, we limited the sum of the CPU times allocated to these four processes to 60%. Figure 5.9 shows the changes of the CPU utilization of these four processes. The averages of the CPU utilization are 7.8%, 13%, 18%, and 24%, respectively, and its ratio is 1.3 :

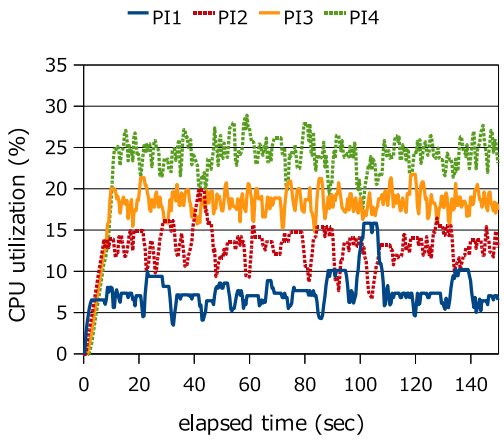


Figure 5.9. The CPU utilization in proportional-share scheduling for processes in one VM.

2.2 : 3 : 4. The Monarch scheduler achieves the target ratio approximately. These standard deviations are 2.2%, 2.1%, 1.3%, and 1.9%, respectively.

5.7 System-wide Proportional-share Scheduling

We ran two MEncoder processes, E_1 and E_3, in VM 1 and one MEncoder process, E_2, in VM 2. We assigned 1 share to E_1, 2 shares to E_2 and 4 shares to E_3, so that the ratio of the allocated CPU times is 1 : 2 : 4. From Figure 5.10, the averages of the CPU utilization are 19%, 39%, and 74%, respectively, and its ratio is 1 : 2.1 : 3.9. Even when target processes are in two VMs, the Monarch scheduler achieves the target ratio approximately. These standard deviations are 5.5%, 11%, and 5.7%, respectively.

5.8 System-wide Multi-OS Process Scheduling

We examined the idle time scheduling targeted at Windows and Linux to show that the Monarch Scheduler can schedule processes with one scheduling poly in the multi-OS environment. We ran lighttpd in VM1 with Linux, and ran SearchIndexer in VM2 with Windows. SearchIndexer is a process to create regularly indexes to search files on Windows. The scheduling policy of the idle-time scheduling is the same as the section 5.4, replacing the name

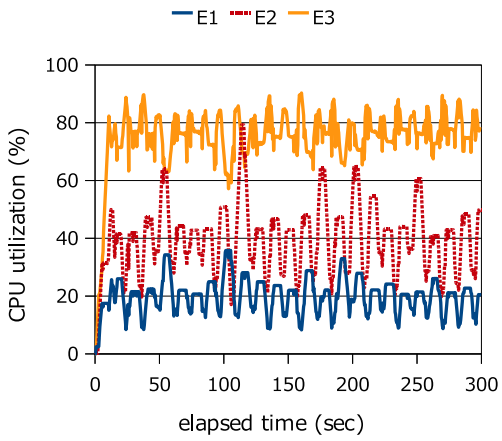
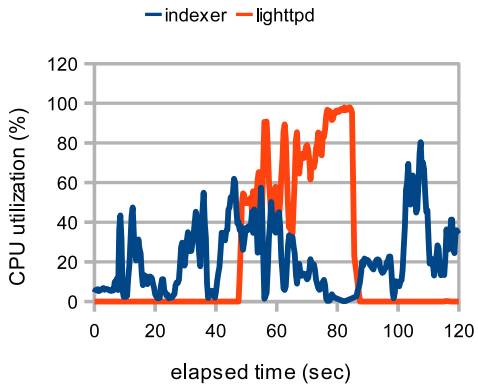
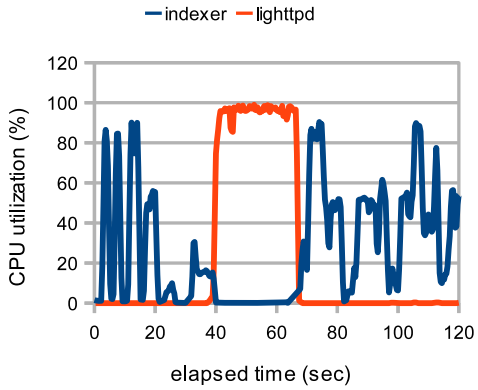


Figure 5.10. The CPU utilization in proportional-share scheduling for processes among two VMs.

of HyperEstraier with SearchIndexer.



(a) Default Scheduling



(b) Idle-time Scheduling

Figure 5.11. System-wide idle-time scheduling across multiple OSes.

Figure 5.11(a) shows the CPU utilizations of the processes without Monarch Scheduler. Lighttpd and SearchIndexer ran simultaneously, the throughput of lighttpd degraded by 25% and the response time increased by 34%. Next, we ran Monarch Scheduler to run SearchIndexer whenever the whole system is idle. Figure 5.11(b) shows that the SearchIndexer in the VM2 stopped whenever the lighttpd ran in the VM1, and Monarch Scheduler could control SearchIndexer according to the policy of the idle time scheduling. Therefore it is showed that Monarch Scheduler can control processes in another OSes with the identical policy. In this situation, the throughput of lighttpd degraded by 4.3%, the response time increased by 4.5%.

5.9 Dependence on Guest OSes

The Monarch scheduler depends on the internal structures of guest OSes because it directly monitors and manipulates the kernel data. Even for the same OS, its internal structures can change among different versions. Several data structures are altered by refactoring, adding new features, changing the scheduling algorithm. To examine how much the Monarch scheduler has to be modified when the Linux kernel is updated, we inspected 33 versions of the Linux kernel 2.6.

The kernel was largely modified between these versions, but it was shown that the Monarch scheduler was not affected by most of kernel updates because it depends only on the scheduling and management of processes. When new data structures are added to an OS, the Monarch scheduler can ignore them if it does not refer to them. Even if data structures that the Monarch scheduler refers to are changed, the Monarch scheduler just obtains type information on them from the debug information of the kernel again.

To examine how much the Monarch scheduler has to be modified when the Linux kernel is updated, we inspected 33 versions of the source code of the Linux kernel 2.6. Figure 5.12 shows the changes of the lines of code in the process scheduler from 2.6.0 to 2.6.32. The scheduler was changed in most of kernel updates and the lines of code increased more than 5 times. The Table 5.1 shows, however, that there were several small changes that we had to modify manually. In 2.6.14, a field of the spinlock_t structure was changed so that it was contained in another structure. In 2.6.18, the runqueue structure was simply renamed to rq. In 2.6.30, the calculation of the address of a run queue was changed so that a fixed offset was added to the value of the GS base register. For these changes, we could easily modify the Monarch scheduler.

On the other hand, the scheduling algorithm was changed from the O(1) scheduler to the completely fair scheduler (CFS) in 2.6.23 [50]. The O(1) scheduler uses doubly-linked lists of processes as run queues while CFS uses a red-black tree. Since the new scheduler changed both data structures and a scheduling algorithm, we needed to modify the Monarch scheduler largely. According to our deep inspection, we could modify the Monarch scheduler so that it can change the behavior of CFS.

Version	Changes	Difficulty
2.6.14	The internal structure of spinlock_t has changed.	Easy
2.6.18	The name of runqueue has renamed to rq.	Easy
2.6.23	The process scheduler has changed from O(1) to the Completely Fair Scheduler(CFS)	Difficult but Possible
2.6.30	The way to calculate the address of runqueue has changed.	Easy

Table 5.1. Modifications of the Monarch scheduler when the Linux kernel is updated.

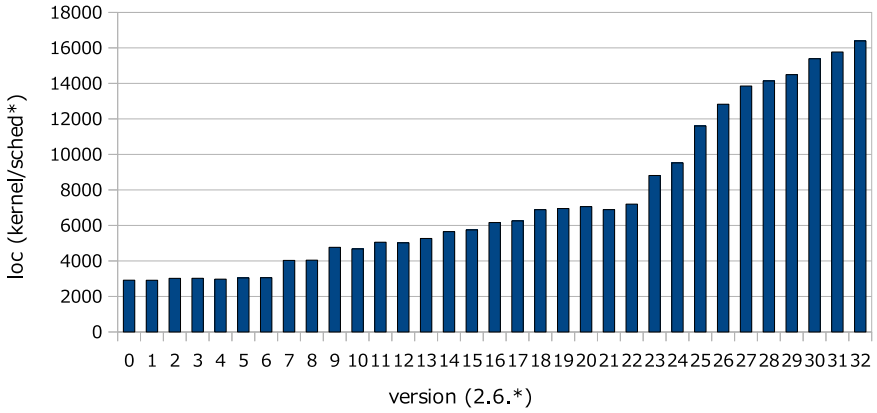


Figure 5.12. The changes of the lines of code in the Linux process scheduler.

5.9.1 The Cost of Supporting CFS by the Monarch scheduler

The new CFS scheduler's runqueue is implemented by the Red Black Tree [81, 47] library in `lib/rbtree.c`. To examine how much the Monarch scheduler has to be modified when accessing the runqueue by the VMM or the domain 0. We inspected rbtree.c of the source code of the Linux kernel 2.6.23. To access the domain U's CFS runqueue, the VMM or the domain 0 access the structure of the red brack tree. It is plausible to reuse the rbtree.c. Figure 5.13 shows the code to obtain the last node of the tree in rbtree.c. When accessing this structure from the VMM or the domain 0, We rewrite all pointer accesses via mapping domain's memory. Figure 5.14 shows the code to obtain the last node of the tree from the VMM. The code shows that

```
struct rb_node *rb_last(struct rb_root *root)
{
        struct rb_node *n;

        n = root->rb_node;
        if (!n)
                return NULL;
        while (n->rb_right)
                n = n->rb_right;
        return n;
}
```

Figure 5.13. A code fragment of red black tree library in the Linux kernel

every memory access has to be replaced by the memory map code. Next, we inspected whole rbtree.c. rbtree has 397 line of code. There is about 91 pointer access and the Monarch scheduler has to modify 91 places to accessing via memory map code.

5.10 The Comparison between the Monarch scheduler and the Central scheduler

We have implemented the the Central scheduler as the process in the domain 0 in Xen 3.3.0 [3].

Domain 0 is a privileged VM for managing the other VMs and it is often regarded as a part of the VMM.

5.10.1 Architecture

The Central scheduler is implemented as a userland process in domain 0, as shown in Figure 5.15. One advantage of this implementation is that any modification is not required in the other parts except that process, such as the VMM, the OS kernel in domain 0, and all domain Us. This is important to easily apply the Monarch scheduler to production systems. Another advantage is ease of development. The developers can easily develop custom scheduling policies by a trial-and-error approach. They can repeat modifying a scheduling policy and restarting a scheduler process without rebooting the

```
struct rb_node *rb_last(struct rb_root *root, struct domain* d)
{
        struct rb_node *n;

        n = map_domain_memory(d, root + offsetof(root, rb_node));

        if (!n) {
                unmap_domain_memory(n);
                return NULL;
        }

        while (true) {
            struct rb_node *right;
            right = map_domain_memory(d, n + offsetof(n, rb_right));

            if (!right) {
                 unmap_domain_memory(right);
                 break;
            }

            n = right;
        }

        return n;
}
```

Figure 5.14. An example code of red black tree library accessing domain U's memory by the VMM or the domain 0

whole system. If the Monarch scheduler is implemented inside the VMM, the developers need to reboot the system whenever they change their scheduling policies.

The Monarch scheduler invokes a registered scheduling function at regular intervals. Before that, it pauses target domains by using a hypervisor call and checks whether it can access kernel data consistently. The invoked function inspects kernel data in target domains and manipulates them to achieve a custom scheduling policy. After the inspection and manipulation, the function returns to the Monarch scheduler and then the Monarch scheduler continues all paused domains.

Since the scheduler process in domain 0 has to always run properly, the

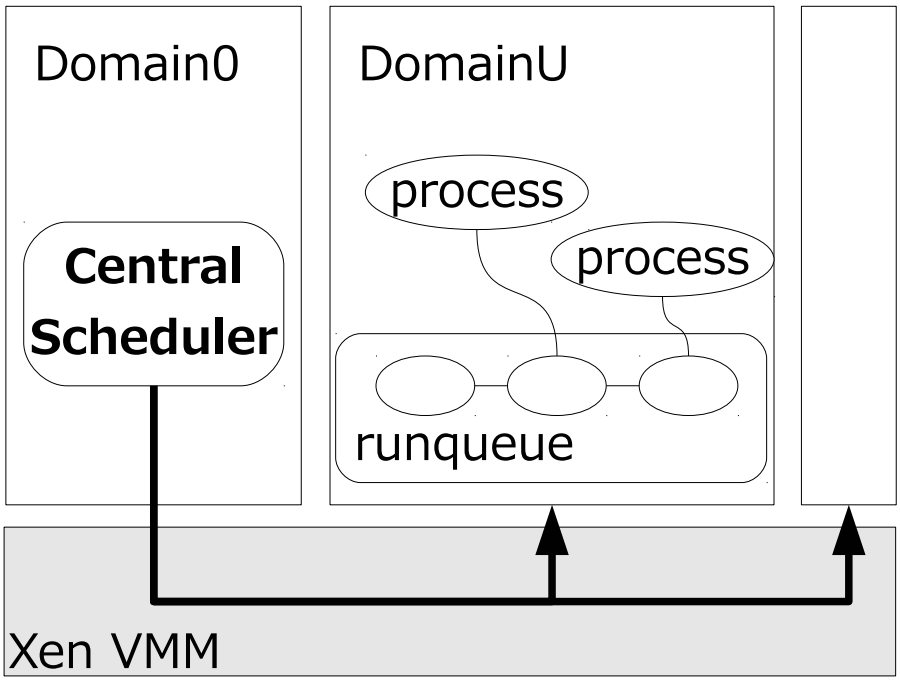


Figure 5.15. The architecture of the Central scheduler.

Monarch scheduler does not allow any custom scheduling policies to be applied to the scheduler process itself. In addition, the VMM allocates sufficient CPU time to domain 0 by exploiting the VM scheduler, so that sufficient CPU time is allocated to the scheduler process. We believe that such special treatment of the scheduler process is not a serious problem. The special treatment of only one process is simpler than that in cooperative scheduling, which has to specially treat scheduling threads in all VMs.

Xen manages the memory of domains by binding machine memory to pseudo-physical memory. Machine memory is physical memory installed in the machine and consists of a set of machine page frames. For each machine page frame, a machine frame number (MFN) is consecutively numbered from 0. Pseudo-physical memory is the memory allocated to domains and gives the illusion of contiguous physical memory to domains. For each physical page frame in each domain, a physical frame number (PFN) is consecutively numbered from 0.

In Xen, a process in domain 0 can map arbitrary machine page frames allocated to domain Us to its address space using a hypervisor call. To access a specific virtual address in a domain, it first translates a virtual address to

a PFN by looking up a page table in the domain. Since the page table is also located in the memory of the domain, the process looks up a PFN by mapping machine page frames used for the page table. The physical address of the page table is obtained from the `CR3` register. Next, the process looks up a MFN corresponding to the PFN by mapping a translation table from PFNs to MFNs, which is also located in the memory of the domain. Finally, the process obtains a local address where a machine page frame corresponding to the MFN is mapped.

The Monarch scheduler maintains the result of this translation from a virtual address in domain U to a local address in domain 0 as a cache. When it accesses the same or near virtual address in the same domain, it can obtain the local address from the cache. This is very efficient because any memory mapping is not necessary for the address translation. In addition to this cache, the Monarch scheduler maintains the memory pages mapped once from domain U, so that it can access local addresses obtained from the cache without any memory mapping. The Monarch scheduler invalidates the cache and unmaps all the memory pages in domain U before it continues the domain. The page table and the translation table in domain U can be changed by its guest OS while the domain is running.

When the Monarch scheduler accesses a certain field in a data structure, it maps only the memory page including the field. To do this, it calculates the virtual address of the field from the address of the data structure and the offset of the field. If the size of a data structure is more than that of a memory page, one data structure occupies multiple pages. Although these pages may be contiguous in pseudo-physical memory, they are not always contiguous in machine memory. Therefore, the Monarch scheduler has to map such pages one by one and it takes time depending on the number of pages.

5.10.2 Disadvantages of the Central scheduler

One disadvantage of this implementation is its high overheads. Accessing domain U's memory by the domain 0 is massively slow because the domain 0 must access domain U's memory repeatedly. If the domain 0 wants to access a virtual address of the domain U's memory, firstly the domain 0 must translate its virtual address into machine frame number by accessing domain U's page table. This is about 5 times must be accessed due to page table is 5 level on x86-64 architecture. We examined this overheads in the Section 5.10.

what is doing	time (μs)
stop and restart of DomainU	11.1
calculation of frame number corresponding to virtual address	58.8
map and unmap the page to process of Domain0	7.0
access of 1 word of mapped memory	0.0

Table 5.2. The breakdown of the time needed for accessing memory of DomainU from Domain0.

Another is that the Central scheduler cannot treat processes of the domain 0. Since the Central scheduler is implemented as the process in the domian 0, the Monarch scheduler does not make the processes the scheduler's target. If the Monarch scheduler suspends the process of the Monarch scheduler itself, the scheduler has reached into the state of deadlock. Not only the suspension but also lowing the priority may cause the problem. If the process of the Monarch scheduler runs in the low priority, the schedule may not happen periodically so that the accurate scheduling cannot be achieved.

5.10.3 Costs of Accessing Memory of DomainU

We conducted an experiment on the time of accessing memory of DomainU from Domain0. We measured the breakdown of the time by executing the following code 100 million times. Firstly, we stopped a DomainU, and calculated machine frame number corresponding to an address of a process' virtual address space. Secondly, we mapped the address to an address of a process of Domain0, and read one word from the mapped address. Lastly, we unmapped the mapped address, and restart the DomainU.

Table 5.2. shows an average time of each process. The experimental result suggests that the greatest part of time is the calculation of machine frame number from virtual address. It is because that it takes long time to lookup the page table of Guest OS.

5.10.4 The Accuracy of Scheduling Interval

We examined the accuracy of scheduling interval about Monarch Scheduler and the Central scheduler, show that the Monarch Scheduler can achieve the

more accurate scheduling interval. We compared the accuracy of two versions scheduler, one is implemented in the VMM and another is in the domain 0 which are set up to run regulary 10ms. The scheduler in the domain 0 which is not overloaded resulted in the 20ms of scheduling interval. It is considered that the process scheduling of the domain 0 may affect the interval. On the other hand, the scheduler in the VMM achieved the accurate 10ms scheduling interval.

Next, we tested the same experiment on the condition that Bonnie++ [76] ran in the domain U and the domain 0 and the domain U used the same physical CPU. In the case of the scheduler process in the domain 0, the scheduling interval increased to 32ms. On the other hand, in the case of the scheduler in the VMM, the scheduling interval did not increase, but remained 10ms. We think this is because the CPU time was not allocated for the scheduling process in domain 0 due to the load. The scheduler process in the VMM can schedule accurately processes even if the load is heavy.

5.10.5 The Accuracy of Measuring Process Times

We conducted experiments showing that measuring process times in the VMM is more accurate than doing in the domain 0. We ran two VMs, VM1 and VM2, with Linux 2.6.16.33 as guest OSes, and ran the process of infinite loop on each VM. We setup that these VMs share one physical CPU and their priorities of the VMs are the same level. We compared the CPU time used by the process in the VM1, between a value measured by the VMM and one measured by the domain 0. Because two same priority processes share one physical CPU, the actual CPU time used by the process in the VM1 shall become the half of the total execution time.

Figure 5.16 shows the execution time and the process times. As Monarch Scheduler does, the process time tracked by the VMM is accurate as the half of the total execution time. On the other hand, as the Central scheduler did, the process time obtained from a guest OS is not accurate as the same of the total execution time. This is because that the guest OS cannot understand the time which does not use physical CPUs.

5.10.6 The Proficiency of Process State Rewriting

We conducted experiments showing that the process state rewriting contributes to the more accurate process control. We examined the proportion of the process which is stopped by the stop operation in a constant time.

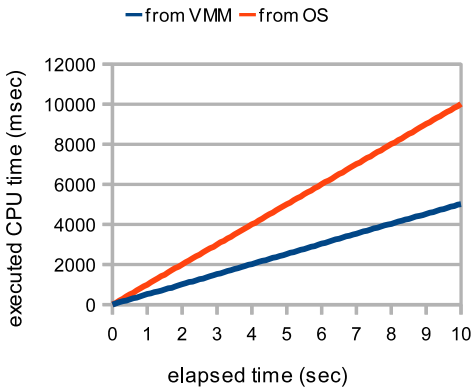


Figure 5.16. The difference between the process times obtained from a guest OS and that tracked by the VMM.

We examined two cases, one case consisted of only the runqueue operation as the Central scheduler did. Another case consisted of both the runqueue operation and the process state rewriting The target processes are the process of infinite loop and Bonnie++, we ran each process alone. The process of infinite loop always used CPU time, its state is always running because it ran alone. Bonnie++ used 53% of its time as blocked state because it had many I/O and waited for the I/O completion.

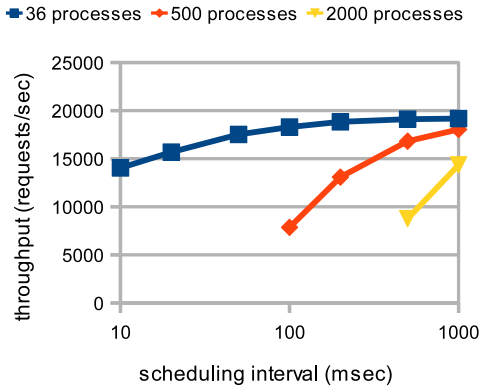
Figure 5.17(a) shows that only runqueue operation stopped only 11% of the process of infinite loop in 60 sec. However, using both the runqueue operation and the process state rewriting stopped the processes acurrately. This is because the process state rewriting enables to control the process which could not be controlled only by the runqueue operation. Figure 5.17(b) shows, in case of Bonnie++, only the runqueue opration could stop the 66% of the process in 1 sec, however, using both the runqueue opration and the process state rewriting could stop the 93% of the process in 1 sec. This shows the I/O heavy process could be controlled.

5.10.7 Performance Degradation by the Central scheduler

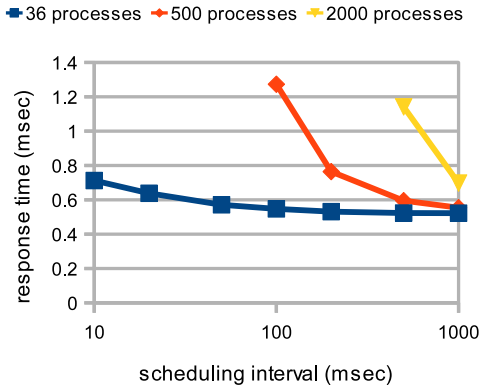
We performed experiments to exaine performance degradation by the Central scheduler. We conducted the same experiments as section 5.3

Figure 5.18 shows the results. When the scheduling interval was 10ms and the number of process is 36, the throuput of lighttpd degraded by 27%

and the response time increased by 27%. Even when the scheduling interval was 100ms, both the throuput and the response time degraded by more than 50%. These shows that if the number of processes increases, the scheduling interval cannot be shorten.

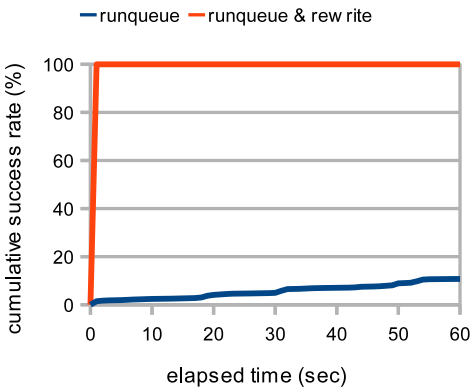


(a) Throuput

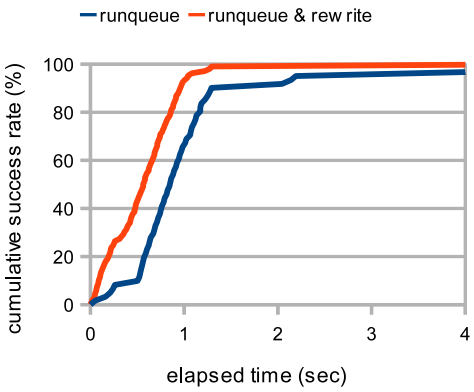


(b) Response Time

Figure 5.18. The performance degradation of a web server with the Central scheduler.



(a) infinite loop



(b) Bonnie++

Figure 5.17. The distribution of times elapsed for stopping a process.

Chapter 6

Experiments for VMCrypt

We performed experiments to measure the overheads of VMCrypt and confirm that VMCrypt prevents information leakage via domain 0. We used two PCs, each of which has one Intel Xeon processor 2.67 GHz with 8 cores, 12 GB of memory, a 1 TB of SATA HDD, and a Gigabit Ethernet NIC. We ran modified Xen 4.0.1 for the x86-64 architecture on these PCs. For domain 0, we allocated 6 GB of memory and ran Linux 2.6.32-5-xen-amd64. For domain U, we allocated 1 GB of memory if not specified in each experiment and ran para-virtualized Linux 2.6.32.27. These PCs were connected with a Gigabit Ethernet switch.

Through the experiments, VMCrypt used AES-XTS with a key size of 256 bits for the encryption of domain U's memory. We have implemented the AES-XTS support in the VMM but not yet fully optimized. The performance can be improved by using AES-NI, a set of special instructions for AES. To exclude the overhead of cryptographic operations, we also used the null cipher, which did not encrypt or decrypt data. For comparison, we conducted the experiments in the vanilla Xen.

6.1 Overhead of Constructing an Encrypted View

To examine the overhead of constructing an encrypted view, we measured the time needed for mapping and unmapping a memory page of domain U on domain 0. We performed this experiment for writable mapping, read-only mapping, and the mapping of uninitialized memory. VMCrypt encrypts and decrypts a writable page, only encrypts a read-only page, and only decrypts an uninitialized page, respectively. We repeated memory mapping and unmapping 100000 times. Figure 6.1 shows the mean time.

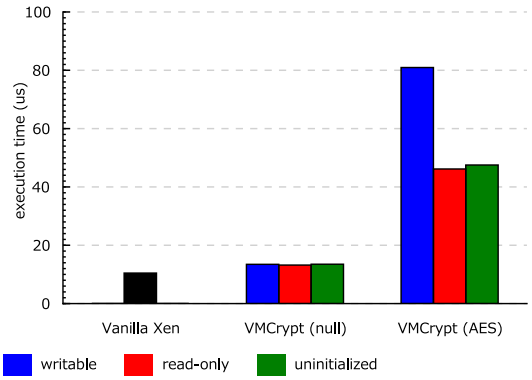
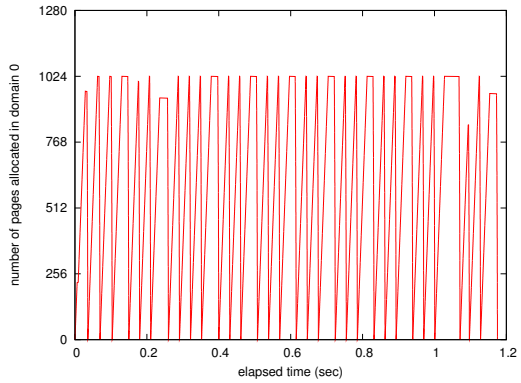


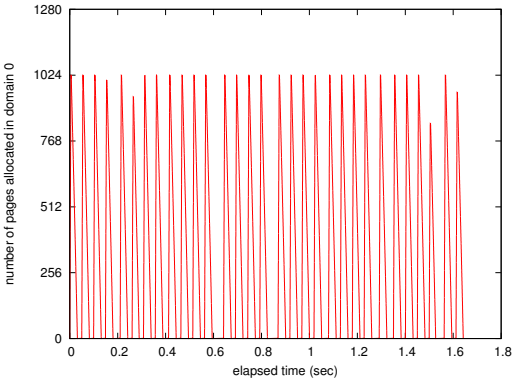
Figure 6.1. Time for the domain 0's mapping a page of a domain U.

When VMCrypt used the null cipher, the execution time increased by 3 μs in comparison with the vanilla Xen. This overhead comes from examining the encryption bitmap and replicating a page. The optimization for encrypted replication was not effective because the memory copies were completed only on the CPU cache. When VMCrypt performed both encryption and decryption of AES, the execution time was 81 μs, which was 7.7 times longer than that in the vanilla Xen. When the optimization was enabled, the execution time was reduced to less than 60 % of the non-optimized case. This shows that our optimization for reducing cryptographic operations is effective.

6.2 Memory Overhead for an Encrypted View



(a) VM suspend



(b) VM resume

Figure 6.2. The number of extra pages used for replication during VM suspend and VM resume.

We examined the number of extra pages allocated for an encrypted view. When domain 0 maps domain U's pages, the VMM allocates new pages in domain 0 for encrypted replication. At worst, the number of the allocated pages can be equal to the total number of domain U's pages. In this experiment, we allocated 128 MB of memory to domain U. Figure 6.2(a) and Figure 6.2(b) show the changes of the number of extra pages while we were suspending and resuming domain U, respectively. The maximum number of extra pages was 1024. This is because the suspend and resume programs mapped 1024 pages at once and unmapped all of them after the memory manipulation. 1024 pages are 4 MB of memory and not so large.

6.3 Overhead for VM Boot

We measured the time needed for building domain U at the boot time. The build time we measured was from when we started creating domain U until the VMM completed the unpause hypercall. We changed the allocated memory size of domain U from 26 MB to 4 GB. To exclude the impact of the page cache in domain 0, we recorded the build times after the second boot. The build times are shown in Figure 6.3 for VMCrypt with AES and the vanilla Xen. We did not measure the time for VMCrypt with the null cipher because VMCrypt does not perform any cryptographic operations on booting domain U. The build time was slightly proportional to the memory size of domain U and less than one second in any cases. The overhead due to VMCrypt was only 1 % when the memory size of domain U was 4 GB.

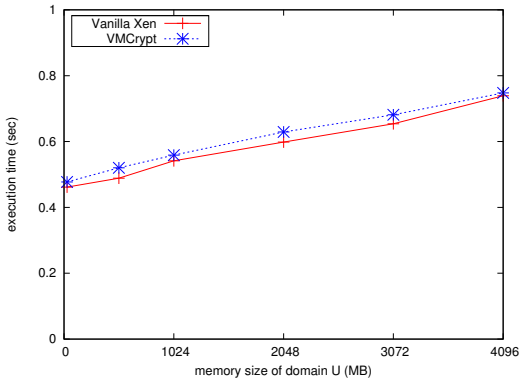


Figure 6.3. The time for booting domain U.

6.4 Overheads for VM Suspend and Resume

First, we measured the time needed for suspending domain U. We changed the memory size of domain U as in the above experiment. Figure 6.4 shows the results when we suspended domain U. The suspend time was proportional to the amount of domain U's memory. The suspend operation in VMCrypt with the null cipher was as fast as that in the vanilla Xen. Even the overhead due to VMCrypt with AES was only 7 % when the memory size of domain U was 4 GB. This is because the overhead of AES is hidden by slow disk I/O. On suspending, domain 0 maps pages of domain U and writes the contents to the disk asynchronously. While the VMM encrypts the pages, domain 0 can write the memory image to the disk. When we used tmpfs, a RAM disk,

to reduce the impact of disk I/O, the suspend time in VMCrypt with AES became 7 times longer than that in the vanilla Xen.

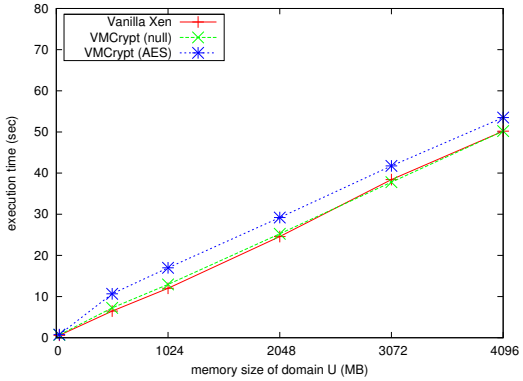


Figure 6.4. The time for suspending domain U.

Next, we measured the time needed for resuming domain U. Figure 6.5 shows the resume times when we resumed domain U from the file on the hard disk. Like suspend, the resume time was also proportional to the memory size of domain U. The overhead of VMCrypt with the null cipher was negligible, but the resume time in VMCrypt with AES was 90 % longer than that in the vanilla Xen. This is due to sequential processing of disk reads and decryption. On resuming, domain 0 reads the saved file, maps pages of domain U, writes the memory image to them, and unmaps them. Then the VMM has to decrypt the unmapped pages synchronously. The overhead of the decryption is not hidden by disk I/O at all.

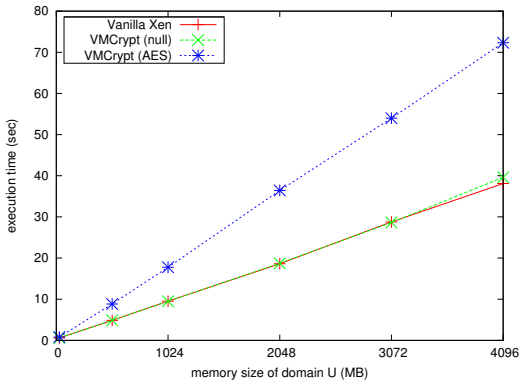


Figure 6.5. The time for resuming domain U.

6.5 Overhead for VM Migration

We measured the time needed for migrating domain U between two hosts. We used two machines with the identical hardware, which were connected with a Gigabit Ethernet switch. For comparison, we also performed migration with SSL. Xen provides the mechanism of migrating domain U with an SSL connection. Although SSL can prevent information leakage on the network, domain 0 at both hosts can still steal information. Transferred data on the SSL connection is decrypted at domain 0. Xen used AES with a key size of 256 bits for encryption.

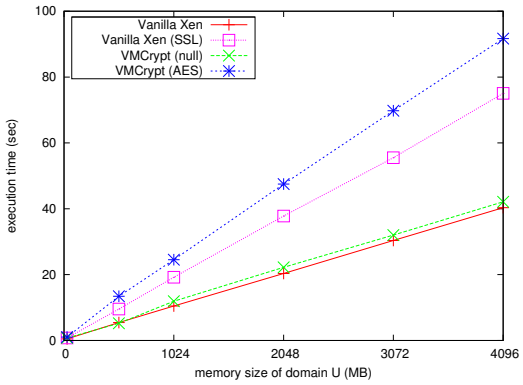


Figure 6.6. The time for migrating domain U.

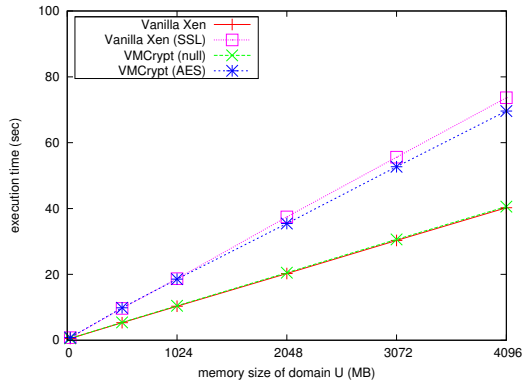
We performed non-live migration, which stops domain U during migration, and the results are shown in Figure 6.6. We changed the memory size of domain U as in the other experiments. Like suspend and resume, the migration time was proportional to the memory size of migrated domain U. The overhead of VMCrypt with the null cipher was 4 % while the migration time in VMCrypt with AES was 2.3 times longer than that in the vanilla Xen. Moreover, the migration time in VMCrypt with AES was 22 % slower than that in the vanilla Xen with SSL. The reason is that the sender has to wait for decryption at the receiver in VMCrypt. In non-live migration, the sender transfers 16 MB of the memory image at once, but the receiver decrypts only 4 MB at once. Therefore, the sender blocks until the receiver completes the decryption because it cannot transfer the data exceeding the buffer size at the receiver. When we modified the migration program so that the sender transfers 4 MB at once, the migration time became almost the same as that in the vanilla Xen with SSL. This problem does not happen in the vanilla Xen with SSL because the sender encrypts the next data while

the receiver decrypts received data.

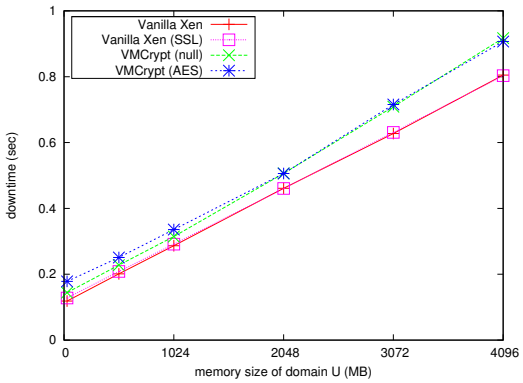
6.6 Overhead for Live Migration

With VMCrypt, we performed live migration, which transfers the memory image without stopping domain U and reduces the downtime of domain U. For comparison, we also performed migration with SSL. Xen provides the mechanism of migrating domain U via an SSL connection. Although SSL can prevent information leakage on the network, domain 0 at both hosts can still steal information. Transferred data on the SSL connection is decrypted at domain 0. Xen used AES with a key size of 256 bits for encryption.

We measured the time needed for migrating domain U between two hosts. We changed the allocated memory size of domain U from 26 MB to 4 GB. The results are shown in Figure 6.7(a). The migration time was proportional to the memory size of migrated domain U. The overhead of VMCrypt with the null cipher was only 1 %. With AES, however, the migration time was 1.7 times longer than that in the vanilla Xen. These results mean that encryption and decryption degraded the performance largely. Note that the performance in VMCrypt with AES is almost the same as that in the vanilla Xen with SSL. The reason why the vanilla Xen with SSL is slightly slower than VMCrypt with AES is that SSL checks the data integrity as well.



(a) Migration time



(b) Downtime

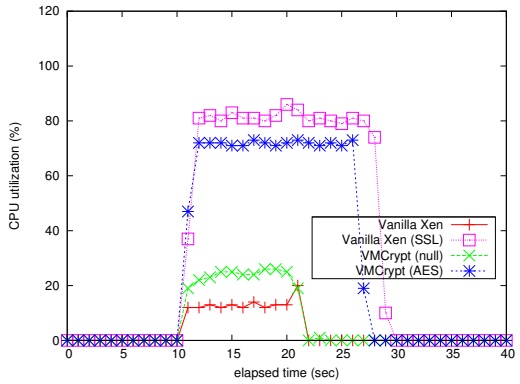
Figure 6.7. The performance of live migration.

Next, we measured the downtimes due to live migration of domain U. Here, the downtime is the time from when domain U is stopped at the source host until it is restarted at the destination host. As shown in Figure 6.7(b), the downtime was also proportional to the memory size of domain U. In VMCrypt with AES, it was still less than one second even when the memory size was 4 GB. It was 13 % longer than that in the vanilla Xen. These overheads come from the transfer of the remaining memory after domain U is stopped. The time for which VMCrypt deals with such memory pages is included in the downtime.

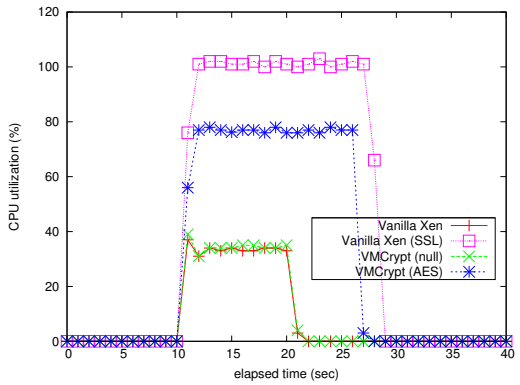
Third, we measured the number of re-transferred pages, which increases due to the re-transfer of the embedded bitmap. On average, 29.5 extra pages were re-transferred, but this is negligible to the total pages of domain U.

Finally, we measured the CPU utilization of domain 0 during live migration. Figure 6.8(a) and Figure 6.8(b) show the CPU utilization of domain 0

at the source and destination hosts, respectively. The measured CPU time includes the CPU time consumed in the VMM by hypercalls issued by domain 0. Compared with the vanilla Xen, the CPU utilization in VMCrypt with AES only increased by 35 % and 30 % at the source and destination hosts, respectively. However, that in the vanilla Xen with SSL increased by 70 % and 65 %, respectively. One of the reasons of such high CPU utilization with SSL is the integrity check.



(a) Source host



(b) Destination host

Figure 6.8. The CPU utilization during live migration.

6.7 Performance Degradation of Domain U

We measured the impact of VMCrypt on the performance of domain U. VMCrypt does not perform encryption or decryption unless domain 0 maps or unmaps pages of domain U, but the VMM checks several operations even

when domain U modifies page tables. We ran two benchmarks, lmbench [51] and UnixBench [86], on domain U in the vanilla Xen and VMCrypt. Figure 6.9 shows the performance degradation in lmbench due to VMCrypt and Figure 6.10 shows the one in unixbench. Context switching in VMCrypt was 1.8 % slower than in the vanilla Xen, but the overall average of performance degradation in VMCrypt was 0.7 %. For UnixBench, the overall average of performance degradation in VMCrypt was also 0.7 %.

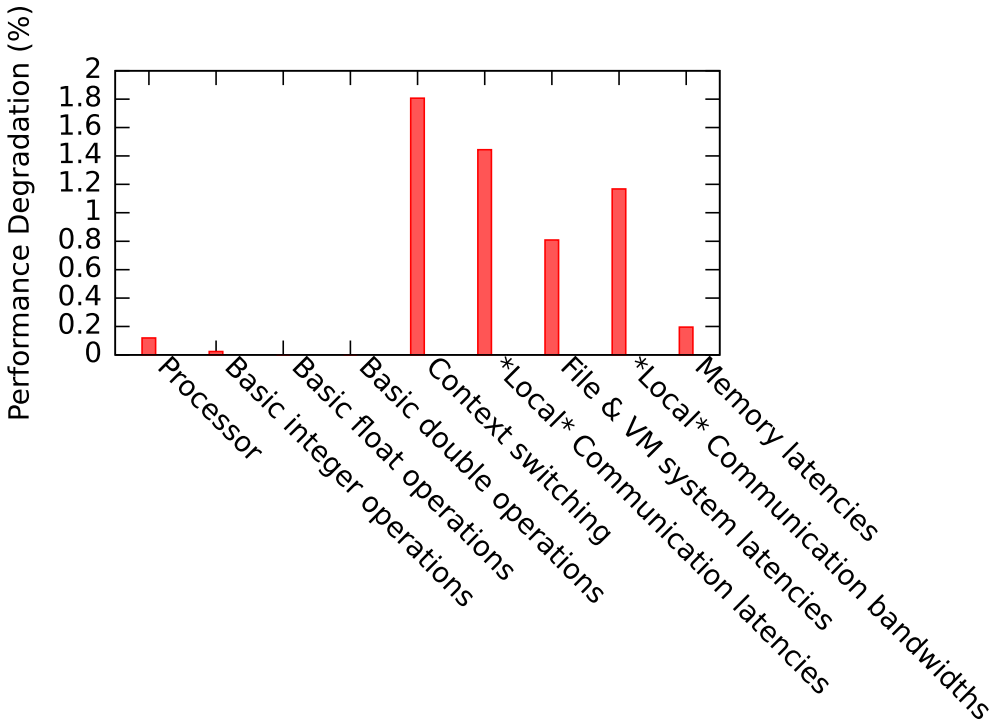


Figure 6.9. LmBench: Performance degradation of domain U by VMCrypt.

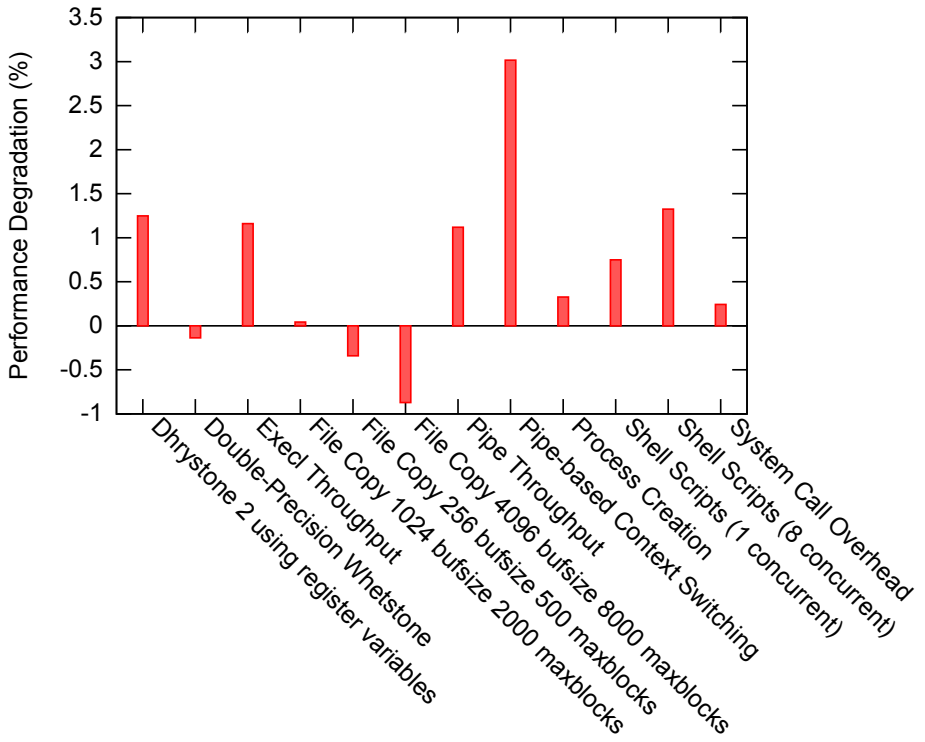


Figure 6.10. UnixBench: Performance degradation of domain U by VMCrypt.

Next, we examined how live migration in VMCrypt affected the performance of a web server running in domain U. We ran the lighttpd web server [34] in domain U and measured its throughput with ApacheBench [95] in a client host. The client host had one Intel Core 2 Quad processor 2.83 GHz, 8 GB of memory, and a Gigabit Ethernet NIC. The server and client hosts were connected with a Gigabit Ethernet switch. Figure 6.11 shows that the throughput in VMCrypt with AES was higher than that in the vanilla Xen. This is because live migration in the vanilla Xen occupies the network bandwidth and the web server cannot use network sufficiently. However, the throughput in VMCrypt with AES is lower than that in the vanilla Xen with SSL. In VMCrypt, the sender transfers 4 MB of the memory image at once and causes bursty network traffic intermittently. This bursty traffic affects the throughput of the web server. With SSL, in contrast, the traffic is not bursty because the sender slowly transfers the data while encrypting.

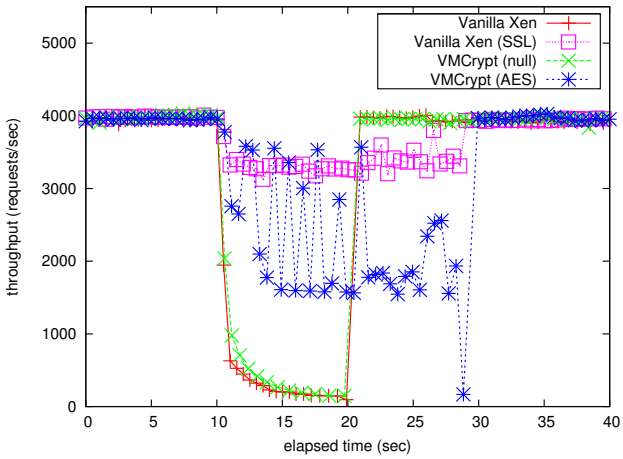


Figure 6.11. The throughput of a web server in domain U during live migration.

6.8 Overhead of Remote Attestation

We examined the overhead of remote attestation, which is performed only when a host is booted. First, we measured the time needed for the measurement of the VMM using TrustedGRUB 1.1.5 [85]. TrustedGRUB is a boot loader that calculates the SHA-1 hash value of not only the VMM but also the kernel in domain 0 and stores the value in the TPM [100]. We compared the reboot time for TrustedGrub with that for GRUB Legacy [20]. The time for the measurement was 0.5 sec.

Next, we attempted the verification of the masurement by the TC, but OpenPTS [59] did not work in our experimental environment. According to the literature [13], the verification process takes about one second. The time should depend on the network performance.

6.9 Leakage Tests with VMCrypt

We confirmed that VMCrypt prevented information leakage from domain U's memory.

6.9.1 Finding Keys from Processes' Memory

```
root@mach# aeskeyfind quattro1.dump
bb2e3fe052aedffe8ddffd3fbcfa7d09
ea9b7567ae60e300d00bde56096d3170
Keyfind progress: 100%
```

Figure 6.12. Finding AES shared keys from domain U's memory.

```
root@mach# rsakeyfind quattro1.dump
FOUND PRIVATE KEY AT 3804dec0
...
```

Figure 6.13. Finding RSA private keys from domain U's memory.

We attempted an attack to find AES shared keys used by OpenSSH [97] processes. First, we logged in domain U using SSH, so that the SSH server generated a shared key of the AES128-CBC encryption method by default. Next, we obtained the memory dump of the domain U by executing xm dump-core from domain 0. The memory of running processes is included in the memory dump. Then we used the aeskeyfind tool [23, 63] to find AES keys from the memory dump. We could not find any keys due to VMCrypt. Without the memory encryption by VMCrypt, we could obtain several keys as in Figure 6.12.

Similarly, we attempted an attack to find RSA private keys generated by OpenSSL [98]. First, we generated an X.509 certificate signing request [33] by executing the openssl command in domain U. Next, we obtained the memory dump of the domain U and used the rsakeyfind tool [23, 64] to find RSA private keys. Even if processes are terminated, a part of their data still resides in the memory. When VMCrypt was not enabled, we could find several keys as in Figure 6.13. VMCrypt prevented the leakage of private keys from domain U's memory.

6.9.2 Obtaining Passwords on the Page Cache

```
root@mach# strings quattro1.img | grep 'root:\$'
acroot:$6$aCJuBx5O$5HqjJyEGM.hDUBnczt2J.j6jN41.G02k
HlNXHZrurOZpqL/Elnbc489ZrZqLD2gsPDB.yVcK6trNXAquhKF
kG0:14879:0:99999:7:::
```

Figure 6.14. Finding a shadow password from domain U's memory.

We attempted an attack to obtain shadow passwords from the page cache in domain U's memory. First, we logged in domain U as root, so that the contents of `/etc/shadow` were stored in the page cache. Next, we suspended the domain U by executing xm save from domain 0 and the memory image was saved into a file. The page cache is included in the file. Then we applied the strings [18] command to the file and searched the string "`root:$`" from the result with the grep [19] command. A root password in `/etc/shadow` begins with this string. As shown in Figure 6.14, we could find a root password when the page cache in the saved memory image was not encrypted by VMCrypt. However, VMCrypt disabled such a password search.

Chapter 7

Conclusion

In this thesis, we proposed coordinated and secure server consolidation, which is enabled by the Monarch scheduler and VMCrypt. The Monarch scheduler mediates CPUs among processes in different VMs to achieve system-wide scheduling policies. To control the execution of processes, it suspends and resumes processes by using a technique called direct kernel object manipulation (DKOM). To hide the details of DKOM for various guest OSes, the Monarch scheduler provides a high-level API for writing scheduling policies. VMCrypt prevents the management VM from stealing sensitive information in the VM's memory. VMCrypt encrypts the VMs' memory only for the management VM and allows the administrators uses the existing management software as is in the management VM. Although the existing management software can basically run for encrypted memory, it requires accessing unencrypted contents only for several memory regions. Therefore, VMCrypt does not encrypt such memory regions, which are automatically identified and maintained during the life cycle of a VM.

Future Work

The Monarch Scheduler

One of the future work is supporting Windows guest OSes completely. In the current implementation, the Monarch scheduler can manipulate Windows processes in the run queues but cannot suspend ones in the running or blocked state. We need different techniques from ones used for Linux. Another direction is developing various system-wide scheduling policies using the Monarch scheduler, such as a fair share scheduler [40]. Possible scheduling policies also depends on scheduling algorithms in guest OSes. In addition, separating a policy from the VMM is needed. In the current implementation, policies should be compiled and linked with the VMM in advance. Changing a policy in the VMM with another securely at runtime is important. It needs not only a dynamic linking technique but also verifying that the injected code is truely secure.

VMCrypt

One of the future work is to reduce the overhead of memory decryption. In the current implementation, pages mapped on the management VM are decrypted on unmapping synchronously. To overlap the decryption with I/O, the VMM should decrypt unmapped pages asynchronously. To do so, we need to implement multithreading support in the Xen VMM. Second, current CPUs have a set of special instructions for encryption such as AES, called AES-NI [32]. We believe that the performance can be improved by using AES-NI. Supporting fully-virtualized guest OSes is also necessary for applying VMCrypt to real environments. Unlike para-virtualized ones, the management VM needs to access other memory regions of user VMs such as video memory and DMA memory. The VMM has to identify the memory regions used for framebuffers and DMA as unencrypted pages while preventing information leakage.

Bibliography

[1] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 35(5):43–56, 2001.

[2] F. Baiardi, D. Maggiari, D. Sgandurra, and F. Tamberi. Psycotrace: Virtual and transparent monitoring of a process self. *Proc. Euromicro Int. Conf. Parallel, Distributed and network-based Processing*, pages 393–397, 2009.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. Symp. Operating Systems Principles*, pages 164–177, 2003.

[4] BitVisor. BitVisor – A Secure and Lightweight Hypervisor – . `http://www.bitvisor.org/`.

[5] J. Bonwick. The Slab Allocator: An Object-caching Kernel Memory Allocator. In *Proc. USENIX Summer 1994 Technical Conf.*, pages 6–6, 1994.

[6] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *In Proc. IEEE Symposium on Security and Privacy*, 2011.

[7] H. Brian and N. Kara. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008.

[8] bugcheck. GREPEXEC: Grepping Executive Objects from Pool Memory. http://uninformed.org/?v=4&a=2&t=pdf, 2006.

[9] J. Butler. DKOM (Direct Kernel Object Manipulation). Black Hat Windows Security, 2004.

[10] C. Weinhold and H. Härtig. VPFS: Building a Virtual Private File System with a Small Trusted Computing Base. In *Proc. European Conf. Computer Systems*, pages 81–93, 2008.

[11] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2008.

[12] ClamWin Team. ClamWin Free Antivirus. http://www.clamwin.com/.

[13] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: A Scalable Fault Tolerant Network Manager. In *Proc. Symp. Networked Systems Design & Implementation*, 2011.

[14] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *SIGOPS Oper. Syst. Rev.*, 33(5):261–276, December 1999.

[15] DWARF Standards Committee. The DWARF Debugging Standard. http://dwarfstd.org/.

[16] L. Eggert and J. Touch. Idletime Scheduling with Preemption Interval. In *Proc. Symp. Operating System Principles*, pages 249–262, 2005.

[17] Free Software Foundation, Inc. Computing billions of PI digits using GMP. http://gmplib.org/pi-with-gmp.html.

[18] Free Software Foundation, Inc. GNU Binutils. http://www.gnu.org/software/binutils/.

[19] Free Software Foundation, Inc. grep. http://www.gnu.org/software/grep/.

[20] Free Software Foundation, Inc. GRUB Legacy. `http://www.gnu.org/software/grub/grub-legacy.html`.

[21] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symp.*, pages 191–206, 2003.

[22] Google, Inc. Google Desktop. `http://desktop.google.com/`.

[23] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proc. USENIX Security Symp.*, pages 45–60, 2008.

[24] Hidekazu Tadokoro, Kenichi Kourai and Shigeru Chiba. Process Scheduling among Virtual Machines. *IPSJ Transactions on Advanced Computing Systems (ACS)*, 1(2):124–135, 8 2008.

[25] Hidekazu Tadokoro, Kenichi Kourai and Shigeru Chiba. A Secure System-wide Process Scheduler across Virtual Machines. In *Proc. the 16th IEEE Pacific Rim Intl. Symp. on Dependable Computing (PRDC'10)*, pages 27–36, December 2010.

[26] Hidekazu Tadokoro, Kenichi Kourai and Shigeru Chiba. A Practical Process Scheduler across Virtual Machines. *IPSJ Transactions on Advanced Computing Systems (ACS)*, 4(3):100–114, 5 2011.

[27] Hidekazu Tadokoro, Kenichi Kourai and Shigeru Chiba. Preventing Information Leakage from Virtual Machines' Memory in IaaS Clouds. *IPSJ Transactions on Advanced Computing Systems (ACS)*, 5(4):1–11, 2012.

[28] M. Hirabayashi. Hyper Estraier: a Full-text Search System for Communities. `http://hyperestraier.sourceforge.net/`.

[29] Hiroshi Yamada and Kenji Kono. FoxyTechnique: Tricking Operating System Policies with a Virtual Machine Monitor. In *Proc. Int'l Conf. Virtual Execution Environments*, pages 55–64, 2007.

[30] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed

Multimedia Applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[31] IEEE P1619 Security in Storage Working Group. IEEE P1619. `http://siswg.net/`, 2007.

[32] Intel Corp. Intel Advanced Encryption Standard Instructions. `http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/`.

[33] ITU-T. Public-key and attribute certificate frameworks. `http://www.itu.int/rec/T-REC-X.509-200508-I`, 2005.

[34] J. Kneschke. lighttpd. `http://www.lighttpd.net/`.

[35] Jacob Gorm Hansen and Eric Jul. Self-migration of operating systems. In *Proc. the 11th workshop on ACM SIGOPS European workshop*, 2004.

[36] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection through VMM-based "Out-of-the-box" Semantic View Reconstruction. In *Proc. Conf. Computer and Communications Security*, pages 128–138, 2007.

[37] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proc. USENIX Annual Technical Conf.*, pages 1–14, 2006.

[38] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting Past and Present Intrusions through Vulnerability-specific Predicates. In *Proc. Symp. Operating Systems Principles*, pages 91–104, 2005.

[39] P. Kamp and R. Watson. Jails: Confining the Omnipotent Root. In *Proc. Int. SANE Conf.*, 2000.

[40] Kay, J. and Lauder, P. A fair share scheduler. *Commun. ACM*, 31(1):44–55, January 1988.

[41] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 261–276, 1999.

[42] D. Kim, H. Kim, M. Jeon, E. Seo, and J. Lee. Guest-aware Priority-based Virtual Machine Scheduling for Highly Consolidated Server. In *Proc. Int. Euro-Par Conf. Parallel Processing*, pages 285–294, 2008.

[43] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware Virtual Machine Scheduling for I/O Performance. In *Proc. Int. Conf. Virtual Execution Environments*, pages 101–110, 2009.

[44] Kim, Gene H. and Spafford, Eugene H. The Design and Implementation of Tripwire: a File System Integrity Checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[45] Y. Kinebuchi, M. Sugaya, S. Oikawa, and T. Nakajima. Task Grain Scheduling for Hypervisor-Based Embedded System. In *Proc. Int. Conf. High Performance Computing and Communications*, 2008.

[46] KVM Project. KVM: Kernel Based Virtual Machine . `http://www.linux-kvm.org/`.

[47] Leonidas J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[48] C. Li, A. Raghunathan, and N. K. Jha. Secure Virtual Machine Execution under an Untrusted Management OS. In *Proc. Intl. Conf. Cloud Computing*, pages 172–179, 2010.

[49] C. Li, A. Raghunathan, and N. K. Jha. A Trusted Virtual Machine in an Untrusted Management Environment. *IEEE Transactions on Services Computing*, 2011. IEEE Computer Society Digital Library.

[50] Linux Kernel newbies. Linux 2 6 23. `http://kernelnewbies.org/Linux_2_6_23`.

[51] L. McVoy and C. Staelin. lmbench. `http://www.bitmover.com/lmbench/`.

[52] Microsoft. Windows Virtual PC. `http://www.microsoft.com/windows/virtual-pc/`.

[53] Microsoft Corp. Debugging Tools for Windows. `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`.

[54] Microsoft Corp. SQL Server.

[55] Microsoft Corp. Windows ISV Software Security Defenses. http://msdn.microsoft.com/en-us/library/bb430720.aspx.

[56] mlocate. mlocate: a locate/updatedb implementation. https://fedorahosted.org/mlocate/.

[57] Moving Picture Experts Group. MPEG-4 description: ISO/IEC JTC1/SC29/WG11 N4668. http://mpeg.chiariglione.org/standards/mpeg-4/mpeg-4.htm, 2002.

[58] MPlayer Team. MPlayer – The Movie Player. http://www.mplayerhq.hu/.

[59] S. Munetoh. Open Platform Trust Services. http://sourceforge.jp/projects/openpts/.

[60] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security through Disaggregation. In *Proc. Intl. Conf. Virtual Execution Environments*, pages 151–160, 2008.

[61] Jr. N. Petroni and M. Hicks. Automated Detection of Persistent Kernel Control-flow Attacks. In *Proc. Conf. Computer and Communications Security*, 2007.

[62] N. Quynh and K. Suzaki and R. Ando. eKimono: A Malware Scanner for Virtual Machines. In *HITB SecConf 2009*, 2009.

[63] Nadia Heninger and Ariel Feldman. AESKeyFinder: Tool for locating AES keys in a captures memory image. https://citp.princeton.edu/research/memory/.

[64] Nadia Heninger and J. Alex Halderman. RSAKeyFinder: Locating BER-encoded RSA private keys in memory images. https://citp.princeton.edu/research/memory/.

[65] T. Newhouse and J. Pasquale. A User-Level Framework for Scheduling within Service Execution Environments. In *Proc. Int. Conf. Services Computing*, pages 311–318, 2004.

[66] T. Newhouse and J. Pasquale. ALPS: An Application-Level Proportional-share Scheduler. In *Proc. Int. Symp. High Performance Distributed Computing*, pages 279–290, 2006.

[67] NTsyslog. Windows NT/2000/XP syslog service. http://ntsyslog.sourceforge.net/.

[68] oldnewthing. Why are process and thread IDs multiples of four? http://blogs.msdn.com/oldnewthing/archive/2008/02/28/7925962.aspx.

[69] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. Int. Conf. Distributed Computing Systems*, pages 22–30, 1982.

[70] P. Barham and B. Dragovic and K. Fraser and S. Hand and T. Harris and A. Ho and R. Neugebauer and I. Pratt and A. Warfield. Xen and the Art of Virtualization. In *Proc. Symp. Operating Systems Principles*, pages 164–177, 2003.

[71] B. Payne, M. Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *Proc. Annual Conf. Computer Security Applications*, pages 385–397, 2007.

[72] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proc. Symp. Security and Privacy*, pages 233–247, 2008.

[73] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proc. USENIX Security Symp.*, 2004.

[74] Quynh Nguyen Anh, Takefuji Yoshiyasu. A Novel Approach for a File-System Integrity Monitor Tool of Xen Virtual Machine. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 194–202, 2007.

[75] Quynh, Nguyen Anh, Takefuji, Yoshiyasu. Towards a Tamper-Resistant Kernel Rootkit Detector. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 276–283, 2007.

[76] R. Coker. Bonnie++. http://www.coker.com.au/bonnie++/.

[77] ReactOS Foundation. ReactOS. http://www.reactos.org/.

[78] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proc. Conf. Computer and Communications Security*, pages 199–212, 2009.

[79] F. Rocha and M. Correia. Lucy in the Sky without Diamonds: Stealing Confidential Data in the Cloud. In *Proc. Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments*, 2011.

[80] rsyslog. The enhanced syslogd for Linux and Unix rsyslog. `http://www.rsyslog.com/`.

[81] Rudolf Bayer. Symmetric binary B-Trees: Data structure and maintenance algorithms. In *Acta Informatica*, volume 1, pages 290–306, 1972.

[82] M. Russinovich and D. Solomon. Microsoft Windows Internals, Fifth Edition: Covering Windows Server 2008 and Windows Vista.

[83] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *Proc. Workshop on Hot Topics in Cloud Computing*, 2009.

[84] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. Bitvisor: A thin hypervisor for enforcing i/o device security. In *Proc. Intl. Conf. Virtual Execution Environments*, pages 121–130, 2009.

[85] Sirrix AG. TrustedGRUB. `https://projects.sirrix.com/trac/trustedgrub/`.

[86] I. Smith. UnixBench. `http://code.google.com/p/byte-unixbench/`.

[87] Sony Computer Entertainment Inc. PlayStation Portable. `http://www.jp.playstation.com/psp/`.

[88] Sourcefire, Inc. Clam AntiVirus. `http://www.clamav.net/`.

[89] Stephen S. Yau and Ho G. An. Confidentiality Protection in Cloud Computing Systems. *International Journal of Software and Informatics*, 4(4):351–365, 2010.

[90] Stephen T. Jones and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 14–24, 2006.

[91] syslog-ng. Nsyslog-ng - Multiplatform Syslog Server and Logging Daemon. http://www.balabit.com/network-security/syslog-ng.

[92] syslogd. Kernel and system logging daemons. http://www.infodrom.org/projects/sysklogd/.

[93] A. Tamches and B. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proc. Symp. Operating Systems Design and Implementation*, pages 117–130, 1999.

[94] Tetsuya Yoshida, Hiroshi Yamada, and Kenji Kono. FoxyLargo: Slowing Down CPU Speed with a Virtual Machine Monitor for Embedded Time-Sensitive Software Testing. In *In Proc. of the 2008 Int'l Workshop on Virtualization Technology(IWVT'08)*, pages 1–11, June 2008.

[95] The Apache Software Foundation. Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/.

[96] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[97] The OpenSSH Project. OpenSSH. http://www.openssh.org/.

[98] The OpenSSL Project. OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org/.

[99] Tripwire, Inc. tripwire. http://www.tripwire.com/.

[100] Trusted Computing Group. TPM Main Specification Version 1.2. http://www.trustedcomputinggroup.org/.

[101] VMware, Inc. VMWare. http://www.vmware.com/.

[102] VMware, Inc. VMware vSphere for Enterprise. http://www.vmware.com/products/vsphere/mid-size-and-enterprise-business/overview.html.

[103] XenSource, Inc. Credit-Based CPU Scheduler. http://wiki.xen.org/wiki/CreditScheduler.

[104] J. Yang and K. G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis. In *Proc. Intl. Conf. Virtual Execution Environments*, pages 71–80, 2008.

[105] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proc. Symp. Operating Systems Principles*, pages 203–216, 2011.

[106] J. Zhang and M. Wong. Database Test 3. `http://osdldbt.sourceforge.net/`.