Stony Brook University

OS-level Virtualization and Its Applications

A Dissertation Presented

by

Yang Yu

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2007

Stony Brook University

The Graduate School

Yang Yu

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation.

Professor Tzi-cker Chiueh – Dissertation Advisor
Department of Computer Science

Professor R. Sekar – Chairperson of Defense
Department of Computer Science

Professor Scott Stoller
Department of Computer Science

Professor Xin Wang
Department of Electrical and Computer Engineering,
Stony Brook University

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

OS-level Virtualization and Its Applications

by
Yang Yu

Doctor of Philosophy
in
Computer Science

Stony Brook University

2007

OS-level virtualization is a technology that partitions the operating system to create multiple isolated *Virtual Machines* (VM). An OS-level VM is a virtual execution environment that can be forked instantly from the base operating environment. OS-level virtualization has been widely used to improve security, manageability and availability of today's complex software environment, with small runtime and resource overhead, and with minimal changes to the existing computing infrastructure.

A main challenge with OS-level virtualization is how to achieve strong isolation among VMs that share a common base OS. In this dissertation we study major OS components of Windows NT kernel, and present a *Feather-weight Virtual Machine* (FVM), an OS-level virtualization implementation on Windows platform. The key idea behind FVM is access redirection and copy-on-write, which allow each VM to read from the base environment but write into the VM's private workspace. In addition, we identify various communication interfaces and confine them in the scope of each individual VM. We demonstrate how to accomplish these tasks to isolate different VMs, and evaluate FVM's performance overhead and scalability.

We present five applications on the FVM framework: secure mobile code execution service, vulnerability assessment support engine, scalable web site testing, shared binary service for application deployment and distributed *Display-Only File Server*. To prevent malicious mobile code from compromising desktop's integrity, we confine the execution of untrusted content inside a VM. To isolate undesirable side effects on production-mode network service during vulnerability scans, we

clone the service to be scanned into a VM, and invoke vulnerability scanners on the virtualized service. To identify malicious web sites that exploit browser vulnerabilities, we use a web crawler to access untrusted sites, render their pages with browsers running in VMs, and monitor their execution behaviors. To allow Windows desktop to share binaries that are centrally stored, managed and patched, we launch shared binaries in a special VM whose runtime environment is imported from a central binary server. To protect confidential files in a file server against information theft by insiders, we ensure that file viewing/editing tools run in a client VM, which grants file content display but prevents file content from being saved on the client machine. In this dissertation, we show how to customize the generic FVM framework to accommodate the needs of these applications, and present experimental results that demonstrate their performance and effectiveness.

To my parents, my wife, and my son.

Contents

List of Tables

List of Figures

Acknowledgments

First, I would like to thank my respectable advisor, Professor Tzi-cker Chiueh, who supports me financially and directs my PhD study for the past five years. Professor Chiueh always leads me to novel ideas and valuable research projects, which benefit my technical skills of both research and engineering in computer system area. Professor Chiueh also gives me many helpful advices on paper writings and technical discussions. It has been an impressive experience working with him.

I wish to thank all the other members of my dissertation committee. Professor R. Sekar and Professor Erez Zadok provided me valuable comments on my dissertation proposal to improve my work. Professor Scott Stoller and Professor Xin Wang make a great effort to attend my dissertation defense. Without generous helps from these committee members, I would not be able to schedule my dissertation defense on time.

I would also like to thank all the past and present colleagues in the Experimental Computer Systems Lab and Rether Networks Inc. Especially, I want to thank Dr. Lap-chung Lam for his insightful advices and helps on my dissertation projects, and Sheng-I Doong, President of Rether Networks, for her kind support for all these years. I also want to give my thanks to Susanta Nanda, Fanglu Guo, Chao-Chi Chang, Hariharan Kolam_govindarajan, Subhadeep Sinha, Vishnu Navda, Pi-Yuan Cheng, Wei Li, Jiawu Chen and other colleagues for their great helps on my research work and their friendship.

I want to give my sincere gratitude to my whole family. My grandmother had given me endless love and care when she was alive, and my parents always provide me with best support and encouragement in every aspect of my life. Without their support and care, I would never be able to grow up and start my PhD study.

Finally, I would give my special love and thanks to my wife Huiling, who has been accompanying me in this important stage of my life towards a PhD, and to my lovely 10-month-old son Hanxiang, who makes our home full of happiness.

Publications

- **Yang Yu**, Hariharan Kolam_govindarajan, Lap-Chung Lam and Tzi-cker Chiueh, "Applications of a Feather-weight Virtual Machine", to appear in Proceedings of the 2008 International Conference on Virtual Execution Environments (VEE'08), March 2008

- Lap-Chung Lam, **Yang Yu** and Tzi-cker Chiueh, "Secure Mobile Code Execution Service", in Proceedings of the 20th USENIX Large Installation System Administration Conference (LISA'06), December 2006

- **Yang Yu**, Fanglu Guo and Tzi-cker Chiueh, "A Virtual Environment for Safe Vulnerability Assessment (VA)", in Information Assurance Technology Analysis Center (IATAC) IAnewsletter, Volume 9, Number 3, Fall 2006

- **Yang Yu**, Fanglu Guo, Susanta Nanda, Lap-chung Lam and Tzi-cker Chiueh, "A Feather-weight Virtual Machine for Windows Applications", in Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06), June 2006

- Fanglu Guo, **Yang Yu** and Tzi-cker Chiueh, "Automated and Safe Vulnerability Assessment", in Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005), December 2005 (**Outstanding Paper Award**)

- **Yang Yu** and Tzi-cker Chiueh, "Display-Only File Server: A Solution against Information Theft Due to Insider Attack", in Proceedings of the 4th ACM Workshop on Digital Rights Management (DRM'04), October 2004

Chapter 1

Introduction

1.1 Overview of Virtual Machine

Virtual Machine (VM) is a software implementation of a real machine or execution environment that enables multiple virtual environments on a single physical machine. Each of the VMs is isolated from one another and the underlying physical machine, and gives users the illusion of accessing a real machine directly. Virtual machines have been widely used in the following applications:

- Server consolidation. Consolidate multiple under-utilized server machines into fewer physical machines by installing servers in VMs. This deployment scheme can greatly save the hardware expense and maintenance cost.
- Intrusion and fault tolerance. Isolate the execution of untrusted programs inside VMs to protect the integrity of physical machine resources. This is particularly useful in creating a secure computing environment for end users or a *honeypot* environment to catch malware or attacks.
- System migration. As a software implementation of a real machine, a VM can be migrated easily from one physical machine to another, thus reducing the down time due to hardware failure.
- Virtual appliance. Package the applications, and possibly its operating environment together and allow mobile users to carry their computing environment with portable storage device.
- Debugging and testing. Set up a test-bed with multiple VMs on the developer's physical machine to speed up the debugging and testing.

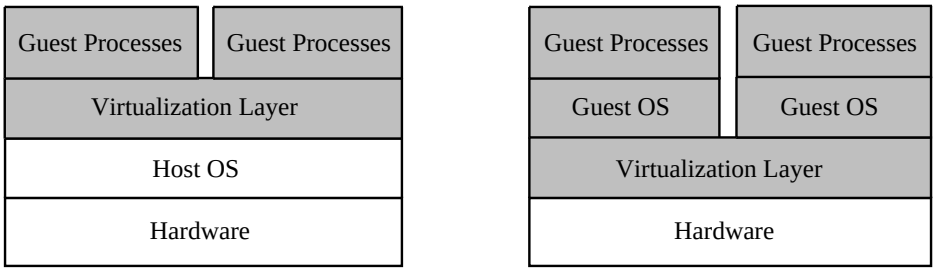

Figure 1.1: The OS-level virtualization vs the hardware-level virtualization.

To support virtual environments with software approaches, a *virtualization layer* must be placed at certain levels along the machine stack. This virtualization layer thus partitions physical machine resources and maps the virtual requests from a VM to physical requests. The virtualization can take place at several different levels of abstractions, including the *Instruction Set Architecture* (ISA), *Hardware Abstraction Layer* (HAL), *operating system level* and *application level*. ISA-level virtualization emulates the entire instruction set architecture of a VM in software. HAL-level virtualization exploits the similarity between the architectures of the guest and host machine, and directly execute certain instructions on the native CPU without emulation. OS-level virtualization partitions the host OS by redirecting I/O requests, system calls or library function calls. Application-level virtualization is typically an implementation of a virtual environment that interprets binaries. Different levels of virtualization can differ in isolation strength, resource requirement, performance overhead, scalability and flexibility. In general, when the virtualization layer is closer to the hardware, the created VMs are better isolated from one another and better separated from the host machine, but with more resource requirement and less flexibility. As the virtualization layer is moved up along the machine stack, the performance and scalability of created VMs can be improved.

1.2 OS-level Virtualization

OS-level virtualization partitions the physical machine's resources at the operating system level. This means all the OS-level VMs share a single operating system kernel. Figure 1.1 compares the virtualization layers of OS-level virtualization and hardware-level virtualization.

1.2.1 Motivation

The *isolation* among multiple VMs and between a VM and the host environment make virtual machines an effective platform to support fault-tolerant and intrusion tolerant applications. When applying virtual machine technology to such applications, a common requirement is to execute a potentially malicious transaction in a specially created VM, which is an instance of the host operating environment. One can satisfy this requirement by creating a traditional hardware-level VM [1, 2, 3], and duplicating the host environment to the new VM. However, the physical-to-virtual conversion is not efficient and takes nontrivial time to complete. The reason is that hardware-level VMs are fully isolated and each of them runs a separate operating system kernel. Initializing such a VM incurs large overhead in terms of both resource requirement and startup delays.

In contrast, all the OS-level VMs on the same physical machine share a single operating system kernel. Additionally, its virtualization layer can be designed in a way that allows processes in VMs to access as many resources of the host machine as possible, but never to modify them. In other words, a VM can share the execution environment of the host machine, but any state changes in the VM are confined in the VM's local environment. Therefore, such a VM has minimal startup/shutdown cost, low resource requirement, and high scalability. Moreover, for an OS-level VM, it is possible for the VM and the host environment to synchronize state changes when necessary. For example, the legitimate state changes in a VM can be easily committed to the host environment, while patches or new configurations of the host environment can be visible immediately to all the VMs. Essentially, the OS-level virtualization allows users to quickly spawn or destroy new isolated instances of the current operating environment on inexpensive commodity hardware. Therefore, it is particular useful to fault-tolerant and intrusion-tolerant applications that require frequent invocation and termination of disposable VMs.

1.2.2 Virtualization Approach

The basic idea to support OS-level virtualization is to redirect and confine the access requests from a VM to its local resource partition on the physical machine. For example, if a process in one VM (say vm1) tries to access a file named "/a/b", the virtualization layer can redirect the access to a file named "/vm1/a/b". When a process in another VM (say vm2) accesses "/a/b", it can be redirected by the virtualization layer to access a different file "/vm2/a/b", which is different from the file "/a/b" in vm1. Such a virtual-to-physical mapping is transparently performed inside the virtualization layer, which can be at the system call interface, system library interface, or filter driver interface. The process accessing the file "/a/b"

cannot observe that it is in fact accessing a different file. This approach is similar to the *chroot* mechanism on UNIX-like systems.

However, duplicating common resources to each VM's partition obviously incurs significant resource cost on the physical machine, and also increases a VM's initialization and termination overhead. Therefore, OS-level VMs typically share most resources with the host environment, and create private resource copies in a VM when they are to be modified by the VM. The access redirection logic in the virtualization layer also follows such a *copy-on-write* scheme to redirect a process to access the correct resource, which can be inside a VM or on the host environment.

1.3 Dissertation Overview

A main technical challenge with OS-level virtualization is how to achieve strong isolation among VMs that share a common base OS. In this dissertation, we study major OS components of Windows NT kernel, and present a *Feather-weight Virtual Machine* (FVM), an OS-level virtualization implementation on Windows platform. The key idea behind FVM is access redirection and copy-on-write, which allow each VM to read from the base environment but write into the VM's private workspace. As a result, multiple VMs can physically share resources of the host environment but can never change resources visible to other VMs. In addition to this basic redirection mechanism, we identify various communication interfaces, especially the Windows inter-process communication interfaces, and confine them in the scope of each individual VM. This allows the FVM layer to achieve strong isolation among different VMs. We demonstrate how to accomplish these tasks to isolate different VMs, and evaluate FVM's performance overhead and scalability.

The resource sharing design greatly reduces a VM's startup and shutdown cost, and thus provides a flexible platform for fault-tolerant and intrusion-tolerant applications that require frequent invocation and termination of disposable VMs. Moreover, each new VM environment can be simply a clone of the current host environment, and there is no need to change the existing computing infrastructure. The current hardware-level virtual machines do not have such flexibilities due to the *full* isolation they have achieved.

Most complete OS-level virtualization implementations are for UNIX-based OS, such as FreeBSD Jails [4], Linux VServer [5] and Solaris Containers [6]. However, most end users machines use Windows OS as the operating platform. There have been increasing demands of virtualizing the Windows environment for various applications. Some Windows-based virtualization implementation, such as

PDS [7], Softricity [8] and Virtuozzo [9] have made successful progress in this direction. However, their virtualization environments are mainly used to support certain specific applications, or are not as complete as FVM in terms of the isolation strength. We hope our work on FVM can provide researchers and developers a clear picture of how a comprehensive OS-level virtualization can be accomplished on the Windows platform, and can promote more novel applications on this virtualization framework.

This dissertation also presents five complete applications on the basic FVM framework: *secure mobile code execution service*, *vulnerability assessment support engine*, *scalable web site testing*, *shared binary service for application deployment* and *distributed Display-Only File Server*. The former three applications require isolation of the execution of untrusted code from the host environment, and the latter two applications require separation of application program, configuration or I/O data from the host environment.

To prevent malicious mobile code from compromising an end user machine's integrity [10], we confine the execution of untrusted content inside a VM by running web browsers, email clients and downloaded programs in the VM. We design a system call logging and analysis module that can log selected system calls and extract high-level behaviors from the system call logs. Therefore, we can recognize the exact scope of contamination in a VM, and allow state changes that are absolutely legitimate to be committed to the host environment.

To isolate undesirable side effects on production-mode network services when these services are scanned by vulnerability scanners [11], we clone the service to be scanned into a VM, and invoke vulnerability scanners on the virtualized service. The VM environment is identical to the host environment so the same set of vulnerabilities can be exploited. As a result, even if a vulnerability scanner is intrusive to the target service, the original production-mode network services are not affected.

To identify malicious web sites that exploit browser vulnerabilities, we need a virtual machine technology to isolate the potential malignant side effects of browser attacks from the underlying host environment. FVM's small VM startup and shutdown overhead are perfect match for this application. We use a web crawler to access untrusted sites, render their pages with browsers running in VMs, and monitor their execution behaviors. The experiment result shows that FVM-based scanning significantly improve the throughput of web site testing systems.

To support shared binary service, which requires an end user machine to fetch all its application binaries from a central server, we need to provide applications installed on a shared binary server but executed on a client machine rather than a server execution environment. All the application's registry entries, configuration files, DLLs and COM objects physically reside on the binary server. FVM's OS-level virtualization technique can be easily tailored to this application, and makes it

possible to reap the performance benefit of client-server computing and the centralized management benefit of thin-client computing.

Display-Only File Server (DOFS) [12] is an information protection technology that guarantees bits of confidential documents never leave a protected file server after being checked in. Because DOFS is based on the thin client computing architecture, it faces both application compatibility and scalability problems. Distributed DOFS, which attempts to extend the control of the central DOFS server to individual clients, leverages FVM to run a special VM in each client machine that is considered as a logical extension of the central DOFS server. As a result, distributed DOFS successfully eliminates the compatibility/scalability problems of centralized DOFS while retaining its effectiveness of protecting confidential documents against information theft.

In this dissertation, we show how to customize the generic FVM framework to accommodate the needs of these applications, and present experimental results that demonstrate their performance and effectiveness.

1.4 Contributions

In summary, this dissertation and the research work it is based on make the following contributions:

- We built a comprehensive OS-level virtualization framework on the Windows NT kernel. Each VM on this framework has minimal resource usage (zero for a new-created VM) and negligible startup/shutdown latency (less than 2 seconds on average) in comparison with most hardware-level VMs. Compared with most existing OS-level virtualization work on the Windows platform, FVM presents a more complete application virtualization framework that allows clones from a base Windows environment, confinement of IPCs through Windows GUI components, fine-grained resource control, and process behavior logging/analysis.

- We resolved the safety problem of running vulnerability assessment scanners on production-mode network services. The problem is that every existing network-based vulnerability scanner has the risk of compromising the systems and services being scanned. The approach we use is to scan cloned network services in FVM.

- We improved the scanning throughput of a web sites testing system by using FVM to host and monitor web browsers visiting untrusted web sites. Under one common testing condition, the FVM-based testing system achieves a

throughput of 50,000 to 70,000 URLs per day per machine, while a testing system based on hardware-level VMs typically scans less than 10,000 URLs per day per machine.

- We proposed a shared binary service for application deployment on Windows desktop environment. Binary servers have been widely used on UNIX-like systems, but not on Windows systems before due to complicated dependency requirements of Windows applications.

- We distributed the server-based computing architecture of an existing information theft prevention solution (DOFS) by confining information accesses inside a client VM.

- We plan to open-source the FVM prototype, which currently works on Windows 2000 and Windows XP. We believe the source of FVM will benefit the research community as well as many industry partners on OS-level virtualization. In addition, the API and system call interception mechanisms used by FVM can be easily extended to support other applications that aim at customizing close-source Windows program behaviors.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 summarizes related work on hardware-level virtual machine, OS-level virtualization, and different levels of Windows interception technologies. Chapter 3 presents the design and implementation of the FVM virtualization layer. We describe the general rule of namespace virtualization, copy-on-write and communication confinement in FVM, and discuss the implementation of each virtualization component. Chapter 4 evaluates the FVM's isolation strength, performance overhead and scalability. In Chapter 5, we present five complete applications on the FVM framework: secure mobile code execution service, vulnerability assessment support engine, scalable web site testing, shared binary service for application deployment and distributed *Display-Only File Server*. Finally we discuss FVM's future work and conclude the dissertation in Chapter 6. In the appendix of the dissertation is an introduction of FVM's source code organization.

Chapter 2

Related Work

2.1 Hardware Level Virtualization

Hardware-level virtualization technologies simulate the complete or a subset of hardware to run unmodified guest OS, or modify the guest OS to utilize the underlying physical hardware. Examples of the hardware-level virtualization technologies are Bochs, VMware, Virtual PC, Xen, and etc.

2.1.1 Full System Simulation

Full system simulations emulate the entire instruction set architecture in software, so they can support an unmodified guest OS for the same or different architecture. The emulator executes instructions from a virtual machine by translating them to a set of native instructions and then executing them on the native hardware. Bochs [1] is x86 emulator that runs on many platforms and supports different guest OS, including DOS, Windows and Linux. It emulates the Intel x86 CPU, common I/O devices and a custom BIOS. Another example of full system simulation is the Mac version of Virtual PC [13]. Because every instruction of the guest system needs to be interpreted by the emulator software, full system simulations usually have significant performance overhead.

2.1.2 Native Virtualization

Native virtualization simulates a complete hardware environment to support an unmodified guest OS for the same type of CPU. It intercepts and simulates privileged instructions, and allows certain un-privileged instructions to be executed directly on the physical hardware. This reduces the overhead of emulating certain instructions and thus improves the VM's performance. VMware [2, 14] and Microsoft

Virtual PC [3] are examples of native virtualization technologies. They virtualize common PC hardware such as processor, memory and peripheral I/O devices and support different type of guest OS on a single physical x86 machine. Native virtualization have used the virtualization extensions in recent x86 processors [15, 16] to improve their performance.

2.1.3 Paravirtualization

The *paravirtualization* technologies provide special interfaces, called *hypercall*, for the guest OS to achieve high performance that is close to the performance of native executions. However, to support paravirtualization, the guest operating systems are usually required to be modified. Examples of paravirtualization technologies are Xen [17], Denali [18] and User-Mode Linux (UML) [19]. Xen is a virtual machine monitor for multiple architectures, and allows both Windows and Linux guest OS to run with the Linux-based host OS (*Domain 0*). With the assistance of hardware virtualization technologies, it is possible that the guest OS does not need to be modified to run within Xen virtual machines.

Full system simulation, native virtualization and paravirtualization all place the virtualization layer at a level close to the hardware. The advantage of these technologies is the full isolation between different VMs and the host machine. However, some of these technologies are involved with large resource requirement, high performance overhead and poor scalability. Although paravirtualization and hardware-supported virtualization can improve the runtime performance to almost native speed and can achieve large scalability, they are still not as flexible as operating-system level VMs when used for applications that require frequent invocation and termination of disposable VMs. In addition, deploying these hardware-level virtualizations requires end users to change their existing computing infrastructure, such as installing the *Virtual Machine Monitor* (VMM) and performing physical-to-virtual conversions. Finally, in some *stand-alone* VMM implementation such as VMware ESX server, the hardware compatibility is always a concern because the VMM sits on the bare hardware and thus needs to provide device drivers for hardware access.

2.2 OS-level Virtualization

We present related works on server virtualization, application virtualization, versioning file system and compatibility layers in the following. All of these technologies create virtual views or environments on top of a single OS kernel. The OS-level virtualization layer can be at system library APIs, system call interface or

file system driver stack. Please note that the difference between server virtualization and application virtualization is, the former usually creates complete virtual operating environments (including private user accounts and management utilities), that can support server consolidations, while the latter is usually a subset or a particular application of a complete virtualization, such as sandboxing untrusted application programs.

2.2.1 Server Virtualization

To partition a physical machine environment into multiple OS-level VMs, the first step is to create a separate file system name space for each VM. The Unix *chroot()* system call sets the root directory of the calling process and its child processes. This set of processes therefore establish their own file system name space and cannot look up files outside of the new root. Many Unix-like OS-level virtualization systems base on this system call to virtualize their file system name space. The FreeBSD Jail [4] facility creates OS-level VMs called *jails*, each of which has its own file system root, processes, IP address and user accounts. Processes in a jail cannot access directories outside of that jail. This is implemented by leveraging *chroot()* and preventing ways to escape from *chroot()*. Activities that cannot be contained inside a jail, such as modifying kernel by direct access or loading kernel modules, are prohibited for all jailed processes. Moreover, several types of interactions between processes inside and outside of the jail, such as delivering signals, are prevented. With these isolations and protections, Jail can be used in the web service provider environment, to protect the provider's own data and services from their customers, as well as protecting data of one customer from others, by hosting each customer in a separate jail. However, virtualization of file system name space is not enough to create isolated OS-level VMs for a Windows desktop environment, where many other types of name spaces than files, such as registry and kernel object exist. In terms of storage utilization, creating a jail requires system libraries and data files to be duplicated under that jail's root directory. In contrast, FVM virtualizes major types of non-file name spaces in a Windows environment, and implements a special copy-on-write mechanism to share system resources as much as possible, therefore minimizing the resource requirements when multiple VMs coexist.

The Linux-VServer [5] is a more advanced Jail-like implementation on Linux. It partitions a Linux machine into multiple *security contexts*, each of which supports an OS-level VM, called a *Virtual Private Server(VPS)*. Besides isolating the file system name space, the Linux-VServer isolates shared memory and other IPCs among different VPSs. By implementing context capabilities, context accounting

and context resource limits, it provides more control to processes in a VPS. As a result, processes in a VPS may not perform a denial-of-service attack on other VPSs on the same physical machine. An extension to the original Linux-VServer project, called FreeVPS, supports similar isolation functionalities. To reduce the overall storage usage, the Linux-VServer does not duplicate common files, for example, shared libraries, in each of the VPSs. Instead, each VPS can create hard links to common files. To prevent modifications to these files by a VPS from affecting other VPSs, but still to allow file updates in each VPS, these shared files are given special attributes to make them immutable but unlinkable. Therefore, a process in a VPS cannot modify the file data but can still remove the link and create a new version. However, the Linux-VServer must identify the common files and prepare them with the special attributes in a process called *unification*, to initialize the file system name space of every VPS. In comparison, the FVM implementation does not require such a process because each VM is created with the identical environment as the base environment of the host, and its private states are constructed dynamically with the copy-on-write scheme.

Solaris 10 from Sun Microsystems allows multiple OS-level VMs, called *zones*, to be created on top of one Solaris OS kernel [6]. Every zone can share portions of the file system data with other zones and the host environment (called *global zone*) to reduce the amount of disk space. This file system model in a VPS is called *sparse-root*. Unlike the Linux-VServer, file sharing in a sparse-root zone is implemented by using the read-only loopback file system (lofs) to mount the shared directories from the host environment. Since most kernel and core system components are shared, patching to these files from the host affects all the zones automatically. However, processes writing to some shared directories, such as an installer process, may not proceed in a sparse-root zone. To resolve such problems, a zone may be configured to duplicate a portion of or even entire file system data with private disk space. Resource usage in a zone can be flexibly controlled using mechanisms like fair-share scheduling, which ensures each zone to get an equal share of CPU usage. With the resource management facility, a zone is also referred to as a Solaris *container*. FVM uses the Windows job object to implement similar resource control mechanisms for each of its VMs, and its copy-on-write scheme further minimizes the time taken to create every new VM.

OpenVZ [20] is another OS-level virtualization project for Linux and is the basis of the Virtuozzo [9] server virtualization product from SWsoft. It not only provides comprehensive isolations and resource management, but also allows checkpointing processes in a VPS and live migration to a different physical server without noticeable delays or downtime. Virtuozzo even supports Windows VPSs on the Microsoft Windows 2003 OS and is therefore close to the FVM implementation. The

main difference between Virtuozzo for Windows and FVM is the resource sharing mechanism. For file system sharing, a Virtuozzo VPS is created with a tree of copy-on-write symbolic links pointing to the real files on the host. This mechanism is slightly different from FVM because the directory hierarchy still needs to be duplicated and links with the same file name be created in a VPS, and is similar to the unification process of the Linux-VServer. With respect to Windows registries, Virtuozzo statically duplicates all the registry data used by the OS and the applications into the VPS without sharing, but FVM only copies a portion of registry entries from the host registry into a VM at the time when processes in the VM intends to update them.

2.2.2 Application Virtualization

Application virtualization creates isolated virtual environments on a host OS to sandbox the deployment and execution of selected application programs. Such virtual environments are mainly used for four purposes: resolving application conflicts, migrating running processes, isolating untrusted programs, and supporting portable application environments.

To resolve conflicts between applications, an application is not physically installed on the desktop. Instead, its runtime environment is packaged into a OS-level VM, which is then shipped from a central server to the desktop as data on demand. Maintenance and updates of the application are performed on the central server, turning software as a service. The execution of the packaged application on the client is confined within the VM via redirection techniques. Thinstall [21] compresses all the files and registry settings of an application into a single executable file by logging the application's installation on a clean machine. The generated executable is self-contained, responsible for decompressing the entire runtime environment, loading proper libraries into the memory and redirecting file or registry requests. However, the size of the packaged executable can be rather large, depending on the complexity of the containing applications. Similary, Altiris [22] places application installations or data into managed virtual software packages, which can be delivered to end user machines in various ways. To reduce application startup overhead, many other similar techniques, such as the Progressive Deployment System [7], Microsoft SoftGrid [8], AppStream [23] and Citrix Streaming Server [24], organize an application' runtime environment into separate components, which are shipped gradually to the client on demand of the actual execution flow. This improvement allows almost immediate startup of a packaged application that resides on a central server, and is referred to as *application streaming*. The redirections and isolations on the client environment are achieved by implementing a program

loader at the user level [21], intercepting Win32 APIs [7] or intercepting kernel system services [23]. However, the isolations are mostly targeted at files and registries. IPCs are generally not confined within a VM, as what FVM has implemented. This is because application streaming technique is a new software deployment scheme rather than a desktop security solution. The same application encapsulation idea is also implemented by Trigence AE [25] and Meiosys [26] on Linux/Solaris.

To effectively support process migrations from one machine to another, a virtualization layer must be introduced to preserve the consistency of resource naming scheme before and after the migration. Zap [27] uses private namespaces to provide processes to be migrated a host-independent virtualized view of the OS on Linux. It virtualizes process IDs and IPCs by system call interpositions, and virtualizes file systems by *chroot* system call. Common files are shared and mounted from an NFS server. Zap also develops techniques similar to NAT to maintain open connections of networked applications before and after the migration. [28] uses system call interpositions and file system stacking to further restrict virtual resources a Zap process can access if it is untrusted. MobiDesk [29] extends the Zap idea to the server-based computing environment and implements a Linux terminal server where each computing session is separated from other sessions and can be migrated to other servers to ensure high availability. Since these technologies are mainly developed to support process migrations, they do not perform file copy-on-write to protect the host system's integrity. The current FVM virtualization does not supports process migration, but we can allow simple cold migration of an FVM state by moving persistent state changes and restarting FVM processes on a new host. Process migrations on the Windows platform have been discussed in the papers of checkpoint facility on NT [30] and *vOS* [31]. In general, process states in user space, such as thread context, heap and static data, are easy to capture at the checkpoint time. However, to restore system states associated with the checkpointed process on Windows, we need to record system calls before the checkpoint time and then re-invoke the system calls at the restore time.

Alcatraz [32] implements a one-way isolation idea for safe execution of untrusted programs on Linux. Untrusted processes are confined in an isolated environment with read-access to the host environment, and their write attempts are redirected into the isolated environment. Alcatraz provides isolated executions by system call interposition to redirect file modifications made by untrusted processes to a dedicated area of the host file system, called *modification cache*. Read operations from these processes are also modified to incorporate updates in the modification cache. A summary of the file modifications are presented to the user on the termination of a process, and users can decide whether to commit or discard. One problem with it is its high performance overhead due to its system call interception mechanism, which is completely at the user level with a monitoring process.

Safe Execution Environment (SEE) [33] extends Alcatraz by moving the redirection logic into OS kernel at the Linux VFS layer. Besides copy-on-write to regular files, it also supports copy-on-write on directories using a *shallow copy* operation, which copies a directory but leaves its directory entries pointing to the original files on the host. These improvements make its performance overhead to be typically less than 10%. Moreover, SEE restricts access to device special files, network access, and IPCs, and designs a systematic approach to resolve conflicts on committing file modifications. Finally, SUEZ [34] extends SEE on a single host to multiple networked hosts to isolate any effects of client-server applications. All these projects are very similar to our FVM approach with respect to the motivation and application scenarios. However, FVM must take care of various types of name spaces and communications on the close-source Windows OS. This makes the FVM implementation more challenging. In addition, FVM provides virtual machines management with VM states, resource restrictions and VM operations, and its commit operation is not focusing on detecting conflict but on automatically detecting malicious behaviors.

GreenBorder [35] creates a special OS-level VM on Windows called a *Desktop DMZ* in which internet content is executed but isolated from the host environment. Web browsers and email client programs run inside the virtual environment. Malware coming from the web or email attachments only affects the virtualized system state, and can be easily eliminated as the VM is reset. It also creates another type of VM called *Privacy Zone*, where sensitive online transactions can be performed but isolated from other untrusted environments. A similar idea is implemented in the security wrappers for Windows NT/2000 [36], which virtualizes operations of MS Office, Outlook and Internet Explorer that violate security rules. This virtualization ensures processes run under their prescribed security policy without the user's intervention. Windows Vista also provides a built-in file/registry virtualization feature [37]. This feature helps to resolve the potential conflict between Vista's user account control mechanism and legacy applications that require an administrator's privilege. When such an application try to update protected directory, the updates will be redirected to a virtualized directory. These systems and features minimize the potential damages caused by malware execution without interfering with legitimate application behaviors, and therefore are more user-friendly than behavior blocking approaches. Compared with these systems, FVM is a more general OS-level virtualization technique that can execute arbitrary Windows applications within multiple isolated Windows environments, without conflicts among application instances.

MojoPac [38] and Ceedo [39] allows users to carry virtual application environments in portable storage devices, such as an iPod or a USB drive. Different from Moka5 [40] or VMware ACE [41], their virtualization layers are at the user level

and do not require a VMM to be installed on the connected host machine. They provide their own shell, file manager and window desktop with similar look-and-feel to a standard Windows OS. Application files and configurations are installed within the portable devices. When users plug the portable device into a PC running Windows OS, they can launch any applications carried by the device, or launch applications on the connected machine with customized settings carried by the device. The executions of the carried application are confined in the virtual file system and registry inside the device. The overall architecture is similar to Thinstall and SoftGrid, except that applications are pre-loaded into the portable device instead of being downloaded from a streaming server. The U3 smart drive [42] and the U3 specification enables a similar virtual application environment on a USB flash drive, but the USB drive is specially formatted and applications may need to be modified to be fully U3 compliant.

2.2.3 File System Versioning

File system versioning [43, 44] allows multiple versions of a file to coexist and thus provides the flexibility to undo changes to the file system. Intrusion recovery systems [45, 46] use extensive file system logging and versioning to automatically repair a compromised network file server. Transactional file system [47] groups file updates into a transaction and hides all the updates from applications outside the transaction until they are committed. Many of the versioning file systems use block-based versioning rather than file-based versioning to avoid duplication of common file blocks. The FVM virtualization is similar to versioning of various system resources plus visibility control. Each VM starts with a base version of files and registries stored on the host and gets a separate new version at the first write attempt. In particular, to generate a new version of file into a VM, the FVM's file virtualization module performs copy-on-write on the entire file instead of affected file blocks. This approach introduces some additional overhead on both disk space and execution time, but simplifies the implementations in handling multiplexing logic and memory-mapped file I/O.

2.2.4 Compatibility Layer

Finally, many techniques try to create a compatibility layer at the OS-level to emulate the execution of binaries for other OSes. Examples of such projects include Wine [48] and its descendants, which emulates Windows applications on a Unix-like OS, and Interix (Microsoft Windows Services for UNIX) [49], which emulates a Unix environment subsystem on a Windows NT kernel. These techniques implements all the system libraries that translates system calls of the emulated application

into system calls supported by the host OS, and maps name spaces of the emulated application to completely different name spaces of host OS.

2.3 Interception Technique on Windows

To insert additional code in order to modify the program logic, we can either modify and recompile the program's source code, or intercept the program's execution at certain OS interfaces. When an application's source code is not available, especially on the Windows environment, interception is main technique to change the application's default behavior. Interceptions can be at the user level or the kernel level. We discuss common interception techniques on Windows OS in the following.

2.3.1 User-level Interception

The Windows OS provides a set of standard APIs, based on which user processes request OS services. These APIs are implemented in system libraries such as *kernel32.dll* and *user32.dll*, and are the most common interception points at the user level. To intercept a target API function, we can either modify the API function's address saved within some tables of the calling process, or rewrite the API function's instruction bytes in the target DLL in memory.

A Win32 executable uses *Portable Executable (PE)* file format to organize its code sections, data sections and other control data structures. In memory, its image content follows the same PE format. The process usually relies on a special data structure called *Import Address Table (IAT)* to make an API call in a DLL. The IAT is a table of function addresses, which are filled by the OS loader when the corresponding DLL is loaded. With the filled IAT, calling an API function is an indirect call to the function address saved in an IAT entry. Therefore, intercepting an API function can be achieved by locating the IAT in the process address space and modifying the function address in the IAT to our own function(how and when to modify the IAT will be explained later).

Although IAT modification can intercept API functions in most cases, it cannot intercept functions if they are not called via an IAT entry. For example, an application program can specify the delay-load option at compile time so the target DLL is not loaded until a function in the DLL is called. Another example is that a process can explicitly load a DLL and obtain the address of an API function in that DLL. In both examples, the process uses *LoadLibrary()* and *GetProcAddress()* API without accessing any IAT entry. Moreover, it is also possible for an unfriendly

process to use some hard-coded addresses to call certain API functions directly, because Windows loads some system DLLs at the same base address per Windows version. Therefore, a more reliable interception approach is to directly rewrite the API function's instruction bytes in the target DLL in memory.

The Detours [50] library replaces the first few instructions of the target API function with an unconditional jump to the user-provided function, called a *detour function*. The replaced instructions from the target function are saved in a *trampoline function*, with an unconditional jump to the rest code of the target function. Therefore, a process's call to a target API is routed to the detour function, which can call its trampoline function when it requests service from the original target API function. This code rewriting approach is more effective than the IAT modification approach because the target function is always intercepted no matter how the process makes the function call. However, it also has a drawback: the contents of the physical page to be modified in the target DLL are copied to a new physical page due to copy-on-write protections, thus processes sharing the same target DLLs consume more physical memory, especially when many functions in different DLLs are to be intercepted. Rewriting the DLL file on disk statically avoids this problem, but it compromises the integrity of OS system files and is often prohibited by the OS. For example, the Windows File Protection (WFP) prevents user processes from overwriting Windows system files.

In order to modify the IAT or rewrite the target function's instructions, a user-provided DLL is injected into the target process's address space. When this DLL is being loaded, its initialization routine is invoked to perform the IAT modification or function rewriting. This DLL provides the intercepting function that has the same function signature as the target function and is called when the target function is called. There are several ways to inject a user-provided DLL into the address space of another process, as follows:

- Add the DLL name (e.g. hook.dll) into the Windows *AppInit_DLLs* registry key. The hook.dll will be loaded automatically into all the processes that are loading user32.dll.

- Use the *SetWindowsHookEx* API to register a message handler for windows in other processes. This message handler is provided by a DLL (e.g. hook.dll), which will be loaded automatically into all the processes with access to that type of window messages.

- Create a remote thread running in the address space of specified target process. The thread simply invokes the *LoadLibrary* API to load a DLL (e.g. hook.dll), whose name is written to the target process memory in advance.

- Create a process in the suspended state and then modify its IAT to include the DLL name before resuming the process's execution.

- Create a process in the suspended state, and then replace the instructions at the program entry point with the instructions of *LoadLibrary*. Once the process resumes execution, the specified DLL is loaded and the original instructions at the program entry point are restored.

Another user-land interception technique is to statically modify the *Export Address Table (EAT)* of the target DLL file. The EAT of a DLL contains the addresses of its exported API functions. When a process is loading a DLL, the OS loader fills the corresponding IAT entry of the process according to the DLL's EAT. All the addresses saved in the EAT are relative to the DLL's image base. Therefore, we can modify the DLL file by appending an additional code section containing the intercepting function, and replacing the relative address of the target function in the EAT with the address of the intercepting function. In this way, the OS loader fills the corresponding IAT entry with the address of the intercepting function. The problem with this approach is still the integrity issue of system files, and the compatibility issue across all Windows versions.

DLL forwarding technique renames a target DLL to be intercepted, and constructs a proxy DLL with the same name and the same set of exported functions as the target DLL. Only interesting API functions to be intercepted are handled by the proxy DLL code. All the other exported functions in the proxy DLL simply use an unconditional jump to invoke the corresponding functions in the target DLL. Because system DLLs usually export a large number of functions, it is time-consuming to manually input the source code for the proxy DLL. Therefore, an automatic tool should be used to traverse the target DLL's EAT to generate a template of the source code that compiles to the proxy DLL.

A *Layered Service Provider (LSP)* [51] is a DLL that uses Windows socket APIs to insert itself into the TCP/IP stack. It intercepts all the Winsock commands before they are processed by the underlying Winsock library, such as binding to certain network address. LSP is a documented interception technique that has been used for content filtering, QoS and other applications.

User-level interceptions can be at higher level than the system libraries. For example, window subclassing is used to intercept the message handling of specified application window, and application-specific plug-ins are used to customize application behavior, such as extension handlers for Windows Explorer, Browser Helper Objects for Internet Explorer, etc. These techniques are out of the scope of this report.

Part of the FVM virtualization layer is at the user level through IAT modifications that provides virtualizations to window message, window visibility and daemon processes, and a LSP DLL that virtualizes the network interface. Details about the user-level virtualization layer is presented in Chapter 3.

2.3.2 Kernel-level Interception

A process makes system calls to request OS services, and the system calls may deliver the user requests to certain device drivers, which process the requests and return the result to the caller. Therefore, we can intercept system calls or driver functions to change the system behavior. Compared with user-land interceptions, interception at kernel level is more reliable because it cannot be bypassed by user application processes.

To make a system call on Windows NT kernel, a process loads the *EAX* register with the ID of the system call and executes *INT 2E* or *SysEnter* instruction. This causes the calling thread to transit to kernel mode and to execute the system call dispatch routine. The despatch routine locates the address of the system call stored in the Windows system call table, called *System Service Dispatch Table (SSDT)*, and then starts execution of the system call. The SSDT stores the addresses of all the Windows system calls, indexed by the system call ID. To intercept a system call, we replace the system call's SSDT entry with the address of our own function, which can call the original system call function plus pre and post-processing. There are three issues with modification of the SSDT. First, the SSDT resides in kernel memory, so a kernel driver is usually loaded in order to modify the SSDT. Second, the SSDT is write-protected in Windows XP and later version of Windows, so methods to turn off the page protection on the fly should be used, such as modifying the *write protect (WP)* bit in the *CR0* control register. Third, the SSDT is prevented from modification on more recent Windows kernel on x64 platform. The prevention mechanism is called PatchGuard [52], which periodically verifies the integrity of critical kernel data structures and sensitive register contents. To modify the SSDT on such kernels, PatchGuard needs to be bypassed in advance. Although system call interception is often used in security-related projects, Microsoft does not support it due to integrity concerns, especially on newer Windows OS. In practise, alternative approaches such as filter drivers for different system components are more promising in the long run.

A system call usually delivers the user request to the responsible device drivers in the format of *I/O Request Packet (IRP)*, which are processed by the IRP handler functions of each related driver. The addresses of all the handler functions provided by a driver are also stored in a function table. Therefore, we can locate the function table and replace certain function addresses in the table in the same way as the

SSDT modification. Another interception approach is based on the layered driver architecture, under which a middle layer driver can be inserted into the original driver chain to modify the behavior of its lower-level driver. This middle layer driver is called a filter driver, which observes all the I/O requests routed to the driver below it. The filter driver provides all the IRP functions supported by its underlying driver. Before an I/O request is passed to the underlying driver, the filter driver's handler routine is invoked first. It passes the modified request to the underlying driver or even processes it without passing the request down. The most frequently-used filter driver is the file system filter driver, which is used for file content encrytion/compression/scannning. Filter drivers for Windows registry is supported by Windows XP and newer version of Windows via *RegistryCallback*.

The FVM virtualization layer is mainly a kernel driver that intercepts Windows NT system calls manipulating file, Windows registry and kernel objects. In newer version of Windows OS, part of the virtualization layer can also be implemented using filter drivers.

Chapter 3

Feather-weight Virtual Machine

3.1 System Overview

An execution environment can be described as a set of user-mode processes that access the underlying hardware through multiple interfaces at different levels: system library, OS system call, device driver and I/O instruction. To support multiple execution environments, or VMs, on one physical machine without interference with each other, one of these interfaces should be modified so that the same access requests from different VMs can be mapped or converted to requests to different OS objects. The modified interface, called *virtualization layer*, determines the VM's scalability, runtime performance and degree of isolation. In general, when the virtualization layer is close to the hardware, a VM can be completely isolated from other VMs, but its scalability and performance suffer. This is because many system resources are duplicated without sharing, and because some hardware has to be emulated.

The *Feather-weight Virtual Machine (FVM)* project is designed to facilitate sharing (OS kernel, file system, hardware) in order to improve the scalability and runtime performance, while still maintaining a certain degree of isolation. The virtualization layer in FVM is at the OS's system call interface, as shown in Figure 3.1. All the VMs share the host OS's kernel-mode components, such as OS kernel and device driver. Moreover, critical system daemons and the file system image are also shared by default. Each new VM starts with exactly the same operating environment as the current host OS. Therefore, both the startup delay and the initial resource requirement for a VM are minimized. The virtualization is performed by renaming system call arguments instead of device emulation or instruction interpretation. Therefore an application's runtime performance in a VM may also be improved.

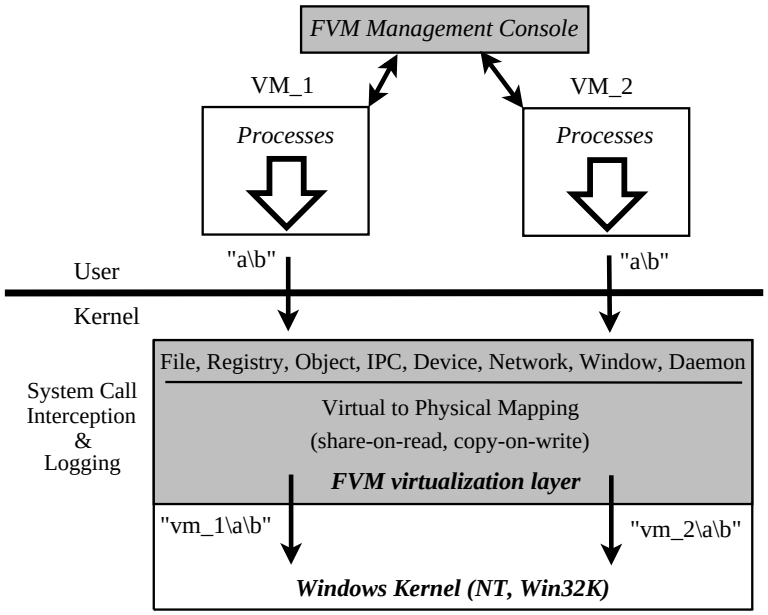

Figure 3.1: The FVM virtualization layer supports multiple VMs by namespace virtualization and copy-on-write. The FVM management console allows users to perform FVM operations on specified VM.

The FVM virtualization layer observes all the access requests from user-mode processes. As a result, it can redirect access to the same OS object from different VMs to requests against different versions of the same OS object. FVM uses namespace virtualization and resource copy-on-write to implement the access redirection and isolation between different VMs. When a new VM (say `vm1`) is created, it shares all the system resources (disk files, system configurations, etc) with the host machine. Later on, when different types of requests from a process *p* in the VM pass through the FVM layer, these requests can be redirected as follows:

- If *p* attempts to create a new file `/a/b`, the FVM layer redirects the request to create a new file `vm1/a/b`.
- If *p* attempts to open an existing file `/a/b`, the FVM layer redirects the request to open a file `vm1/a/b`. If file `vm1/a/b` exists, no further processing is made in the FVM layer; otherwise, the FVM layer checks the access flag of the open request. If the access is "open for read", the request will go to the original file `/a/b`; if it is "open for write", the FVM layer copies `/a/b` to `vm1/a/b`, and then redirects the request to open `vm1/a/b` again.
- If *p* attempts to read or write an existing file, the FVM layer passes the request

through without additional processing, because read/write request is based on a file handle, which is returned by a previous open request. If the open request has been redirected, all the subsequent read/write requests on the same file handle are redirected inherently.

- If *p* attempts to delete an existing file `/a/b`, the FVM layer only marks the file as deleted by adding its name /a/b to a per-VM log. The file `/a/b` is not deleted from the file system.

- If *p* attempts to make any types of interprocess communications, such as sending window message, to another local process, the FVM layer examines the two processes and blocks the communications unless they are running in the same VM.

- If *p* attempts to access and communicate with certain devices, the FVM layer denied the access unless it is permitted by a policy.

We only list a few access requests in the above example to explain the basic redirection mechanisms implemented in the FVM layer. There are many other access requests the FVM layer must intercept to guarantee the consistency of the application semantic in a VM. In addition, there are various types of OS objects on Windows, such as file, registries and kernel objects. A fundamental issue with FVM design is to identify all these objects that should be virtualized in order to isolate different VMs. First, file and registry represent persistent data and system settings and thus must be virtualized. Second, Windows applications can use kernel objects and window managing mechanisms to synchronize with each other. For example, many application programs (e.g. Microsoft Office) allow only one instance of itself to be started on the same machine at one time. In other words, no matter how many files the program are opening simultaneously, there is at most one process of the program on the same machine. This instance restriction can be implemented by checking the existence of certain kernel objects, which share a single namespace; or by broadcasting window message to other existing windows, which make replies to declare its existence. Therefore, to break the instance restriction and enable multiple instances of the same application program to run independently of each other in different VMs, kernel objects must be virtualized, and many Windows-specific IPCs such as window message should be confined. Third, a process can access and communicate with some devices, whose device drivers are responsible for handling requests from the process. To prevent the drivers from modifying system state outside of the VM, device access should be confined based on policies. Fourth, many network server applications (e.g. Apache) start as daemon processes, which are managed by a special system daemon called *Service Control Manager*. To enable

multiple instances of the same network server application to run in different VMs, the daemon process management mechanism should be virtualized. Finally, the network address should be virtualized as well so each server application instance can start successfully by listening on the same port but at a different IP address.

Once all these types of OS objects and IPCs are virtualized, the FVM virtualization layer is able to allow multiple isolated VMs to coexist on the same physical machine. To prevent a VM from exhausting the whole system's resource, each VM is accompanied with a set of resource restriction policies, such as disk space and CPU time. To manage different VM configurations or to migrate a VM to the same or different physical machine, a user-level *FVM management console* is provided. This management console keeps track of the FVM states, associates a process with a particular VM, and performs various FVM operations. Finally, the FVM virtualization layer must protect itself against malicious processes that attempt to break out of the VM resource boundary.

3.2 FVM States

Under the FVM architecture, the state of a VM is defined as follows:

- A virtual machine identifier,
- A set of processes running within the VM,
- A root file directory containing file updates by the VM,
- A root registry key containing registry updates by the VM,
- A root object directory containing object updates by the VM,
- A log of files and registry entries deleted/renamed by the VM,
- An IP address,
- A set of policies regarding to resource quota, device access and network access.

The virtual machine identifier is a unique number representing a VM. It is allocated when the VM is created. A VM is always started with a root process. When a descendant processes of the root process starts to run, it is associated with the same VM context.

Three types of directories are created at the VM creation time. They are the private workspace containing copy-on-write version of files, registries and objects

of the VM. These workspace directories themselves are created within the host directory namespace but only visible to the VM itself. They are named by the VM identifier, so different VMs have different workspace directories.

Deleting or renaming a file by a VM process should not affect the file on the host environment. Therefore, a per-VM log is introduced containing the names of all the deleted/renamed files as well as registries. When the VM is stopped, the log data is dumped into a file as part of the VM state.

An IP address can be assigned to a VM when the VM is created. Having a separate IP address allows multiple instances of the same network server application to coexist on the same host machine. Each of the applications runs in a different VM and binds to different IP address.

To prevent Denial-Of-Service in a VM due to the execution of unfriendly processes, a set of policies regarding to resource quota and network access can be specified when a VM is created. These policies set limits to the system resource the VM processes can consume, such as disk space, physical memory, etc. To prevent processes in a VM from accessing devices to affect the host environment, a device access policy can also be specified. A process in a VM cannot access a device unless the device is included in the device access policy.

When a VM is stopped, all of its processes are terminated and its persistent states are kept on the disk. A VM's persistent states include the VM ID, the workspace directories for files and registries, the delete log, the IP address and the VM policy. Because FVM currently does not support process checkpoint and restart, the running state of all the processes in a VM, including kernel objects opened by those processes, are not maintained when a VM is stopped.

Essentially, a VM is a virtual execution environment that tightly depends on the host system but does not alter the host system's state. The execution environment of a VM consists of the depending host environment, the FVM virtualization layer and the VM state. The file system image visible to a VM should be the union of the VM's workspace directory and the current host directory, minus the files whose name appear in the VM's delete log. The same semantic applies to a VM's registry and kernel object images. Consequently, when a process in the VM queries its current execution environment, e.g. using *Dir* command to list a directory's entries, the FVM virtualization layer is responsible for producing a complete and correct image of its environment. Another fact with the VM's execution environment is that some updates to the host environment can be immediately reflected in the VM environment. This is one advantage of OS-level virtualization: when OS and application security patches are applied to the host environment, all of the VMs on the same host can also be patched because the patched system files and libraries are usually shared by the host and the VMs.

3.3 FVM Operations

The FVM management console provides a comprehensive set of operations for users to manipulate VMs. A VM's state information is maintained inside the FVM virtualization layer. Therefore, to support operations that update a VM's state, the management console typically use *DeviceIoControl* API to send commands to and exchange data with the FVM layer. The FVM operations are defined as follows:

CreateVM creates a new VM whose initial state is identical to the host environment at the time of creation. The new VM starts a *VM shell* process, which is a file browser similar to the Windows explorer process. A unique VM identifier is allocated to the VM, and its private workspace directories for file, registry and objects are created. Users can start arbitrary processes under the VM shell by clicking a file icon or typing a command. The ID of the VM shell process is associated with the VM ID. All the descendant processes of the VM shell are associated with the same VM.

CopyVM creates a new VM whose initial state is duplicated from another VM. The copy operation copies the VM's private workspace directories and state files. If a remote physical machine has all the environment settings (files, registries, etc) of the current host environment, users can use this operation to copy a stopped VM to the remote machine and start the VM there, thus enabling cold migration of a VM.

ConfigureVM creates a new VM with an initial state that users can configure explicitly. This operation allows one to limit the visibility of a new VM to part of the host environment. This can prevent untrusted processes from accessing sensitive information on the host environment. For example, one can initiate a new VM configuration that restricts the VM's read access to a protected directory on the host.

StartVM starts a stopped VM, initializes it with the saved VM state and launches a VM shell process.

StopVM terminates all the active processes running in a VM, saves the VM's state on the disk and renders it inactive.

SuspendVM suspends threads of all the processes in a VM. In addition, for each process in the VM, it sets the working set size to zero and makes all windows of the process invisible. As a result, all the processes in the suspended VM stop utilizing CPU and physical memory, and the system resource held by the VM is minimized. However, the VM is not really suspended. Its process states are not checkpointed and cannot be live-migrated to a different environment. Process checkpoint and restart will be available in the future work. This special "suspend" implementation simply makes the VM appear to be suspended.

ResumeVM is the reverse operation of *SuspendVM*. It resumes threads of all the processes in a VM, restores the working set size of each process in the VM to

normal and renders all the process windows visible.

DeleteVM deletes a VM and its state completely. All the files and registries in its private workspace are deleted.

CommitVM merges file and registry state of a stopped VM to the host machine and then deletes the VM. It overwrites files and registries on the host environment with files and registries in the VM's private workspace, and deletes files and registries on the host environment that are in the VM's delete log. The effect of this operation is that the processes in the VM have been running on the host environment directly. This can help users to try out untrusted application programs in a VM before install them permanently on the host environment.

There are three issues with the commit operation. First, committing a VM's state cannot be completed successfully if the target file or registry on the host environment is being accessed by other processes. To resolve this access conflict, the FVM management console finds out which process in which VM is locking the file or registry, and gives users suggestions on either terminate that process or wait for some time before committing again. Second, committing a VM's state may not be completed correctly if the target file or registry on the host have been modified by a host process since the VM's private copy is created. If the target file or registry is overwritten by the VM's private copy, a host process depending on the file or registry may suffer from inconsistency. FVM should compare the creation time of the private copy with the last modified time of the host copy, and leave the commit decision to the user if the creation time is earlier than the last modified time.

The third issue with the commit operation is that committing a VM's state may not be safe, especially when untrusted or vulnerable processes have been running in the VM. Although users can manually examine the VM's private workspace or scan the VM's private workspace with anti-virus tools, it is still difficult to be sure that the private workspace contains no malware, especially zero-day malware that has not been recognized before. To facilitate the user's decision on whether to commit or not, the FVM management console monitors selected file and registry locations, such as all the *Auto-Start Extensibility Points (ASEP)* [53]. ASEP is a set of OS and application configurations that can be modified to enable auto-start of programs without explicit user invocation. It is the common targets of infection for most malware. If the FVM management console detects suspicious states in FVM's private workspace, e.g. a few updates to the ASEP, it disables the commit operation unless users explicitly request and confirm the commit. To identify a VM's process behaviors more completely in order to provide more accurate commit decisions, FVM also supports logging of security-related system calls made by processes running in a VM, and analyzing the system call logs to derive a process's high-level behaviors, thus providing a useful execution environment for malware analysis. We will discuss the design and implementation of system call logging and analysis under FVM

in Section 3.6.

3.4 Design of Virtualization Layer

The FVM virtualization layer is responsible for mapping access requests in a VM's namespace to access requests in the host namespace. The virtualization layer must be *comprehensive*: it should virtualize all types of OS objects to ensure the isolation between VMs. The virtualization layer must also be *efficient*: it should minimize the runtime overhead due to access redirections. Finally, the virtualization layer must be *secure*: it should not be bypassed or subverted by processes running in VMs.

3.4.1 Software Architecture

The FVM virtualization layer is inserted at the OS system call interface by system call interceptions. The Windows system calls are exposed to user applications through a set of user-level *Dynamic Link Libraries*(DLL). We prefer kernel-level interceptions because it is possible for a user process to bypass the system library interface and to make system calls directly. Filter drivers are another candidate to place the virtualization logic. However, existing filter driver interfaces are only for file access, registry access or network access, respectively. There are other types of OS objects that should be virtualized, such as kernel objects and window management. As a result, we choose the system call interface as the virtualization layer.

On NT-based Windows, basic OS services, such as file I/O, registry access and kernel object operation, are implemented in NT kernel. These system calls are well documented in [54], and their addresses are stored in the *System Service Dispatch Table*(SSDT). To virtualize file I/O, registry access and kernel object operation, the FVM virtualization layer can be implemented as a kernel driver that modifies SSDT to intercept these system calls. However, some OS services cannot be intercepted easily through SSDT modifications. These OS services include network access, window management and daemon process management. Network access on Windows goes through the Windows socket library, which invokes device-control system call with undocumented parameters. Window management are supported by system calls implemented in win32k.sys. At the time of FVM development, these system calls have no clear documentation. The daemon process management is completely under the Win32 subsystem using Win32 APIs. Each daemon control API can invoke a few NT system calls, which are difficult to correlate together. As a result, the part of FVM virtualization layer that intercepts network access, window management and daemon process management was moved to user-level

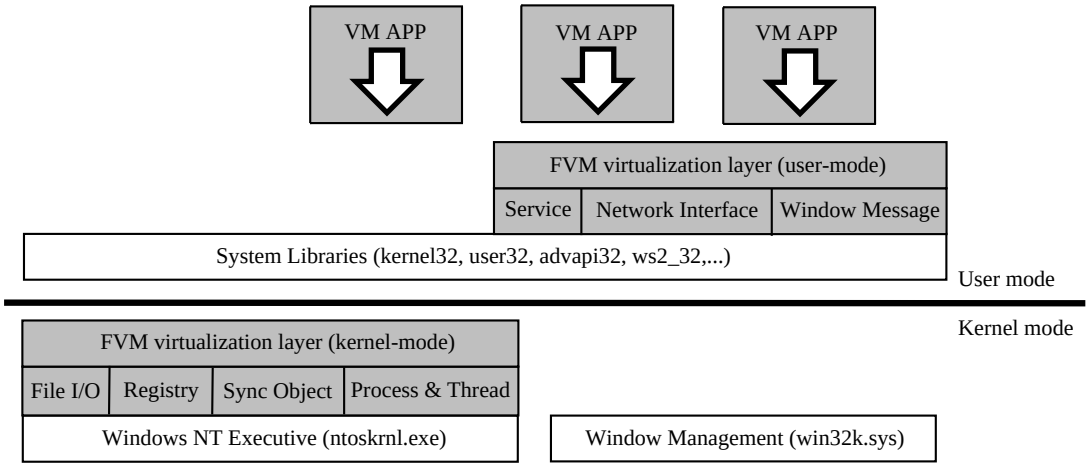

Figure 3.2: The current prototype of FVM virtualization layer consists of a kernel-mode component and a user-mode component.

interception. Network access is intercepted with a *Layered Service Provider (LSP)* DLL. Window management and daemon process management are intercepted with a DLL that modifies the *Import Address Table (IAT)* of processes in a VM. The current prototype of FVM virtualization layer consists of a kernel-mode component and a user-mode component, as shown in Figure 3.2.

3.4.2 Renaming of System Call Argument

On a system without multiple execution environments, processes accesses files, registries or other OS objects using names that are under global namespaces. When two processes use the same name to access the same type of OS objects, they are essentially accessing the same OS object. However, when a process in a VM accesses an OS object, the name it uses is under the VM's private namespace. To redirect the access to the real OS object, the FVM virtualization layer renames the name argument passed to a system call to the name under the global namespace.

There are mainly three types of OS objects a process can access: files, registries and kernel objects. Files refer to both regular files and device files; registries are the Windows configuration database containing the system-wide and per-user settings; kernel objects refer to all other named objects in memory that is used for synchronization and communication, such as mutant, semaphore, event, etc. All these types of objects are organized by a tree-style structure. To provide a full execution environment, each VM should have its private files, registries and kernel objects. Therefore, when a VM is created, three *workspace directories* are created

Type	Naming Style	Workspace Directory	Original Access	Re-directed Access
File	dir→file|dir	*e:\fvm\VMId*	c:\f.doc	*e:\fvm\VMId*c\f.doc
Registry	key→value|key	*reg\fvm\VMId*	reg\user\app	*reg\fvm\VMId*reg\user\app
Kernel Object	dir→obj|dir	*obj\fvm\VMId*	mutex0	*obj\fvm\VMId*mutex0

Table 3.1: Example of renaming system call argument. VMId is a string converted from the unique VM identifier.

on the host OS to contain the VM's private files, registries and kernel objects, respectively. When a process in a VM makes a system call, the FVM virtualization layer renames the name argument to a name under the workspace directories. Table 1 lists the three types of OS Objects on Windows, their naming styles, examples of the three workspace directories and their renaming approaches.

A process typically use two forms of system calls to access files, registries and other OS objects:

- Name-based system calls, e.g. syscall (IN name, OUT handle, $\cdots$). Given an object name, these system calls look up the object and return a handle representing the object.
- Handle-based system calls, e.g. syscall (IN handle, $\cdots$). These system calls operate the object represented by the given handle. The handle is obtained from the result of previous name-based system calls.

Because the target object of handle-based system calls is determined by the object handle, which is determined by the name of previous name-based system calls, the FVM virtualization layer only needs to intercept name-based system calls in order to redirect the access. However, implicitly renaming the system call argument has the potential risk of violating the application's execution semantic. One example is that a process may query the object name with a handle-based system call. If the handle is the result of a name-based system call whose argument has been renamed, the process gets a name under the host namespace and thus detects inconsistent behaviors. To make up for the renaming fix and guarantee the execution's consistency, some handle-based system calls must be intercepted as well to return the correct information.

3.4.3 Copy-On-Write

Renaming the system call argument to access a VM's private workspace is the first step to isolate access requests from a VM. The next question is how to redirect the access to the real OS object. A simple method is to duplicate all the files and registries from the host environment to the VM's workspace directories when the VM is created. However, this method requires large amount of disk space and memory for each VM in order to contain the VM's private file and registry copies, thus wasting system resources and compromising the scalability. Because most files and registry entries are not written during the normal execution of the processes in a VM, the FVM virtualization layer choose to share all the files, registry entries and other objects among all VMs, and to perform *copy-on-write* when a VM updates its file or registry state.

We use file object as an example to explain the copy-on-write approach in the FVM virtualization layer. After a VM is created, its private workspace directories are empty. A process's access to an existing file is always passed to the file on the host environment without renaming until the first "*open-for-write*" access to the file. Whenever a process in the VM attempts to open the file for write access, the FVM virtualization layer makes a copy of the file under the VM's file workspace directory. From then on, all types of access to this file from processes in the VM are redirected to the workspace copy. If a process in the VM attempts to create a new file, the new file is always created under the VM's private workspace. As a result, updates to file objects (as well as other types of objects) by a VM never affect other VMs or the host environment, and therefore most OS objects can be shared by multiple VMs and the host environment without compromising the system's integrity.

When the FVM virtualization layer copies an OS object on the host environment to a VM's private workspace, it does not change the security context of the thread making the system call. As a result, if the object's access control setting does not allow the thread to read the object, the copy operation will fail. Since no private copy of the object is created, FVM denied the "*open-for-write*" access. This copying strategy usually does not incur problems because it is uncommon that an object is writable but not readable. Even when such access control settings do exist on certain objects, the FVM virtualization layer can still invoke kernel-mode routines to copy the objects directly, bypassing security validations.

A side effect of *copy-on-write* is that a VM's files, registry entries and other OS objects are distributed under two locations: the host environment and the VM's private workspace. When a process in a VM enumerates OS objects under a particular directory, it should see a union view from the two locations. As a result, the system call used to make directory queries should be intercepted to return the union object

set.

3.4.4 Communication Confinement

Through namespace virtualization and copy-on-write, the FVM virtualization layer guarantees that all the *write* operations made by processes in a VM are confined within the VM's workspace. However, a process can communicate with other processes or even the OS kernel. If the communications are not confined properly, processes outside of the VM or the OS kernel can modify other VMs or the host environment as the result of the communication, thus breaking the isolation property of a VM.

A user process generally has two forms of communication channels: interprocess communications (IPC) with other processes and device I/O controls with device drivers. IPC on Windows OS is usually implemented by using synchronization object, shared memory, named pipe, socket and window message. To confine all the IPCs of a process in a VM under the scope of the VM, we need to block IPCs between the process and any other processes on the host or other VMs. For IPCs through named kernel object, FVM renames the name argument of system calls that create or open the object. Because the renaming pattern is based on the VM's unique ID, processes running in different VMs cannot establish connections based on a common name. Thus their communications are blocked. We will discuss various IPC mechanisms and their confinement issues in detail in Section 3.5.3 to 3.5.5.

However, confining IPCs of a process should not break its desirable execution semantic. There are many system daemon processes running on the host environment since the system boots, such as the Windows RPC server daemon. From the above design, the IPC between a process in a VM and those daemon processes should be blocked. This can obviously disturb the process's execution if the process relies on those daemon processes to proceed. We have designed a virtualization approach to install certain daemon processes directly in a VM and executes them in the VM's context. This is discussed in Section 3.5.6. For other system daemons that cannot be duplicated in a VM, we need to identify the communication objects they create, and to allow these objects to be accessible by processes in all VMs.

When a process in a VM uses device I/O control to communicate with a device driver, FVM is facing the same problem as it is communicating with system daemons. A conservative approach for the isolation purpose is to disable I/O controls to all the devices except the FVM device, if the I/O controls are from a process in a VM. A more practical approach is to provide a device access policy that grants necessary device accesses in order not to disrupt the process's execution. For example, to grant the network access to the Intercept Explorer in a VM, the $\backslash Device \backslash TCP$,

\Device\IP and *\Device\Afd* should be accessible at least by the Internet Explorer.

3.5 Implementation of Virtualization Components

The FVM virtualization layer supports virtualization of file, Windows registry, kernel object, network subsystem and daemon process, and confinement of inter-process communications. In the following sections, we describe implementation of each of the components, and implementation of resource constraint in a VM.

3.5.1 File Virtualization

A normal file on standard Windows file system has two names by default: a regular name (long name) and a DOS 8.3 name (short name). The DOS 8.3 name is preserved merely for backward compatibility. It allows a maximum of 8 characters for the file name portion and a maximum of 3 characters for the file extension portion. Certain application processes occasionally access a file or directory with the DOS 8.3 style of file path. Before renaming the file path argument of name-based system calls, the FVM virtualization layer checks each portion of the file path, and converts the portion to a long name if it is a DOS 8.3 name. This guarantees that the FVM's renaming and copy-on-write operate on a uniform file path without introducing inconsistencies.

FVM performs *copy-on-write* when a process in a VM attempts to write a file on the host environment. Please note that the FVM's copy approach is not really the copy-on-write, because a file is copied to the VM's workspace when a process in the VM attempts to open the file for write access. It is not exactly at the *write* time. Consequently, if a process in the VM opens a file for write but eventually never writes into it, this copy operation is wasted. In addition, because the copy operation happens on a file open request, the copy operation has to copy the whole file content. If a process opens a large file but only writes a few bytes into it, this copy approach is definitely inefficient. In fact, intercepting file write system call to copy only affected file blocks can avoid copying unnecessary file bytes.

The reasons we use "*copy-on-open-for-write*" in the FVM design are: (1) The FVM virtualization layer should be as simple as possible, since it adds additional instructions to frequently-used OS system calls. A complicated virtualization layer leads to poor runtime performance. Implementing "*copy-on-write*" requires intercepting the file write system call, which needs to maintain a mapping between file handles, and to maintain metadata about file block updates. This also brings more overhead on subsequent read system calls against this file. In contrast, file copying at the *open* time only happens the first time when this file is opened for write. Once

the file is copied to the VM's workspace, the FVM layer adds neglectable overhead for all future accesses to the file. (2) The FVM virtualization layer should support memory-mapped file I/O, in which case file contents are mapped into a process address space, and no file read or write system calls are made by the process to read or write the file data. Since intercepting file write system call does not help, the file copying has to be made at the *open* time.

A normal file is copied to a VM's workspace directory when it is "opened for write". Here the "open-for-write" access not only includes data modification, attribute update and file delete, but also include "open-for-read-exclusively". If a process in a VM opens a file for read but does not allow sharing access, subsequent access to the same file fails until the opened file is closed. Because we can start multiple VMs concurrently, exclusive access to a file shared by multiple VMs should be prevented.

A directory is copied to a VM's workspace directory when it is opened for write, e.g. attribute update. In addition, when any file entry under a directory is to be created or modified for the first time, the directory is also copied to the workspace. In another words, a private file copy in a VM's workspace is stored under the same directory structure as the the copy on the host environment. Copying a directory only creates a directory using the renamed file path. Because file entries under the directory are not copied, subsequent directory query system calls against this directory are intercepted to return the "union" result from the two directory branches in the workspace and on the host. A process queries a directory's entries by making one or more directory query system calls (*NtQueryDirectoryFile*). How many times the system call is made on a particular directory query depends on the number of file entries, required file name mask and the system call's output buffer size. The basic algorithm to generate the "union" result for a directory query is as follows:

1. A directory query system call against a private directory is executed. If it is invoked the first time on the directory, a mapping between the directory's handle and the file name mask is saved. If it is completed successfully, keep the returned file names in an ordered list associated with the directory's handle and return to the caller. If no directory entries are returned, go to Step 2.

2. Open the corresponding base directory on the host, and save a mapping between handles of the VM private directory and the host directory. Execute the directory query system call on the host directory handle with the previously saved file name mask. If it is completed successfully, traverse the returned file name and remove entries that appear in the *delete log* (a per-VM log keeping

file names the VM has attempted to delete or rename), and entries that have been returned by previous directory queries in Step 1. Return to the caller.

3. Once Step 2. has been reached, subsequent directory query system calls on the private directory handle are executed on the corresponding host directory handle. Again, if it is completed successfully, remove the duplicated file entries or entries in the delete log.

4. When a process finishes the directory query on the private directory handle, it closes the directory handle with *NtClose* system call. FVM intercepts *NtClose*, closes the corresponding host directory handle and frees all the allocated buffers.

When a normal file or directory is copied to a VM's workspace, the file attributes and security settings are also set according to the original file on the host.

One question resulted from the copy-on-write approach is that, when a process attempts to open a file, how do we know whether this file should be accessed from the host environment, or a private copy has been made and the file should be accessed from the VM's private workspace. A straightforward method is to keep all the names (including hard links, if any) of the files that have been copied in the kernel memory. Another method is to always check the existence of the file in the VM's workspace directory. If the file exists in the workspace, then the file is accessed from the workspace; otherwise the file may be accessed from the host. This method keeps away from large kernel memory overhead. However, it needs an additional system call to check the existence of a file. For I/O intensive applications that read large number of files without ever writing to them, such as a compiler or compression program, most of the file existence checking system call will miss, and FVM will have to redirect the file access back to the host. This can lead to large performance overhead. We implemented both methods in the current FVM virtualization layer. We use a binary tree to keep the file names under a VM's private workspace in memory. Each node in the binary tree stores a directory or file name. The right child of a node stores the name of its next sibling under the same parent directory, and the left child of a node stores the name of its first child. The nodes under the same directory level are sorted by the file name. We will give the performance evaluation to the above approaches in Chapter 4.

In addition to the above basic designs, the FVM virtualization layer needs to deal with a few issues specifically on NTFS file system. NTFS is the standard file system on Windows NT and later versions of Windows. It provides a few features based on normal files and directories. Although these features are rarely used by current Win32 applications, the FVM virtualization layer still needs to take care of

them when performing copy-on-write in order to ensure file operation's consistency. These NTFS features are discussed as follows.

NTFS provides *alternate data streams (ADS)* and allows a file to be associated with additional named data streams besides its main data stream. This feature is also called *file system forks*. Every alternate data stream is named with the pattern of "[file name]:[stream name]". Alternate data streams are essentially a special type of file attributes with variable sizes. When a process in a VM opens a file or one of its alternate data stream for write, the FVM virtualization layer queries the file's stream information (*FILE_STREAM_INFORMATION*), and then copies the entire file to the VM's workspace, including the main stream and all the alternate streams.

Hard links are directory entries for a file. A file can have more than one hard links. When FVM copies a file named by one hard link from the host environment to a VM's workspace, other hard links of the file should also be created under the VM's workspace. Otherwise, a subsequent read access to the file using a different hard link will be directed to the file on the host, causing inconsistent result. In NTFS, the hard links of a file are stored in the file's metadata as additional file names, which are similar to the DOS 8.3 file name of the file. To support hard links in FVM, we need to query all the hard links of a file (*FILE_LINK_INFORMATION*) when the file is copied, rename all the hard links to the names under the VM's workspace, and set up the renamed hard links on the private copy of the file.

Volume mount points, *directory junctions* and *symbolic links* are implemented by the same technique in NTFS, called *reparse points*, which allows a file system name lookup to be handled specially. A volume mount point is an empty directory under which a file system volume is attached. A directory junction is an empty directory pointing to another directory. A symbolic link is a file containing a symbolic path pointing to another file. Symbolic links are not supported on Windows versions before Windows Vista. When the FVM copies a file that is a symbolic link, the file system name lookup resolves the path of the target file. Thus a copy of the link target file is created into the VM's workspace, and the original target file is not touched. When a volume mount point or a directory junction is copied into a VM's workspace, information about the target of the mount point or junction is not propagated onto the new private directory of the VM. This is because when a directory read cannot be satisfied by the VM's private directory, the directory read will go to the directory on the host environment, which still carries the mount point or junction information. As a result, volume mount points, directory junctions and symbolic links need no special handling in the FVM *copy-on-write* scheme.

3.5.2 Windows Registry Virtualization

The Windows registry is a database where the operating system settings and per-user settings are stored. Registry data is saved statically in a set of disk files, and is loaded into the kernel memory at the time of system boots and user logins. The registry is organized with a tree-style structure that is comparable to a disk drive. It consists of a few root *keys*, which are comparable to directories. Under each key, there are other keys (called *subkeys*) and *values*, which are comparable to files. Each registry value stores one type of data, such as a string, a number, a buffer of binary data, etc. The registry key and registry value follow the same naming convention as the file system. Since a process often modifies the registry settings during its execution, the registry access must be virtualized to isolate registry changes by processes in different VMs.

FVM uses the same approach as file virtualization to virtualize the registry access. The FVM virtualization layer intercepts name-based system calls on registry key access, and renames the registry key name argument to a key name under the VM's workspace. A VM's registry workspace is embedded in the host machine's registry. More concretely, the root key of a VM is a subkey under the VM owner's per-user registry setting. When a process in a VM creates a new key or opens an existing key in order to modify certain values, the FVM layer performs *copy-on-write* to copy the key into the VM's workspace. Different from directory copying in file virtualization, whenever a key is copied, its immediate children (subkeys and values) are also copied. Please note that the copying is not recursive and stops at the immediate children of the copied registry key. Because the memory used by all the immediate children of a key is usually small, this copying approach introduces neglectable runtime overhead. However, all the immediate subkeys of the created key copy in the VM's workspace are dummy keys containing no children. When a process in the VM accesses these dummy keys later on, the FVM layer populates them with their immediate children from the host environment. With this copying approach, it is straightforward for a process in a VM to enumerate entries under a key in the VM's workspace, and the cost to perform a union operation that is usually required by a directory querying is saved.

3.5.3 Kernel Object Virtualization

Processes on the Windows OS are allowed to use certain named objects for inter-process synchronization and communication. We refer to these named objects as *kernel objects* in this paper. There are many different types of kernel objects. The most frequently-used kernel object types are *mutex*, *semaphore*, *event* and *timer*.

Windows keeps all the kernel objects in kernel memory. Similar to file and registry, kernel objects are also organized with a tree-style structure: there are object containers, called *object directory*, which contains child objects and other object sub-directories. By default, all these objects are created under the global namespace on a physical machine, and are shared by all the processes. Processes usually operate on kernel objects in the following way: the first process creates an object with an object name, and other processes can later on open the same object with the object name to test the object state. This mechanism allows multiple processes to synchronize or communicate with each other according to this object's state. When two processes are running in different VMs, however, such synchronizations or communications should not happen because these processes should appear like running on different physical machines.

Accesses to kernel objects start with opening the object with a specified name. Therefore, the FVM virtualization layer intercepts name-based system calls that create or open a named object, and renames the object name argument. As a result, a process in a VM can only create or open kernel objects under the VM's object workspace. Similar to registry virtualization, FVM creates a sub-object directory in the object directory tree of the host environment as the root object directory of the VM's workspace. Through this renaming approach, FVM resolves the single-instance-restriction problem under FVM for many Windows applications, as mentioned in Section 3.1, and enables a difference process instance of the same application program in each of the VMs.

As discussed in Section 3.4.4, the above renaming approach may occasionally disturb the execution of a process in a VM, when the process must synchronize or communicate with a system daemon process via certain kernel objects the system daemon creates. These kernel objects are residing on the host environment, because the daemon processes creating them are not running within the VM. To avoid breaking the execution of the process, the FVM layer must be aware of these kernel objects and stop renaming in the system calls made by the process to access these objects. One clue FVM uses to identify the names of these objects comes from the fact that more and more Windows applications are terminal-service-enabled. The Windows terminal server supports separation of kernel object namespace inherently in order to execute application programs in a terminal server environment. Applications running in a terminal session have their private object space for kernel objects. When such applications request accesses to global objects created by system daemons, the convention is for them to use an object name prefixed by "*Global*" so the object name lookup is performed in the global namespace instead of the terminal session namespace. Accordingly, in system calls accessing a kernel object with the "*Global*" prefix, FVM does not rename the object name and directs the access to

the host environment. This approach ensures the desired behavior for many applications under FVM. Essentially, this is a policy-based solution and is a tradeoff between the desired semantic of an application and the isolation of the VM.

3.5.4 Network Subsystem Virtualization

The purpose of network subsystem virtualization under FVM is to enable certain network applications, such a web server, to execute in one or multiple VMs without interference with each other. A network server application starts by creating a socket and making a *bind()* call to specify the local IP address and local port number for the socket. In order to have multiple instances of the same network server application to start successfully in multiple VMs, the network interface must be virtualized because the OS does not allow more than one process to bind to the same IP address and port number pair.

The FVM management console allows users to specify a different IP address for each VM when the VM is created, and then uses *IP aliasing* to assign the VM's IP address to the physical network interface: when the VM is started, its IP address is added to the host machine's physical network interface as an IP alias; when the VM is stopped, its IP address is removed from the physical network interface. Similarly, a DNS record can also be added for each VM so a remote client can use the domain name to communicate with the VM.

However, IP aliasing itself does not segregate network packets destined to different VMs. For example, when port 80 is not active in a VM, a packet destined to port 80 of this VM can still be delivered to a server process that listens on port 80 at a wildcard IP address(*INADDR_ANY*) in another VM. To resolve this problem, FVM intercepts the socket *bind* call made by a process running in a VM, and replaces the original IP address argument in the bind call with the VM's IP address. The original IP address argument in a bind call can be a wildcard IP address, an explicit IP address, or an IP address in network 127.0.0.0/8 such as 127.0.0.1. Regardless of which of the three types the IP address belongs to, FVM simply makes the network application bind to the IP address of the VM. In this way, processes in one VM can neither receive packets destined to other VMs nor spoof other VM's IP address when sending packets. Essentially, this is still a renaming approach similar to file virtualization but under the address family namespace. To allow client processes running in the same VM to communicate with the network server process, the *connect* and *sendto* calls are also intercepted. If the original IP address argument in these calls are the IP address of the physical machine or the loopback address, it is replaced with the VM's IP address. The FVM virtualization layer uses a "*Layered Service Provider (LSP)*" DLL to intercept these socket library calls.

Currently FVM does not intercept the bind call made by a server process running on the host machine. If such a process binds its socket to a port with a wildcard IP address (*INADDR_ANY*), the operating system will not allow this port to be reused by any other processes, even if they are running in a VM and binding to a different IP address. A simple solution to this problem is to apply the special socket option *SO_REUSEADDR* to all the network server processes running in VMs.

3.5.5 IPC Confinement

A process can use many interprocess communication mechanisms to communicate with other processes. To isolate one VM from other VMs, FVM requires that a process running in one VM not to communicate with processes running in other VMs or in the host environment, unless it needs to communicate with a daemon process on the host environment, or to communicate with another physical machine.

Common IPC mechanisms on Windows include named pipe, mailslot, synchronization object, shared memory, local procedure call, Windows socket, etc. In addition, there are a few Windows-specific IPC mechanisms, such as window message and clipboard. All of these IPC mechanisms and their confinement techniques under FVM are listed in Table 3.2.

A named pipe or a mailslot is a pseudofile in memory. A process can access named pipes or mailslots with a specified name in the same way as normal file access. Therefore, the FVM layer uses the same system call argument renaming approach as file virtualization to virtualize named pipes and mailslots. Virtualization of synchronization objects and network address have been discussed in previous two sections.

Shared memory on Windows is implemented with *section* objects. A section object is always backed by certain files: when it connects to a normal file, it allows memory-mapped file I/O; when it connects to an executable, it allows the image loader to map the executable image in a copy-on-write scheme; when it connects to the paging file, it provides shared memory. A process accesses shared memory by opening the corresponding section object with the object name. Therefore, the renaming approach still applies here to isolate shared memory by processes in different VMs. Similarly, the local procedure call, or local RPC, is implemented with *port* objects, which is a named object and supports high-speed message passing. Renaming the port object access at the system call interface confines the LPC within the same VM.

A process running in one VM should not be able to share the address space of processes in other VMs. Usually, a process has a private address space that is protected from being accessed by other processes. However, there are two approaches

Common IPC	IPC Confinement Approaches under FVM	FVM level
Named pipe	Rename the named pipe when creating it or connecting to it.	Kernel
Mailslot	Rename the mailslot when creating it or connecting to it.	Kernel
Mutex	Rename the mutex when creating it or opening it.	Kernel
Semaphore	Rename the semaphore when creating it or opening it.	Kernel
Event	Rename the event when creating it or opening it.	Kernel
Timer	Rename the timer when creating it or opening it.	Kernel
Shared memory	Rename the *section* object when creating it or opening it.	Kernel
Local Procedure Call	Rename the LPC *port* object when creating it or connecting to it.	Kernel
Socket	Assign an IP address to a VM by IP aliasing and associate the VM's IP address with a socket.	User (LSP)
Window class	Make a window class associated with a VM process invisible to other VMs.	User
Window message	Block window message when the sender and the receiver are in different VMs.	User
Clipboard	Store the VM ID when writing to clipboard and check the VM ID when reading from clipboard.	User
User-level hooks	Prohibit setting system-wide hooks or writing to other process's address space.	User

Table 3.2: Common inter-process synchronization and communication, and their confinement approaches under FVM.

to bypass this protection. One is to use shared memory; the other is to create a *shared* data section in certain executable file, such as a program image or a DLL. Such a data section is shared among all the processes that map the executable file containing the data section. When a process modifies the data section, the standard copy-on-write protection does not occur. Instead, every process that maps the data section can see the change. To prevent address space sharing among processes in different VMs, FVM must understand the two approaches and disable them. We have discussed how to isolate shared memory in the last paragraph. For the other approach, when an executable file on the host environment is to be mapped into the address space of a process in a VM, FVM has to check the header of each data section of the executable. If any data section is marked as shared, the executable file should be copied into the VM's file workspace. In this way, the sharing only applies to processes in this particular VM.

Window message allows a process to send various types of messages to any window on the same desktop, including windows of different processes. An example of window message is the *Dynamic Data Exchange (DDE)* message, which is a broadcast message sent by the Windows shell in order to check whether a process instance of certain applications is running. Under the FVM design, window messages should be restricted in the scope of each VM. Therefore, FVM intercepts all the window message APIs. Before a message can be sent, FVM obtains the sender's process ID and the receiver's window handle. It then queries the VM ID of the receiver process and compares it with the VM ID of the sender process. The message is dropped unless the sender and receiver processes are running in the same VM or in the host environment. Window message confinement, plus kernel object virtualization, enables many applications such as *Microsoft Office* to start a separate process instance in each VM.

The visibility of windows whose processes run in a VM should be restricted because processes in one VM are not supposed to see windows of processes in other VMs. Each window is created with a static window name and a static class name. A process can enumerate all the windows on the desktop and check their window name or class name. This allows a process to quickly find out whether certain process instances with a particular window name or class name are running. FVM intercepts the window enumeration API, and fails the enumeration result if the enumerated window and the calling process belong to different VMs. Window visibility confinement, plus kernel object virtualization, enables more applications such as *Adobe Acrobat Reader* to start a separate instance in each VM. To make users recognize that a process is running in a particular VM, the title of any window belonging to this process is appended with the VM ID.

The Windows clipboard enables a process to share data in various format with

all other processes. The *copy-and-paste* and *drag-and-drop* GUI operations are implemented by the clipboard. On the Windows OS, all processes associated with the same *window station* share a common clipboard, and can write or read data with a set of clipboard APIs. For isolation purpose, processes in one VM should not share clipboard with processes in other VMs. Therefore, the FVM virtualization layer intercepts clipboard read and write APIs. Whenever a process in a VM writes data into the clipboard, FVM records the VM ID in a shared memory area to associate the VM with the current clipboard data. Later on, when processes in other VMs try to read the clipboard data, FVM compares the process's VM ID with the recorded VM ID in the shared memory area, and rejects the clipboard access unless the two VM IDs are equal. In addition to this simple isolation, FVM can further completely separate the clipboard for different VMs by intercepting the same clipboard access APIs and maintaining the raw clipboard data by itself with shared memory. More concretely, when a process in a VM attempts to write data into the clipboard, FVM redirects the data to shared memory maintained by FVM itself instead of the original Windows clipboard. As a result, such data is not accessible by other processes with a different VM ID.

Finally, a process running in one VM should not be able to modify the address space of processes in other VMs, using *hooks* or direct modifications. A hook is a point during message handling where a callback is invoked by the system to monitor or modify the message traffic. A process is allowed to install a system-wide hook for certain types of messages. The hook usually provides the callback routine in a DLL, which is loaded by the system into all the processes with access to that type of messages. When the DLL is being loaded into a process, it can modify the process's address space. In addition to hooks, a privileged process is allowed to open a process and directly modify the process's address space. To prevent hooks and direct modifications from within a VM, FVM ensures that one process in a VM cannot open other processes' address space unless they are in the same VM. With the help of Windows *job* object, FVM further restricts the system-wide hook under the scope of the VM where the hook is installed.

3.5.6 Daemon Process Virtualization

A daemon is a process running in the background without interacting with users. On Windows OS, a daemon process is called a *service*, and is managed by a system process called *Service Control Manager (SCM)*. The SCM maintains a database where names, image paths, and start parameters of all the daemons are stored. In order to run as a daemon process, a daemon program must use specific Win32 APIs to register with the SCM and to keep its state running. Users can also install new daemons by adding the daemon's name and its program image path into the SCM

database. To start a daemon process, users must invoke specific Win32 APIs that request the SCM to create the daemon process.

FVM associates a new non-daemon process with a VM if the process is the descendant of the VM shell process. Different from a non-daemon process, the parent of a daemon process is always the SCM. The SCM is a critical system process with tight communications with other OS components, and thus cannot be duplicated in each VM. Therefore, the FVM layer needs a special mechanism to place a daemon process in a VM's context. Meanwhile, all the daemon processes, regardless of running on the host or in a VM, have to communicate with a single SCM and SCM database.

The basic technique to place a daemon process in a VM is as follows:

1. Reinstall the daemon in the VM using daemon installation API. The daemon installation API is intercepted to install the daemon with a different name containing the VM ID. Moreover, the daemon's image file is copied to the VM's workspace, and its image path to be stored in the SCM database is also renamed to the path containing the VM's root directory. For example, if the daemon named `S` with a program image path `/a/b.exe` is to be installed in a VM (say `vm1`), the actual daemon's name and image path added to the SCM database will be `S-vm1` and `vm1/a/b.exe`, respectively. In addition, the image file is also copied from `/a/b.exe` to `vm1/a/b.exe`. Without changing the daemon's name or the image path name in the SCM database, SCM will not allow multiple daemon processes of the same program to be installed or started in different VMs.

2. Start the daemon process from the VM using the daemon opening API. This API takes a daemon's name as the argument. FVM intercepts this API and renames the requested daemon name argument from `S` to `S-vm1`. The SCM will then starts the daemon whose name is `S-vm1`.

3. The kernel-mode FVM layer monitors the creation of every process. Once FVM detects that a process is to be created by the SCM and its image file is under a VM's workspace, such as `vm1/a/b.exe`, FVM can infer that the process is a daemon process which should be associated with a VM's context. The target VM ID can be obtained from the path of the image file. Then the ID of the new process is associated with the VM ID, and the daemon process is placed in the VM.

With the above renaming technique, FVM is able to virtualize a number of daemon processes such as Apache and MySQL. With the help of the network isolation component, FVM can start one instance of Apache or MySQL server in each of the

VMs. However, such renaming technique introduces inconsistency for some daemon programs, because the daemon's original name still exists within some daemon's program code or certain registry settings. Although renaming the daemon's name has no problem at the daemon's installation time, it can cause inconsistency that breaks the daemon process at run time. There is no generic solution to resolve all the inconsistency issues due to renaming a daemon's name in the SCM database. Our future goal is to try to emulate the SCM by intercepting and modifying the entire daemon control library. In this way, FVM does not need to rely on the Windows SCM to support daemon processes in multiple VMs.

3.5.7 Process-VM Association

When a VM is started, there is only one process, the *VM shell* process associated with the VM context. All the descendant processes of the *VM shell* process should be associated with the same VM. To dynamically maintain a mapping between the VM ID and a process ID, we register a process creation call-back routine (*PsSetCreateProcessNotifyRoutine*) through the FVM kernel driver. The call-back routine is invoked when a process is created but before the process starts execution, receiving the process ID and its parent process ID. We look up the VM ID associated with the parent process ID, and then associate the new process ID with the same VM ID in the kernel. In this way, given an access request, the FVM layer can look up the requesting process's VM ID, based on which it renames the system call argument accordingly. In addition, the FVM layer does sanity checks on a process's VM ID and its access request to ensure that a process running in a VM cannot access other VM's private workspace for files, registries and kernel objects in any way.

3.5.8 Resource Constraint

FVM allows users to specify certain policies to restrict the system resources used by a VM, because processes in all VMs are running on the same OS kernel. Without resource constraint, a malicious process in one VM can consume large amount of disk space or physical memory, leading to poor performance or even no access in other VMs and the host environment.

To control the memory usage and scheduling priority of each VM, FVM utilizes the Windows *job* object, which is a container of a group of processes. A job object allows these processes to be managed as a unit. A set of constraint on resource usage can be set on a job object, which enforces the constraint to every process associated with the job. The FVM management console associates each new VM with a job object, and initialize the job attributes with the resource constraint policies specified by the user. FVM also ensures that all the processes created in the

VM are associated with the job object. To control the disk space and registry size of a VM, the FVM management console periodically calculates usage of these resources and compares it with the maximum allowed size. When any type of the resource usage exceeds its limit, the VM is suspended for inspection by users. Although not implemented, FVM can further restrict the network traffic of a VM to stop worms or spam generators running in the VM.

The current resource usage constraints on a VM include the following:

- Maximum size of a VM's private file workspace.
- Maximum size of a VM's private registry workspace.
- Maximum size of committed memory of all processes in a VM.
- Maximum working set size of each process in a VM.
- Scheduling priority of all processes in a VM.
- Maximum number of concurrent processes in a VM.

Besides resource constraints, a VM's job object can also prevent processes in the VM from the following operations:

- Modify the VM's resource constraint settings.
- Modify system settings of the physical machine, such as screen settings, icons, fonts, mouse speed, power settings, etc.
- Broadcast window message to windows outside of the VM.
- Install hooks on processes outside of the VM.
- Log off the current session.
- Shut down or restart the physical machine.

3.5.9 System Call Interception

The FVM virtualization layer at the kernel-mode is a kernel driver, which is installed and loaded by the FVM management console. In contrast, the FVM layer at the user-mode is a DLL, which is loaded into each process's address space by the FVM management console. To guarantee the isolation of processes running in different VMs, all the access requests from all the processes in VMs must pass the

FVM layer at any time. In another words, the FVM layer should be in place when a process in a VM starts execution. The kernel-mode FVM layer always fits this requirement because the FVM driver is loaded and the system call table is patched before a VM is created. However, the user-mode FVM layer relies on the IAT modification in a process's address space. It must be loaded at the right time: after all the modules' IATs of the process have been filled by the OS loader, and before the execution of the process. Currently FVM intercepts the process creation flow in the kernel, notifies the FVM management console to modify instructions at the program entry point, and then forces the process to load the FVM layer after the IATs' initialization and before the process execution.

Within an intercepted system call, the FVM layer usually calls the original system call function with a renamed argument. This is straightforward in user-level interception, but when intercepting system calls at the kernel level, it requires special handling to the renamed argument: the renamed argument should be stored in a separate buffer that is allocated from the calling process's address space (using *ZwAllocateVirtualMemory)*. It cannot be allocated from kernel memory. This is because we directly invoke the original system call function, which always checks the previous processor mode to find out where the system call is invoked: from user mode or from kernel mode. If it is originated from a user process, it assumes all the buffers of arguments reside in the user space. The system call function code validates all the input buffers and fails the request with access violation if it recognizes a buffer is in the kernel memory.

Table 3.3 lists all the system calls and Win32 APIs intercepted by the FVM virtualization layer. The interception mechanism in FVM is designed to be extensible so that it can serve as a reusable framework for other projects that require system call interceptions. Initially the FVM prototype was implemented and tested on Windows 2000 Professional, Service Pack 4. We have recently ported and tested FVM on 32-bit Windows XP, and we expect FVM to run on 32-bit Windows Server 2003, and even 32-bit Windows Vista with moderate changes. However, x64-based Windows kernel for XP, 2003 and Vista utilize PatchGuard [52] to prevent patching to critical kernel structures, including the system call table. To support virtualization on these Windows kernels, FVM needs to either disable the PatchGuard with unfriendly approaches, or move the virtualization layer to other interface, such as filter drivers. The current FVM prototype intercepts 40 out of 248 system calls on Windows 2000/XP and additional 21 Win32 API calls, with around 10,000 lines of C code in kernel, and an equal amount of user-level C/C++ code. We plan to open-source most FVM code by the end of this year.

Category	Intercepted System Calls and Win32 APIs by FVM
File	NtCreateFile, NtOpenFile, NtQueryDirectoryFile, NtQueryAttributesFile, NtQueryFullAttributesFile, NtQueryInformationFile, NtQueryVolumeInformationFile, NtSetInformationFile, NtDeleteFile, NtClose
Registry	NtCreateKey, NtOpenKey, NtQueryKey, NtEnumerateKey, NtDeleteKey
Synchronization Object	NtCreateMutant, NtOpenMutant, NtCreateSemaphore, NtOpenSemaphore, NtCreateEvent, NtOpenEvent, NtCreateTimer, NtOpenTimer, NtCreateIoCompletion, NtOpenIoCompletion, NtCreateEventPair, NtOpenEventPair
Special File	NtCreateNamedPipeFile, NtCreateMailslotFile, NtCreateFile, NtOpenFile
Shared Memory	NtCreateSection, NtOpenSection
LPC	NtCreatePort, NtCreateWaitablePort, NtConnectPort, NtSecureConnectPort
Socket	bind, connect, sendto
Device	NtCreateFile, NtOpenFile
Window Message	SendMessage, SendMessageTimeout, SendNotifyMessage, SendMessageCallback, PostMessage, PostThreadMessage, GlobalAddAtom, DdeCreateStringHandle
Window Visibility	FindWindow(Ex), EnumWindows
Clipboard	GetClipboardData, SetClipboardData
Normal Process	NtResumeThread, PsSetCreateProcessNotifyRoutine
Daemon Process	CreateService, OpenService, StartServiceCtrlDispatcher, RegisterServiceCtrlHandler(Ex), NtResumeThread
FVM Protection	NtLoadDriver, NtUnloadDriver, NtCreateFile, NtOpenFile, NtOpenProcess, NtProtectVirtualMemory

Table 3.3: System calls or Win32 APIs intercepted for virtualization. Functions whose name starts with “Nt” are NT system calls.

3.6 System Call Log Analysis for *CommitVM*

In addition to access redirections and communication confinement, the FVM layer has the ability to log system calls or API calls to derive high-level behaviors of processes running in a VM. The logging and analysis modules allow FVM to monitor untrusted programs in a VM, to recognize the scope of each potential compromise by malicious programs, and finally to *selectively commit* only legitimate state changes.

3.6.1 Introduction

When processes inside a VM modify files or registry entries, these modifications are confined under the VM's workspace. By looking for updates to sensitive registry and file locations in the VM's workspace, we can detect if certain processes have made potentially malicious changes in the VM environment. When such changes are found, we discourage users from committing these changes to the host environment. However, examining permanent state changes confined inside a VM is not sufficient to detect all the suspicious behaviors or to recognize the exact scope of the attack due to malware execution. For instance, a mass mailing worm can simply collect email addresses by reading the Windows address book, and then send itself to every email address it finds. Because it does not attempt to modify any permanent state, we cannot detect such behavior by monitoring the VM's private workspace. Moreover, even after we detect a few malicious state changes by some processes, we cannot find out who they are and what other changes they have made in the VM's workspace. To identify all the state changes by suspicious processes in a VM in order to provide an accurate *selective commit* on the VM's state, it is necessary to log and analyze system calls made by processes running in a VM.

Log analysis has been widely used in intrusion and anomaly detection, automated intrusion recovery and dynamic malware analysis. Typically, anomaly detection systems [55, 56, 57] based on system call sequences collect normal behaviors of the monitored program by logging system calls it makes at run time. The system call traces are further separated into multiple sequences with certain length, thus composing a normal behavior dictionary of the program. With the normal dictionary available, the detection system compares the program's system call traces with elements of the dictionary, and flags an anomaly when the number of unknown system call sequences exceeds a threshold.

The repairable file service [45] is a post-intrusion recovery technology that can quickly repair compromised network file servers. It logs all file update requests sent to the network file servers, as well as file-related system calls on each NFS

client. After an intrusion is detected, it performs contamination analysis that correctly identifies other processes that may be contaminated, and undoes their file update operations. The Taser intrusion recovery system [46] uses similar approaches to recover legitimate file system data after an attack occurs. It audits system calls related to process management, file system and socket, identifies all the tainted file system objects due to an attack, and finally redoes legitimate operations from a good file system snapshot.

A few dynamic malware analysis tools execute suspicious code in a safe environment, log system calls or API calls, and analyze the call sequences to identify malicious behaviors. The Norman Sandbox [58] simulates multiple versions of Windows OS and Win32 libraries to create a *faked* Windows environment to test malware. It detects high-level behaviors of malware by analyzing Win32 API call logs. Similarly, CWSandbox [59] intercepts Win32 API calls of malware samples executed in a virtual environment to collect the log data such as function call parameters, and analyzes the data to generate a detailed behavior report. TTAnalyze [60] uses Qemu PC emulator to execute untrusted programs because a PC emulator is more difficult to be detected by malware than virtual machines such as VMware. It intercepts both Win32 and native NT API calls by a callback routine invoked at the start of each translation block, which can be the first instruction of an API function. Differently, SysAnalyzer [61] uses a snapshot-based approach rather than log analysis to report a binary program's behavior. It takes snapshot of active processes, opening ports and sensitive registry entries before and after the execution of a binary program, and reports the difference between the two snapshots. However, this approach may not record all the dynamic malware behaviors. The system call logging and analysis modules in FVM can also be used to perform a thorough dynamic analysis to untrusted programs.

The system call logging and analysis modules in the FVM are designed to derive high-level process behaviors from low-level system call sequences. By comparing these high-level behaviors with typical malware behaviors, we can conclude whether any processes in a VM are potentially malicious. Therefore, when users run certain network applications, such as web browsers or email clients, inside a VM, they can accurately identify benign processes and their permanent changes to the VM. These state changes are definitely legitimate and thus can be committed to the host environment. Other state changes are kept inside the VM and may be deleted. We will describe the design and implementation in the following sections.

3.6.2 Design Overview

Typical malware behaviors include copying itself to startup folders or system directories, injecting itself to legitimate processes, terminating security-related processes, adding an entry to auto-start extensibility points in the registry to execute itself automatically, collecting email addresses for mass mailing, and etc. These operations must be performed by calling Win32 APIs or NT native APIs, which in turn invoke NT system calls to complete the tasks. Therefore, we can log these API calls or NT system calls made by processes running in a VM. There are a number of existing logging tools on Windows, such as Strace for NT [62] that logs system calls, API monitor [63] that logs API calls, and the VTrace tool [64] that logs both. Both the API call logging and the system call logging have their advantages and disadvantages in supporting the log analyzer to recognize high-level behaviors. Logging system calls only needs to intercept a small number of system call functions, because the set of Windows API calls is much larger than the set of underlying NT system calls. A single operation such as creating a process can be accomplished by various API calls, all of which are built on a single system call. In addition, logging system calls is more reliable because malicious programs can directly invoke system calls without going through any API call interface. On the other hand, logging API calls often makes the log analyzer easier to recognize operations on certain OS components implemented by the Win32 subsystem, such as the Win32 services and remote registries. One API call on these components is often accomplished by a sequence of system calls whose high-level behaviors are not easy to be extracted. In the current FVM prototype, we implement logging within the FVM virtualization layer, which is mainly at the OS system call interface.

The log data generated by the FVM layer consists of API call or system call logs of multiple processes running in a VM. It may also come from different sources, such as the system call log and the network event log. Such log data is processed by the log analyzer in the following four steps, as shown in Figure 3.3.

1. *Preprocessor*. The preprocessor decomposes the log data of a VM into a set of sub-logs, each of which is associated with a process running in the VM. If the log data comes from existing logging tools, it rewrites each log entry to a common format. If the log data comes from multiple log files, it still needs to correlate log entries in all the log files. The output of this step is a set of formatted per-process log files, each of which is composed of function calls with key input/output arguments.

2. *Dependency analyzer*. The dependency analyzer walks through each log and decomposes it into a set of function call sequences. A function call is related to other function calls in a sequence through data dependency between their

input/output arguments. An example of the data dependency is a *ReadFile* API followed by a *WriteFile* API, where the latter API's input buffer address and the number of bytes to write match the former API's output buffer and the number of bytes that have been read. Under this condition, the two API calls are related together in a sequence, which may indicate the file copying behavior. However, sometimes it is not obvious that the output argument of a function call is used as an input argument in another function call, because the output has been transformed (e.g. copied to another buffer and then encrypted) before being passed to another function call as an input. In this case, additional taint analysis procedures are required to relate these two function calls. This problem is not addressed in our current prototype.

3. *Function normalizer*. The function normalizer converts each function call in the sequence into a normalized function call, according to a mapping stored in a configuration file. Because sometimes there are more than one API calls that can perform the same operation, without this step, the behavior analyzer has to understand all these API combinations just to identify one behavior. The function normalizer also removes unimportant function calls in a sequence, such as the file open or close function calls. This step simplifies the task of behavior analyzer when comparing the function call sequences with behavior templates, especially when the logged function calls are high-level API calls.

4. *Behavior analyzer*. The behavior analyzer parses the normalized call sequences of each process, and compares the sequences and root input arguments with behavior templates stored in a configuration file. Definitions of the behavior templates are mainly empirical based on the behaviors of known malware [65]. Whenever a match is found, the behavior analyzer outputs the corresponding behavior to the analysis report.

When a process being analyzed exhibits a number of suspicious behaviors, FVM will classify this process as a malicious process. All the file and registry updates by the process are treated as malicious. Moreover, we can also perform a contamination analysis [45] based on the inter-process dependencies in the original function call logs to recognize other processes *contaminated* by the process, and prevent their state changes from being committed to the host environment.

3.6.3 System Call Logger in FVM

As described in Section 3.4.1, the FVM virtualization layer consists of a kernel-mode component and a user-mode component. The kernel-mode component logs

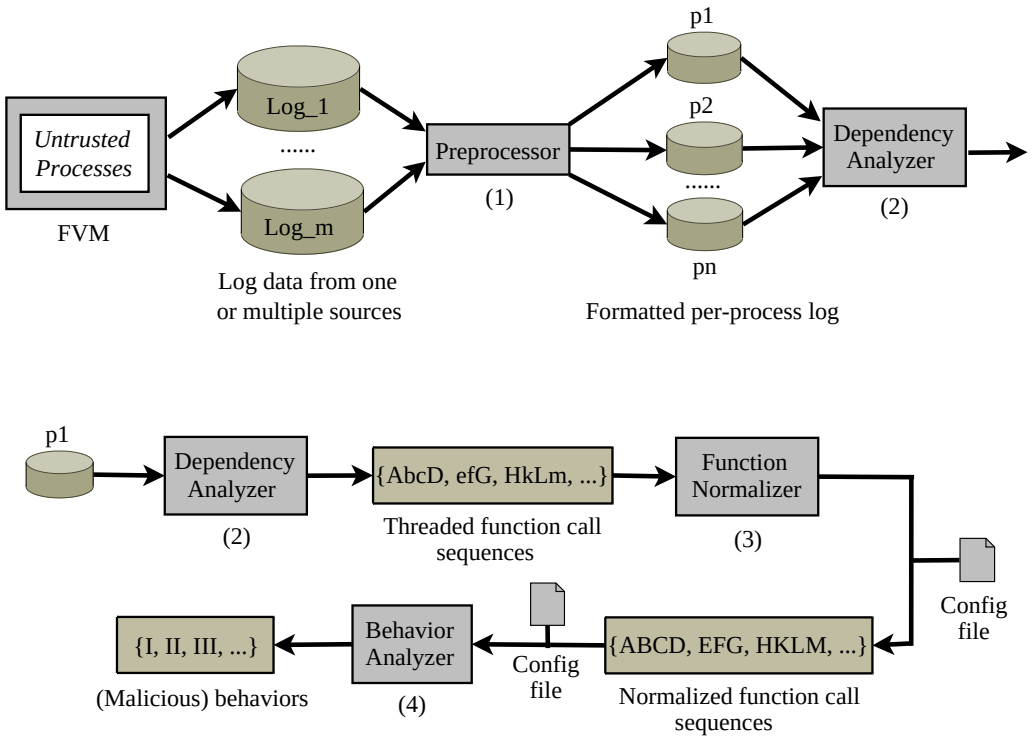

Figure 3.3: The FVM log analyzer consists of four components: preprocessor, dependency analyzer, function normalizer and behavior analyzer. The four components process log data sequentially and output high-level behaviors of tested malware.

Category	Additional Logged System Calls and Win32 APIs by FVM
File	NtReadFile, NtWriteFile
Memory-mapped File	NtMapViewOfSection
Registry	NtSetValueKey, NtQueryValueKey, NtDeleteValueKey
Address Space	NtWriteVirtualMemory
Socket	send, recv

Table 3.4: Additional system calls and Win32 APIs intercepted by FVM. These function calls are intercepted to provide log data only. Functions whose name starts with "Nt" are NT system calls.

selected NT system calls and their arguments, and the user-mode component logs a few network API calls and their arguments with a layered service provider DLL. The current system call logger in FVM logs 19 NT system calls and 3 Win32 API calls, which include a subset of function calls shown in Table 3.3, and a few newly-intercepted function calls, as shown in Table 3.4.

We decide which function calls to log by observing malware's typical behavior. For example, malicious code often attempts to copy itself to other executables. The file copy operation is usually implemented by opening the source file, creating the destination file, reading the source file into a buffer and then writing the same buffer to the destination file. Alternatively, the file copy operation can also be implemented using memory-mapped file I/O, which maps the file into the process address space and duplicates the file content directly to the destination file. To correctly recognize such behaviors, we need to intercept *NtReadFile* and *NtWriteFile* for the explicit file I/O interface, and to intercept *NtMapViewOfSection* for the memory-mapped file I/O interface.

The logged data for each function call includes a *time stamp* represented by a high-resolution performance counter, the *process id*, the *thread id*, the *function call id*, the *return value*, and the *input/output arguments*. A system call or API call can have a number of input and output arguments, and only certain arguments are necessary to be logged for behavior recognition. We classify the function call arguments to be logged into the following two types:

1. *Identity arguments*, which are the target of a function call or a function call sequence, such as a file name, a registry key name, a process id, etc. These arguments are logged to allow the analyzer to recognize a process's operations on sensitive targets.

2. *Dependency arguments*, which are certain abstraction of the target of a function call, such as a file handle, a registry key handle, a process handle, an

I/O buffer (address and size), etc. These arguments are logged to allow the analyzer to correlate function calls into a function call sequence representing a high-level behavior.

Logging only the identity arguments and the dependency arguments separate the dependency analyzer component from detailed function call semantics.

Once we have collected all the necessary information in a function call to be logged, we append an log entry to a log queue in the memory. For performance reason, we cannot write the log entry directly into a log file inside an intercepted function call. Instead, we rely on a worker thread to periodically examine the log queue and to move available log entries into the log file. In the kernel-mode component of the current FVM layer, we use one kernel thread to write all the system call log entries into a log file shared by all the processes. In the user-mode component of the FVM layer, we use a separate user thread in each process to write the process's network log entries into a per-process log file. Because of the asynchronous writing to log files, the system call logger introduces negligible performance overhead (1% to 3%) to the original FVM framework. The log files are explicitly stored in the host environment and thus are protected from being tampered by malware inside a VM.

3.6.4 Log Analyzer

As shown in Figure 3.3, the log analyzer processes log data in four steps: preprocessing, dependency analysis, function normalization and behavior analysis. The preprocessor operates on the system call log file and the per-process network log file, merges them according to the high-resolution time stamp of each function call, and outputs a function call log per process. Each log entry represents one function call with its input/output arguments and occupies one line in the log file. The field of input/output arguments in each log entry has the following format:

IN:[argument] [argument] ... OUT:[argument] [argument] ..., where each argument is represented as: *[type]:[field]?[field]...* Currently we consider three types of arguments: a name (N), an identifier (I) and a buffer (B). Table 3.5 shows a few examples of logged arguments in the log file.

With the formatted per-process log file, the dependency analyzer exploits the data dependency between the input and output arguments of function calls to discompose the log into function call sequences. There are mainly three types of data dependencies: (1) *handle-based dependency*, (2) *buffer-based dependency* and (3) *content-based dependency*. Suppose there are two function calls A and B, then A depends on B if one of A's input arguments is a handle that is one of the output arguments of B, or one of A's input arguments is a buffer that is within the output

Function calls	Logged arguments	Descriptions
NtCreateFile	IN:N:c:\abc.txt OUT:I:32	Create a file "c:\abc.txt" and return its handle 32
NtReadFile	IN:I:32 OUT:B:17c3b8?1024	Read 1024 bytes from a file with handle 32 into a buffer starting at 17c3b8
send	IN:I:44 B:17c3b8?1024 OUT:	Send 1024 bytes starting at 17c3b8 through a socket 44
NtWriteFile	IN:I:48 B:17c3b8?512 OUT:	Write 512 bytes starting at 17c3b8 into a file with handle 48
NtOpenProcess	IN:I:868 OUT:I:120	Open a process whose ID is 868 and return its handle 120

Table 3.5: *Examples of the logged arguments format of five function calls. The I/O buffer (B) arguments are logged with both the address and number of bytes read or to write.*

buffer argument of B, or the memory content of A's input buffer is part of the memory content of one of B's output buffer. Handle-based dependency can be used to correlate function calls that operate on the same target (e.g. a file or a registry key) in one task. Buffer-based and content-based dependency can be used to correlate function calls that work on some common data (e.g. content of a file being copied). Buffer-based dependency allows us to recognize two related function calls as long as they are working on the same buffer, even if the content in the buffer has been transformed between two function calls, or the second function call works on the buffer partially. In contrast, content-based dependency allows us to correlate function calls which work on different buffers with the same content. Therefore, if the content is copied from the original buffer to a new buffer which is passed to the second function call, we still can correlate the two calls. Our current system call logger does not log content in an input or output buffer, and thus we only analyze the handle-based dependency and buffer-based dependency in our log analyzer.

A log entry contains the information of one function call and its input/output arguments, and is a string occupying a full line in the log file. The output of the dependency analyzer is a new log file consisting of function call sequences, each of which is separated from one another by an empty line in the new file. The dependency analyzer parses the original log file twice to generate the new log file. In this first time, the analyzer constructs a linked list of sequence nodes, each of which points to a set of output arguments of all the function calls belonging to the same sequence, and a list of file pointers of each function call's log entry in

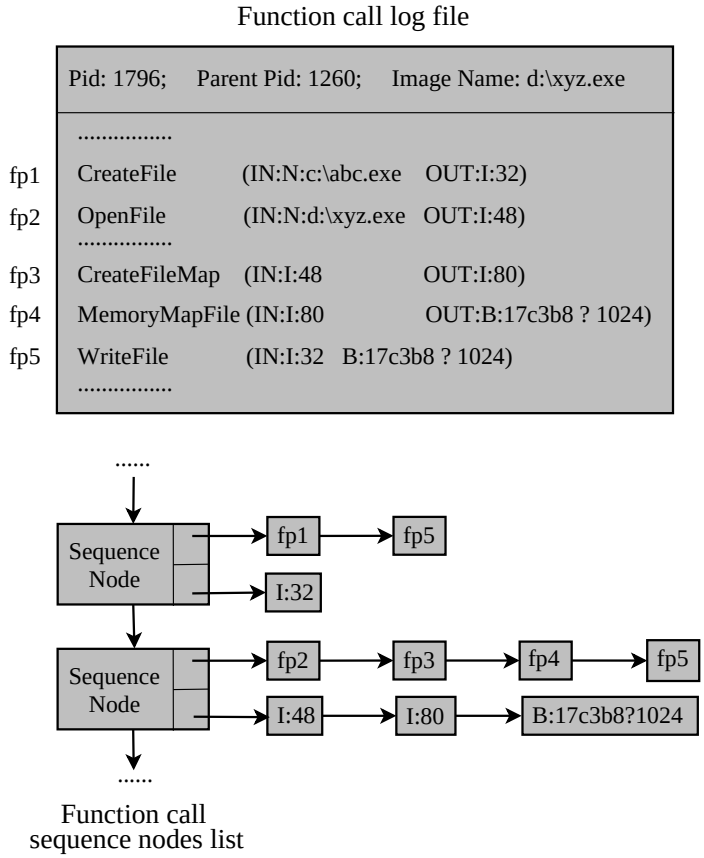

Figure 3.4: An example of function call log entries and the sequence node data structure used by the dependency analyzer.

the original log file. Once the list of sequence nodes for all the function calls are generated, the dependency analyzer walks through the list of sequence nodes, reads appropriate log entries belonging to each function call sequence by moving the file pointer in the original log file, and then writes the log entries into a new log file to form separated function call sequences. Figure 3.4 gives an example of the sequence node data structures used by the dependency analyzer.

The dependency analyzer generates the list of sequence nodes in the following ways: given a function call's log entry in the original log file, we obtain its input and output arguments, and search its input arguments in the output arguments list of each existing sequence node. If we find a match between one of the function call's input arguments and one output argument associated with a particular sequence, we associate the function call with this sequence by inserting its log entry's file pointer in the original log file into the sequence node's file pointer list. Moreover, if the

function call's input arguments find matched output arguments in more than one sequence node, we consolidate all the sequence nodes into one sequence node. If the function call has its own output argument, this argument is also added into the output arguments list of the container sequence. Expanding the output arguments list allows us to identify more function calls related to this sequence. If none of the function call's input arguments match the output arguments list of any existing sequences, this function call is treated as the start of a new function call sequence. We create a new sequence node, initialize its output arguments list and file pointer list, and then move on to process the next function call's log entry.

An output argument of a function call sequence should be removed from the sequence node's output arguments list when it becomes invalid or has been reused. For the handle-based dependency, this means an output handle is closed by a function call. For the buffer-based dependency, this mean an output buffer is reused as an output argument in a function call. When we meet such function calls during the analysis, we remove the corresponding output arguments from the output arguments list of the existing sequence. This operation not only reduces the argument matching overhead, but also avoids the mistake of associating disjunct function call sequences into one sequence.

After the dependency analysis is completed, a new log file is generated containing a set of function call sequences. As an optimization, the function normalizer reduces the number of function calls in each function call sequence by eliminating log entries of repeated function calls, and replacing dependency arguments of certain function calls with identity arguments. In general, any function calls unnecessary for behavior recognition should be removed in the output log file of the function normalizer. Most Win32 API call or NT system calls operate on certain *handles* and rely on the process handle table for name lookup. These handles must be created through certain *Open* calls, and destroyed through certain *Close* calls. Such *Open* calls and *Close* calls do not indicate meaningful application behaviors and thus can be removed. For example, an *OpenFile* call followed by multiple *ReadFile* calls operating on a file handle can be replaced by one *ReadFile* call operating on the file name, and a *CloseFile* call can be simply ignored in the output file.

The function normalizer also converts different function calls performing similar operations to one generic function call, according to a configuration file. This is especially useful when the log data is composed of Win32 API calls, many of which can perform similar operations. For example, the Win32 calls *ReadFile*, *ReadFileEx* and *LZRead* all read file content into a buffer. Therefore, we can use a generic function name *File_Read* to replace all the occurrences of these function calls in the log file. In addition, it is possible to normalize function calls that use different mechanism to perform the same operation, e.g. to normalize function calls

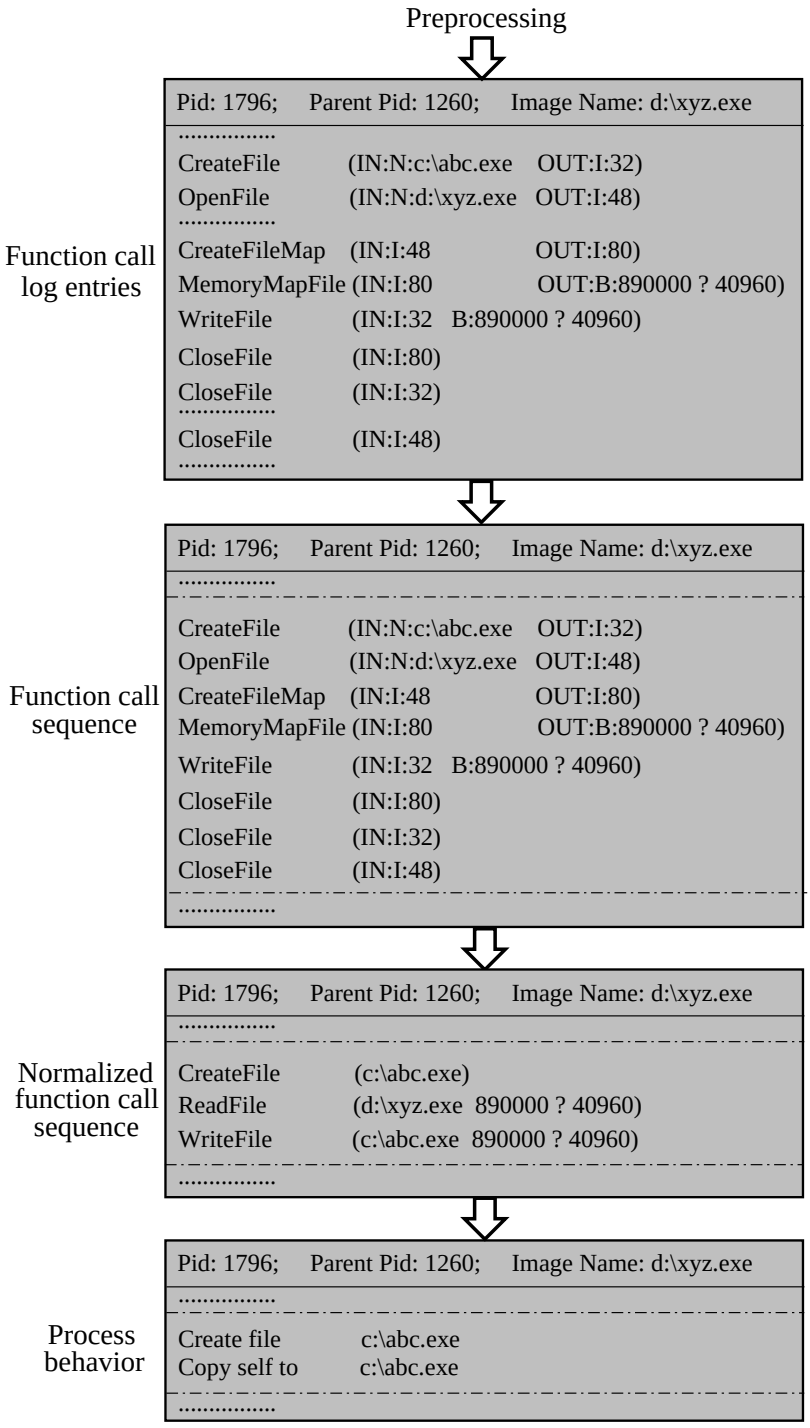

Figure 3.5: The log analyzer processes the input log file step by step and identifies one type of *copying itself* behavior of the analyzed process.

Behavior template configuration file

```
.................

[Sequence]
 ReadFile (%ProcImage%, %p, %u)
 WriteFile (%s, %p, %u)

[Behavior]
 Copy self to %s
.................
```

Figure 3.6: An example of the behavior template used by the behavior analyzer to recognize the *copying itself* behavior.

used for explicit file I/O and memory-mapped I/O to a generic call *File_Read*.

The last step in the log analyzer is to extract the process's high-level behavior according to behavior templates in a configuration file, which provides a mapping from generic function call sequences to the behaviors. For each matched behavior and matched input arguments such as an ASEP registry key, the corresponding high-level behaviors are output to the analysis report. Certain behaviors correspond to one function call such as registry modification, while other behaviors correspond to a sequence of function calls such as file copying. Our current prototype has successfully detected a few suspicious behaviors, including copying itself to other executable, modifying the hostfile to redirect network traffic, modifying 9 ASEP registry locations, accessing the Windows address book, reading a file and sending it to a remote site, and etc. These behaviors are often exhibited in recently-found malware, such as *Trojan.Qhosts.E*, *W32.Minera.A*, *W32.Stemclover* [65]. Figure 3.5 shows the changes made by the log analyzer on initial log entries to recognize one type of *copying itself* behavior. Figure 3.6 shows part of the related behavior template file used by the behavior analyzer. Extending the basic log analyzer framework to recognize more high-level behaviors of a process is one of our future work.

3.7 Limitations of FVM

3.7.1 Architectural Limitations

Under the FVM architecture, all the VMs share the same operating system kernel with the virtualization layer at the OS system call interface. Therefore, FVM

must restrict access to the OS kernel from untrusted processes in a VM. Without this restriction, a process in a VM can subvert the FVM layer to break the VM's isolation, or compromise other kernel components to corrupt other VMs and the host environment. There are mainly two approaches for a process to make changes to kernel memory: (1) loading or unloading a kernel driver; (2) writing to physical memory by accessing *\Device\PhysicalMemory*. To protect the FVM layer and to ensure a VM's isolation, FVM denies the request of loading or unloading kernel drivers or mapping physical memory from a process in a VM. This restriction may conflict with the desired behavior of certain applications running under FVM if they require kernel memory access.

Similarly, since device drivers and some system daemons are shared by all the VMs, device access and inter-process communication with these system daemons are also restricted. This is necessary to guarantee that a VM's state is isolated from other VMs or the host environment. Since there are many legitimate reasons for a process to access device, such as connecting to the network or setting up a volume mount point, FVM allows users to specify a list of devices that are allowed to access. Other devices that are not in the list are denied to access. Although FVM has virtualized daemon process management to start certain daemon processes in every VM, some daemon processes still need to be shared by multiple VMs. These daemons are inappropriate or difficult to be started in each VM, either because they are a critical component of system booting, or because they have close dependencies on some kernel drivers. Therefore, a VM's IPC to these daemons should be enabled sometimes to allow processes in the VM to execute properly. Ideally, FVM should have a virtualization component at the interface of every device and every system daemon. Each of the components is able to redirect access requests originated from a VM to that VM's workspace. However, this model requires understanding each underlying protocol in detail and therefore is not a generic solution. Essentially, the communication confinement problem is an inherent limitation of the OS-level virtualization methodology. To reduce resource requirement and to improve scalability, OS-level virtualization allows as many OS components to be shared as possible. Unfortunately, there is no generic way to construct a virtualization layer that covers all types of interfaces, especially the customized device/IPC interfaces. For interfaces that has no virtualization available, access requests to these interfaces should be confined with a tradeoff between the VM's isolation and the application's functionality.

One application of FVM is to isolate the execution of untrusted or vulnerable programs from the host environment. With the *copy-on-write* approach, processes in a VM have read access to the host environment, but their write access is confined in the VM's workspace with resource constraint policies. As a result, the VM environment prevents *data corruption* and *service denial*, but does not prevent *data*

theft. However, we can configure a VM's initial environment as a subset of the host environment, and make certain directories containing confidential information invisible in the VM. In this way, processes running in the VM are not able to access confidential information on the host environment, thus preventing *data theft* as well.

3.7.2 Implementation Limitations

The FVM virtualization layer intercepts system calls, checks the system call argument, renames the argument if necessary, and finally invokes the original system call. As explained in Section 3.5.9, the renamed argument is stored in a buffer allocated from the calling process's address space. Therefore, the possibility exists that the renamed argument is modified again by other threads of the same process before the original system call accesses the buffer. For example, a malicious process may launches multiple threads to scan its own address space. If one thread detects a buffer with a path name that somehow contains a VM identifier, it can infer that this path name is likely to be a renamed one by the FVM layer, and then quickly rename it back to the original path name before the original system call accesses the buffer. In this way, it is possible for a malicious process to break the VM's isolation and to modify the state outside of the VM. To prevent such attacks, the FVM layer sets the page protection to *read-only* on the buffer immediately after the renamed argument is written to the buffer. Moreover, FVM intercepts the system call to change page protection (*NtProtectVirtualMemory*) and prevents a process in a VM from modifying the page protection of these buffers. The buffers holding the renamed system call argument are freed after the original system call returns to the FVM layer.

The ability to detect a VM environment by a user process is another concern. Under the current renaming approach in FVM, a process can detect if it is running in a VM by keeping creating next-level directories from any directory. The NTFS file system supports up to around 32,000 characters for a full file or directory path. If the path length of a file to be created exceeds the limit, the file creation system call fails. When a process in a VM attempts to create files, the files are actually created under the VM's root directory, which is a sub-directory of the host root directory. In a VM environment, the file creation fails if the path length of a file to be created plus the path length of the VM's root directory exceeds the character limit. As a result, a process can keep creating next-level directories to approach near 32,000 characters in the file path, and then check the file creation result to decide whether it is running on a host environment or in a VM. It can alternatively using similar methods on registry key or kernel object directories to detect a VM environment. If the process detecting a VM environment is malicious, it can temporarily hold off its malicious operations when running in a VM. Therefore, users may incorrectly

commit such a downloaded malicious program to the host environment. To resolve this vulnerability, the FVM layer checks the path length of a file, directory, registry key or kennel object before passing the renamed path to the original system call. If the sum of the path length and the VM's root path length exceeds the limit, FVM renames the path in a different way and creates the file or other objects in a separate directory immediately under the VM's root directory, ensuring that the renamed path length does not exceed the limit. FVM also needs to maintain a mapping between the original path and the new path with a configuration file.

Finally, part of the current FVM layer is still at the user-level by intercepting Win32 APIs. User-level interception can be bypassed by user processes. Although the FVM layer can verify the integrity of certain structures in a process address space, such as the IAT of the process, the code section of the FVM DLL and the code sections of other system libraries, a program can use assembly code to directly make a system call, without going through any Win32 APIs. Consequently, intercepting the corresponding kernel-level system call interface should be a more secure solution.

Chapter 4

Evaluation of FVM

A key design goal of FVM is efficient resource sharing among VMs so as to minimize VM start-up/shut-down cost and scale to a larger number of concurrent VM instances. However, the resource sharing design should not compromise isolations among VMs. In this chapter, we evaluate the strength of isolations FVM can achieve, and measure the FVM's performance overhead and scalability.

4.1 Isolation Testing

To verify that the FVM layer can indeed isolate processes in a VM from the host environment, we start a VM and run a set of programs in the VM. Then we stop and delete the VM. Ideally, after the VM is deleted, all the file and registry modification by processes in the VM should be discarded, and there should be no state changes on the host environment. We use a snapshot tool to detect any state changes between when the VM is started and when the VM is deleted.

Because most malware comes to a user's desktop through web browsers, such as the *Internet Explorer (IE)*, we choose to run IE in a VM to automatically browse a number of web sites, especially untrusted web sites. All the downloaded contents, such as images, Java script files, ActiveX controls and registry settings should be confined within the VM's workspace and discarded as the VM is deleted. In this way, even if malware has infiltrated the VM through browser vulnerabilities, configuration errors or user's ignorance, the integrity of the host environment is still preserved. To drive the IE to browse the Web automatically, we use a Web crawler to scan a set of URLs. For each processed URL, the Web crawler launches IE to render the web page. This automatic web browsing mechanism is similar to the Microsoft Strider HoneyMonkey [66], which automates the IE to mimic human web

browsing in a virtual machine. The purpose of Strider HoneyMonkey is to identify and analyze malicious web sites that exploit browser vulnerabilities to install malware.

The Web crawler we use in the isolation testing is an open-source web site copier called *WinHTTrack* [67], which can scan a number of URLs recursively and mirror the scanned sites within a local directory. We modifies the *WinHTTrack* to launch the IE on every URL to be processed. The crawler waits for 10 seconds after the IE process is created. It then terminates the IE process and goes to the next URL without saving the page to the local machine. The crawler terminates an IE process gracefully by sending the *WM_CLOSE* window message to all the windows associated with the IE process. Since the *WinHTTrack* is running in a VM, all the IE instances it launches are running in the same VM. We collect URLs of 237 untrusted web sites from McAfee's SiteAdvisor and use them as the seeds of crawling. The IE we are using is Internet Explorer 6.0, Service Pack 1. To expose vulnerabilities to malicious web sites, we manually changes the IE's security settings in the testing VM to the lowest level, which allows ActiveX controls and scripts to be executed automatically without a user's consent. To compare the file and registry snapshot before creating the VM and after deleting the VM, we use *WhatChanged for Windows* [68] from Prism Microsystems, Inc. This tool takes a snapshot by recording the full path and size of all the files, and all the registry keys and values in a log file. It can also detect the differences between two snapshots.

After browsing the test web sites for more than five hours, we stopped the crawler and examined the FVM's workspace for files and registry entries. Most of the files created under the VM's workspace were cached images and script files downloaded from the web sites. We also found new files and registry settings resulted from installing the "MyWebSearch Toolbar", which is a toolbar installed into IE and is rated as spyware by a few spyware removal tools. The new registry settings include new values under the *Run* key, which allows an executable related to the "MyWebSearch Toolbar" to be started automatically whenever the machine starts or a user logs in. Modifications to such registry keys and other Auto-Start Extensibility Points are typical actions of malware. After we stopped and deleted the VM, we ran *WhatChanged for Windows* to detect changes of the current file and registry settings from the snapshot we made before creating the VM. The result shows that one file on the host environment is changed during the automatic browsing in the VM. The file is named "Perflib_Perfdata_xxx.dat" and is located under the Windows system directory. We have verified that this file is created by the Windows performance monitor under certain circumstances, and is not related to our isolation testing. Even if we do not browse any web sites, the file is still changed between the moments when the two snapshots are made. Similarly, a few registry setting changes are verified to be "noise". Based on the testing result, we

can conclude that operations in a VM under the FVM architecture can be isolated from the host environment.

The testing VM is created with the policy that all the device access and IPCs with processes outside of the VM is blocked, with only one exception: the communication between the VM and the RPC port mapping daemon process is allowed. This is necessary for the registration/installation of OLE/ActiveX objects on Windows OS. Without this communication available, users can still browse web sites using IE, but the ActiveX control features of the web page will be missing. Enabling this communication can potentially impair the FVM's isolation. We have examined the system call log of the RPC port mapping daemon (the *svchost.exe* process using the *rpcss* DLL). Fortunately, we found that this daemon only performs communications with the *port* object, and never modifies file or registry settings on the host environment. As a result, this communication may be enabled as a default policy when creating a VM.

The approach we use in the isolation testing can be applied to identify malicious web sites. This is similar to the HoneyMonkey approach but has a higher scanning rate and detection speed. The scanning process consists of the following steps:

1. Create N VMs on a host machine;
2. The web crawler starts an IE process on a URL and associates the process with one of the VMs;
3. The IE process runs for K seconds before exits, and the FVM layer monitors system calls made by the process.
4. If FVM does not detect suspicious system calls, the web crawler assigns new IE process to the VM; otherwise, the scanned URL and the suspicious access are logged. The crawler then deletes the VM and create a new VM.

Existing web-based malware monitoring approach usually use VMware or Virtual PC to run a number of IE processes on suspicious URLs. When malicious state changes are detected in a VM, the same set of URLs must be re-tested one by one in different VMs to identify which one is the malicious site. In addition, it takes at least one minute to restart a clean VM from a saved snapshot. The maximum scan rate of HoneyMonkey is around 8,000 URLs per day per machine [66]. In contrast, FVM can support a large number of concurrent VMs (e.g., $N = 50$) on one physical machine, and it takes seconds to rollback to a clean VM. Under the same scanning condition that IE visits every URL for 2 minutes, the maximum scanning rate under FVM is 36,000 per day per machine. We will discuss the FVM's scalability in the performance measurement section, and present an FVM-based scalable web sites testing system in Chapter 5.

4.2 Performance Measurement

Because the FVM virtualization layer is implemented by intercepting selected Windows system calls, the performance overhead of FVM comes from the overhead of executing additional instructions in every intercepted system call. The total overhead can be divided into two aspects:

- *Redirection* overhead, which includes the overhead of looking up the calling process's VM identifier, allocating virtual memory, checking and renaming the name argument. In other words, it is equivalent to the total performance overhead when there are no copy-on-write operations involved.
- *Copy-on-write* overhead, which is introduced when a process in a VM opens a file or registry key for write access in the first time. In some sense, this overhead can be considered as part of the total overhead in duplicating the entire host environment in a VM, only distributed over time.

In this section, we measure the interception overhead of individual system calls, execution time overhead of command line programs, and startup time overhead of GUI programs under FVM. We compare the results with the same measurements in the native (host) environment and in a VM of VMware Workstation 5.5 running on the same host machine. We also measure a VM's resource requirement and scalability under the FVM architecture. The host machine we are using is a Dell Dimension 2400 desktop, with an Intel Pentium 4 2.8GHz CPU and 768MB RAM. The host operating system is Windows 2000 Professional, service pack 4. The VMware VM is configured with 540MB RAM, which is the maximum recommended guest RAM on the host machine. The guest OS is also Windows 2000 Professional, service pack 4, and is installed with minimum number of applications required for the performance measurement.

4.2.1 System Call Interception Overhead

To measure the system call interception overhead, we count the average CPU cycles spent in each system call when browsing a few benign web sites using the *WinHTTrack* crawler. We use the *rdtsc* instruction to read the CPU's time-stamp counter at the entrance and exit of each system call. The difference between the two cycle counts is the CPU cycles spent in a system call. In the testing, we first run the crawler for ten minutes to obtain the number of invocations and average CPU cycles of every system call. Second, we run the crawler in a VM for approximately ten minutes to count the average CPU cycles until the same number of invocations has been reached. For fair comparison, we try to minimize the cost due to file copying

NT System Calls	Invocation Count	Native (CPU cycles)	FVM (CPU cycles)	Overhead (%)
NtCreateFile	12,093	170,869	212,758	25
NtOpenFile	10,526	153,582	207,099	35
NtQueryAttributesFile	17,006	103,097	146,031	42
NtQueryDirectoryFile	3,960	394,556	503,429	28
NtQueryInformationFile	24,818	7,296	8,161	12
NtSetInformationFile	17,992	50,002	51,502	3
NtClose	193,626	8,575	8,661	1

Table 4.1: *We list seven file-related system calls, their average execution time (in CPU cycles) in native environment and in FVM environment. The times each system call are invoked in the testing are shown in the second column. The overhead of system calls is shown in the last column.*

and directory creation in the VM's workspace. Therefore, before the second testing run, we start the crawler in the same VM to browse a few webpages, then stop the crawler and clean the file cache of the Internet Explorer. In this way, a few directories and database files required by the IE's execution can be initialized in advance in the VM's workspace before we start the measurement. As mentioned in Section 3.5.1, we have implemented an optimization to the file virtualization in the FVM layer by keeping all the names of a VM's private files in kernel memory. This optimization saves the cost of making an additional system call to check the file's existence in the VM's workspace. We enable this optimization in this experiment.

Table 4.1 shows seven file-related system calls, their average CPU cycles in our testing machine and their overhead under FVM. System calls operating on files usually take more CPU time to complete, and thus play more important roles in runtime performance than other types of system calls. The seven system calls in Table 4.1 are also the most frequently invoked file-related system calls intercepted by FVM.

As shown in Table 4.1, the *NtQueryAttributesFile* and *NtOpenFile* system calls have larger overhead than other system calls. This is inconsistent with our original expectation, which was that only the *NtQueryDirectoryFile* system call has large overhead due to the expensive directory union operation. When looking at the experiment result of *NtCreateFile*, *NtOpenFile* and *NtQueryAttributesFile*, we found that all of the three system calls cost around 45,000 more CPU cycles in FVM environment than in native environment. Further study shows that such overhead mainly comes from the current implementation of constructing renamed system call arguments, which must be stored in buffers allocated from user space. In the current

FVM prototype, we allocate such buffers on demand inside every intercepted system call and free it at the exit of the system call. This repeated memory allocation using *ZwAllocateVirtualMemory* cost more than 20,000 CPU cycles on average in each system call. A simple solution to this problem is to allocate a large memory buffer, e.g., one page, for each new process. This approach allows all the system calls made by the process to use the same buffer without frequent allocations. However, the Windows NT kernel is a preemptible kernel that uses threads as the basic scheduling unit. Concurrent threads of the same process may have conflict when they require access to the common buffer inside their system calls. Synchronizing the concurrent accesses decreases the responsiveness of the preemptible kernel. Therefore, such memory allocations should be associated with individual threads instead of processes. With this optimization in place, we can reduce most of the system calls' interception overhead to less than 20%.

In fact, moderate overhead of a few intercepted system calls usually do not have significant impact on an application's overall performance. This is because these system calls are only a small portion of all the system calls a process invokes at the execution time. Many system calls with high invocation frequency have neglectable (such as *NtClose*) or zero (not intercepted by FVM) overhead.

4.2.2 Application Execution and Startup Overhead

To evaluate an application's overall performance under FVM, we measure the *execution time* of three command line programs, and the *startup time* of four GUI programs. The execution time is the average elapsed time from when a program is started to when it terminates, and the startup time is the average elapsed time from when the application is started to when it finishes initialization and is waiting for user input. We use a set of simple benchmarks for the measurement, as follows:

1. Run WinZip 8.1 to compress the Borland C++ 5.5.1 installation directory (51MB);
2. Run XCopy to Copy the Borland C++ 5.5.1 installation directory (51MB);
3. Run Borland C++ 5.5.1 to compile its installed examples (127 C++ files, 462KB);
4. Start Microsoft Word 2000 to open a 668KB .doc file;
5. Start Adobe Acrobat Reader 8.0 to open a 5.9MB .pdf file;
6. Start Microsoft Excel 2000 to open a 214KB .xls file;

Command Line Program	Native (msec)	FVM (Opt) (msec)	FVM (msec)	VMware Workstation (msec)
Winzip32.exe	4,899(0%)	5,634(15%)	5,928(21%)	6,369(30%)
xCopy.exe	6,340(0%)	6,784(07%)	7,259(14%)	10,645(68%)
BCC32.exe	24,548(0%)	27,382(12%)	30,021(22%)	34,823(42%)

Table 4.2: *We list the execution time of three command line testing programs in native environment, in FVM environment, in FVM environment without optimization and in a VMware VM. With optimized file virtualization, the overhead in FVM environment is typically less than 15%.*

7. Start Microsoft PowerPoint 2000 to open a 2.7MB .ppt file;

To measure the execution time or startup time, we use a control program to run the above benchmarks with the *CreateProcess* API. This control program then invokes the *WaitforSingleObject* or the *WaitforInputIdle* API to recognize the moment when the testing process terminates or finishes initialization. The elapsed time since the process creation to this moment is calculated using the Windows *high-resolution performance counter*, averaged by twenty consecutive testing runs in the native (host) environment, under FVM and in a VMware Workstation VM.

Table 4.2 shows the execution time overhead of three command line program benchmarks running in the FVM environment, and in a VMware Workstation VM. Without optimization in file virtualization, the execution time overhead is up to 22%, as shown in the compilation benchmark. The overhead is reduced to 12% when the optimization is in place. For other benchmarks, the execution time overhead is typically below 15% with optimization. As a result, the optimized file virtualization sometimes can greatly improve an application's runtime performance, especially when the application is I/O intensive. In contrast, the execution time overhead in a VMware VM is much higher than the FVM environment. In the xCopy benchmark, the overhead in a VMware VM is as large as 68%.

Figure 4.1 shows the startup time of four GUI program benchmarks when they are running natively and running in the optimized FVM environment. The benchmark results indicate that the startup time overhead under FVM is typically below 15%. Figure 4.1 can also explain the *redirection* overhead and the *copy-on-write* overhead in the FVM layer. We define the *initial startup time* as the startup time when a GUI program runs the first time after the host machine reboots, and the *average startup time* as the startup time on average when the program runs since the second time onwards. These two values have the following attributes: (1) The initial

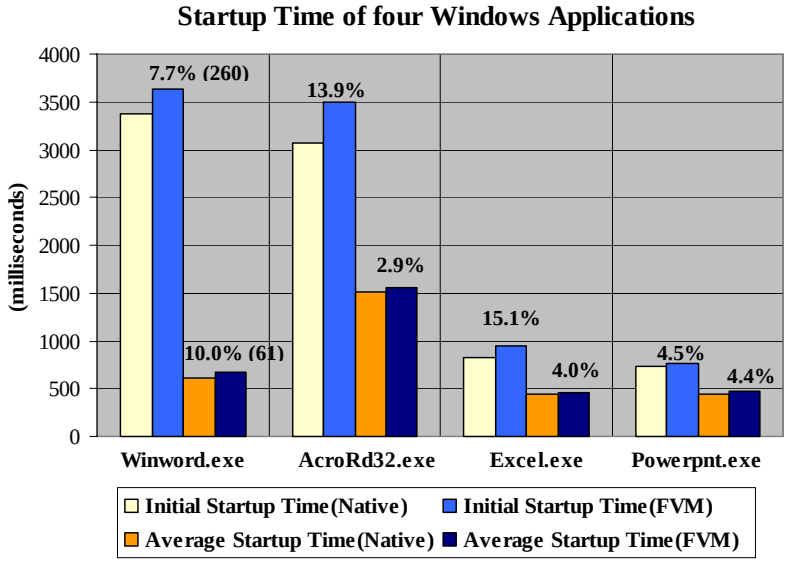

Figure 4.1: The initial startup time and the average startup time of four GUI Windows applications when they are executed natively and under FVM. The overhead of startup time under FVM is typically below 15%.

startup time is larger than the average startup time, no matter whether the tested program runs natively or under the FVM, because the process spends time in building up its working set and caching file data at the first run. (2) The initial startup time is larger when the program runs under FVM than it runs natively. The difference comes from the total of the *redirection* overhead and the *copy-on-write* overhead. (3) The average startup time is larger when the program runs under FVM than it runs natively. The difference comes from the *redirection* overhead only. Therefore, we can roughly estimate *redirection* overhead and the *copy-on-write* overhead for a tested program under FVM. For example, when starting Microsoft Word to open a 668KB .doc file for the first time, as shown in Figure 4.1, the total virtualization overhead is 260 msec, 61 msec of which belongs to the *redirection* overhead, and the rest 199 msec can be attributed to the *copy-on-write* overhead.

4.2.3 Resource Requirement and Scalability

A main advantage of FVM over traditional hardware-level virtualization techniques is the small resource requirement in each VM. Because FVM shares files and registries with the host environment by *copy-on-write*, the disk space and registry

memory requirement in a VM can be very small. In one testing, the *WinHTTrack* keeps invoking IE to browse pages on 50 web sites for more than 4 hours in a VM context. In the end of the testing, the VM's private files consumes a total of 76.7MB disk space, most of which are used by images, cookies, Java scripts and other downloaded files. For Windows registry, the VM's private registry settings increases the size of the physical machine's registry from 18.5MB to 21.1MB, thus requiring only 2.6MB additional memory.

The memory usage in the FVM layer is also small. The FVM layer uses kernel memory to store the associations between processes and VMs, and to store the delete logs for files and registries. To optimize the file virtualization, FVM keeps the names of files that have been copied to a VM's workspace in the kernel memory. In the above crawling testing, it takes about 240KB (or 60 pages) kernel memory to maintain the process-VM association, delete log and the file name tree. The size of the file name tree depends on the number of files and the length of the file names under a VM's workspace. Because these names are stored in a binary tree, and because each node of the tree only stores a relative name in the full file path, the memory usage for this data structure is not significant. In an extreme case, for example, if a VM is created without disk space restriction, and a process in the VM attempts to write every file on the *C* drive of the testing machine, which contains 90,000 files and 7,500 directories that consume 12.7GB disk space, the size of the file name tree due to all the copying is only about 3.2MB. When the size grows beyond certain threshold (5MB for example) under unusual circumstances, the FVM layer can use memory-mapped file to store the file name tree, or switch the file virtualization back to its un-optimized mode without keeping a file name tree in memory.

Given the small resource requirement, we expect high scalability of VMs under the FVM architecture. The scalability of FVM is mainly affected by the kernel memory usage. On the 32-bit Windows OS, kernel memory space normally occupies the upper 2GB[1] of the entire 4GB address space. Kernel memory requested by device drivers or OS kernel components is allocated from pageable system heap (called *paged pool*) or resident system heap (called *nonpaged pool*). The maximum size of the two heaps are determined by the Windows OS at the boot time. The FVM virtualization layer requests kernel memory from the paged pool. Therefore, FVM's scalability relies on the size of the available paged pool. However, Windows 2000 has a limitation in its registry implementation: the registry data is stored using the paged pool after system boots, and new registry data requires allocating additional

[1]The user address space can occupy the lower 3GB with a boot-time option, leaving only 1GB to kernel space.

memory from the paged pool. Since each VM's registry space is in fact under a subkey of the host registry, each created VM will increase the host registry size when processes in a VM make write access to the original host registries, thus consuming more memory from the paged pool. This limitation compromises the FVM's scalability. In our testing machine, the maximum paged pool is 164MB, 25MB of which is occupied by other system components. Before creating any VM, the available paged pool is below 139MB. If we reserve 5MB memory from the paged pool for each VM to store its private registry and state information, and assume no other usage to the paged pool, the FVM layer can support up to 27 concurrent VMs. However, this estimation is too optimistic because other system components and data structures are also consumers of the paged pool. In order to resolve the registry bottleneck and to improve the scalability, we can implement our own registry manipulation mechanism that operates the registry data in a memory-mapped file instead of in the paged pool. Fortunately, Windows XP and later version of Windows already resolve this registry limitation. With the same amount of available paged pool, we expect FVM to support more than 50 concurrent VMs in most cases on 32-bit Windows XP or Windows Server 2003.

The scalability of FVM is also limited by the number of processes running in each VM, and by the memory usage of each process in a VM. This is because each process has a number of data structures stored in the kernel memory using the paged pool or nonpaged pool. When more processes are running in each VM, more kernel memory is consumed and thus the scalability of FVM is decreased. In our Windows 2000 test machine with around 20 Windows system processes running, up to 60 additional IE processes can be started concurrently without thrashing. The FVM management console verifies the availability of kernel memory and disallows more VMs to be created when the size of available memory is below 30MB.

Overall, all the performance measurement results demonstrate that the FVM virtualization layer provides *feather-weight* OS-level VMs with small execution time overhead, low resource requirement and high scalability. Because of the *copy-on-write* mechanism in the FVM layer, VM manipulation operations such creating, starting, stopping and deleting a VM can be completed in seconds. As a result, the FVM architecture can serve as an effective platform to support applications that require frequent invocation and termination of a large number of "playground" VMs. These VMs can be used to analyze malware and malicious web sites on the internet, try out untrusted mobile code, execute vulnerable networked applications, or access confidential information that should be isolated from the host environment. We discuss a few applications on the basic FVM framework in Chapter 5.

Chapter 5

Applications of FVM

OS-level virtual machines powered by the FVM framework make an effective computing platform for many useful applications on Windows server and desktop environment, such as malware sandboxing, intrusion tolerance and analysis, access control, and others that require an isolated but realistic execution environment. In this chapter, we present design and implementation of the following five applications on the FVM framework:

- Secure mobile code execution service to protect desktop integrity;
- Vulnerability assessment support engine for safe vulnerability scans;
- Scalable web site testing to identify malicious sites;
- Shared binary service for application deployment;
- Distributed *Display-Only File Server* (DOFS) for document protections.

We describe how to customize the generic FVM framework to accommodate the needs of the five applications, and present experimental results that demonstrate their performance and effectiveness.

5.1 Secure Mobile Code Execution Service

5.1.1 Introduction

Mobile code refers to programs that come into an end user's computer over the network, and start to execute with or without the user's knowledge or consent. Examples of mobile code include JavaScript, VBScript, Java Applets, ActiveX Controls, macros embedded in Office documents, etc. Because mobile code typically

runs in the security context of the user who downloads it, it can modify system state with the user's privilege, and thus can compromise the system when it is malicious. Malicious mobile code [69], such as *viruses*, *worms* and *Trojan horses*, can cause data corrupt, data theft or service denial, and thus place a significant threat to system security.

A traditional solution against malicious mobile code is *virus signature scanning*, which scans file system and memory for bytes that match the signatures of existing malicious code. Once the malicious code is detected, it is quarantined or removed by the Antivirus program. This technique has been widely used for the past ten years, but it is not sufficient by itself because it is unable to recognize zero-day malicious code before victims report it and signatures for the code are generated. Another approach is *behavior blocking* [70], which sandboxes the execution of untrusted applications by monitoring system calls according to pre-defined security policy. System calls that violate the policy are denied to proceed. However, it is difficult to set up accurate security policies that block all attacks without breaking legitimate applications. To derive an accurate application-specific system call model, comprehensive program analysis on an application's source code [71] or binary code [72] is required.

A third approach is to separate untrusted mobile code and confine their application logic in remote machines, which are disposable because they are not supposed to have any sensitive or critical data or resources. The Spout project [73] transparently separates incoming Java Applet into application logic component and GUI component with a proxy Web server, and execute the untrusted application logic component on a specific Java Application Server outside the firewall. Similarly, the secure mobile code execution service [74] separates mobile code that comes into an end user machine as an email attachment or a web document downloaded through an anchor link. It uses a POP3 proxy server or a Win32 API interception technique to separate untrusted email attachments, and execute them on a remote Windows Terminal Server machine. The GUI of mobile code execution is displayed remotely on the end user machine using terminal services. The advantage of these approaches are *physical isolation*, which ensures that malicious mobile code cannot compromise the end user machine once separated. However, there is no generic approach to separate different forms of mobile code and to force their execution inside a remote "guinea pig" machine.

WindowBox [75] presents a simple security model that enables end users to use multiple desktops on the same physical machine. Each desktop is separated from other desktops in that objects created in one desktop are inaccessible from other desktops. WindowBox modifies the NT kernel to tag an object's security descriptor with the security identifer of the owner desktop, and to deny access to an object if the request comes from a different desktop. This isolation idea is similar to the

FVM idea except that WindowBox uses access checks instead of virtualization to separate different execution environments.

Different from the above approaches, the FVM framework supports an *intrusion-tolerance* solution against malicious mobile code by confining untrusted code and its applications in a VM. Vulnerable networked applications, such as web browsers and email clients, and untrusted content coming through these applications, are always executed in the context of one or multiple VMs. The VM environment has small performance overhead, and is designed to be un-intrusive to most applications running in the VM. Users can start or stop a number of VMs, and start untrusted applications in the VMs in seconds. Any persist state changes in the VM, such as file modification and registry update, are confined within the VM's workspace regardless of being legitimate or malicious. In addition, such confined state changes can be selectively committed to the host environment. To hide confidential files on the host environment from untrusted mobile code in a VM, the default file system image visible to the VM can be configured to a subset of the file system image of the host environment.

One advantage of running untrusted mobile code in FVM is the fast clean and repair. When malicious or undesirable behaviors are detected in a VM, stopping the VM and then deleting it are all that have to be done to clean the system. Another advantage is that, the execution of mobile code in a VM is not subject to access control policies or least-privilege restrictions. Therefore, legitimate mobile code can be executed smoothly without being disrupted from time to time. Finally, the system call logging and analysis module in the FVM layer can detect suspicious behaviors of processes running in a VM, and thus makes an effective platform for malware analysis. There have been a few systems that leverage virtualization technologies to resolve the malicious mobile code problem, such as GreenBorder [35], which uses OS-level VMs to sandbox web browsers and email clients, and Tahoma [76], which constructs a safe *Browser Operating System* (BOS) by using Xen VMs to isolate untrusted Web applications from user's local resources.

5.1.2 Design and Implementation

Mobile code enters a user's desktop through networked applications in different forms. Some mobile code is embedded in email attachments and file downloads. Such mobile code can be separated from the email client and web browser applications. Therefore we can directly open the file containing the mobile code in a VM. However, other mobile code, such as an ActiveX control or a *Browser Helper Object (BHO)*, cannot be separated from the web browser or email client process. Instead, they are executed in the same address space of the web browser or email client process. As a result, not only separated email attachments and downloaded

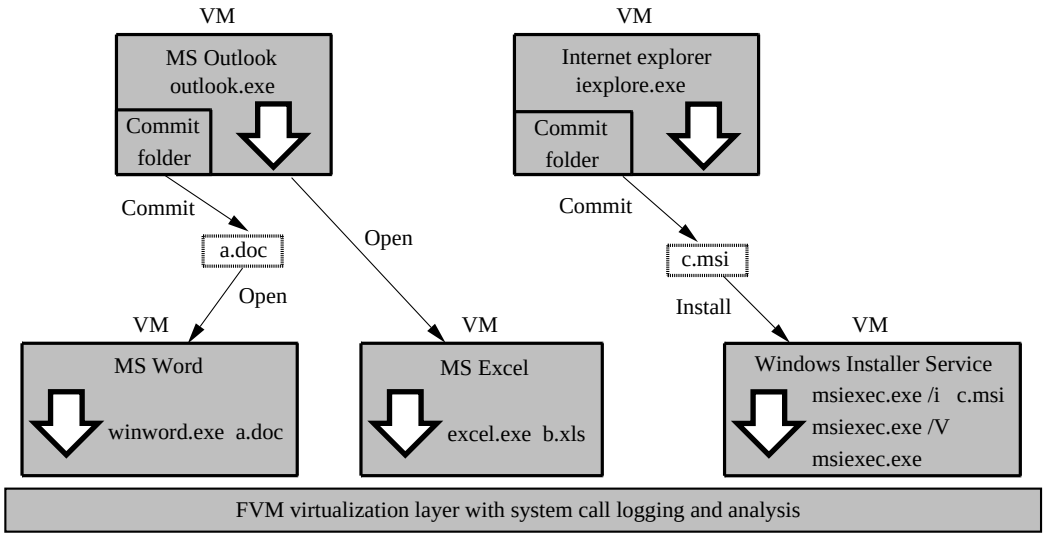

Figure 5.1: An FVM-based secure execution environment for untrusted mobile code coming through MS Outlook and Internet Explorer. In this example, a.doc and b.xls are email attachments, and c.msi is an installer package downloaded from the Internet.

files, but all the web browser and email client applications should be opened or executed in VMs.

Implementing a secure execution environment for untrusted mobile code is straightforward under the FVM architecture, which provides an execution environment that is isolated from the host environment. The only additional work is the following:

- Automatically start a new VM to execute web browser or email client programs, such as Internet Explorer and MS Outlook, when users invoke such programs on their desktops.
- Installer packages (.exe or .msi) downloaded into a VM can be installed in the same VM, by running the installer executable or invoking the Windows Installer Service to extract the msi package.
- When users commit executable or document files created in the VM to the host environment, FVM marks these files as "untrusted". When users later on open a marked file from the host environment, a new VM is started to execute the process that runs the file.

Figure 5.1 shows a secure execution environment for untrusted mobile code

under the FVM architecture. The Internet Explorer (IE) and MS Outlook are associated with two VMs. The FVM virtualization layer is modified to monitor the process creation and termination of IE and Outlook. When any of the two processes is created, its associated VM is started; when the process is terminated, the associated VM is stopped. By default, file and registry updates within the VMs are not committed to the host environment when the VMs are stopped. However, if users explicitly save an email attachment or download file into a special folder in the VM's workspace, called *commit folder*, the file (e.g., a.doc or c.msi in Figure 5.1) will be committed when the VMs are stopped. In addition, the Outlook's personal folder files (.pst files) are also committed because they store all the received emails, address book and other important Outlook settings.

Once committed, a file is moved from the VM's workspace to the host environment. However, there is no guarantee that this file is free of malicious mobile code. Therefore, the secure execution environment tags every committed file a special *mark* which indicates that the file comes from network and may contain malicious code. For example, a.doc and c.msi are marked in Figure 5.1 after they are committed. When users open a.doc from the host environment, the FVM layer recognizes its mark and forces it to be opened by MS Word in a new VM. Similarly, when the installer package file c.msi is executed, the FVM layer forces the Windows Installer Service process to start in another VM to extract the package and perform the installation. We use a named NTFS alternate stream to implement the special mark on committed files.

When an msi program is installed inside a VM using the Windows Installer service, it involves modifications from three processes: a client process initiated by the user (msiexec.exe invoked with /i flag), the installer service process (msiexec.exe invoked with the /V flag) that runs on the background as a daemon, and a process spawned by the installer service process that runs under the context of the requesting user (msiexec.exe). The installer service process modifies all the system-related resources whereas its child process is spawned to handle modifications to user-related resources. FVM forces all the three processes to run inside the VM so it can correctly isolate all state changes resulted from the program installation. After the installation is completed, the hosting VM for this program is not destroyed. Instead, it continues to serve as the execution environment for the installed program.

Although the current secure execution environment only monitors untrusted mobile code coming through IE and Outlook, it can be easily extended to support other web browsers and email clients, as well as other networked applications such as FTP clients and Instant Messaging tools. In addition, the FVM environment logs system calls made by untrusted process running in a VM, in a way similar to the Strace for NT [62]. By monitoring certain Auto-Start Extensibility Points [53], and analyzing system call sequences to derive high-level behaviors, FVM is able

to detect potentially malicious process behaviors, and suggests users to selectively commit only the safe transactions in the VM.

5.1.3 Evaluation

Isolating untrusted mobile code in a light-weight virtual execution environment is our initial motivation to develop the FVM framework. Creating the secure execution environment based on FVM is straightforward, and does not introduce additional complexity and overhead. Therefore, we can refer to Chapter 4 for the evaluation result on FVM's isolation strength, performance overhead, resource requirement and scalability. In the section of "scalable web site testing" in Chapter 5, we also demonstrate the FVM's ability to identify malicious behaviors due to compromised web browsers.

5.2 Vulnerability Assessment Support Engine (VASE)

Vulnerability Assessment (VA) is the process of identifying known vulnerabilities due to programming errors or configuration errors in computer systems and networks. Different from *Intrusion Prevention System (IPS)* and *Antivirus* software, VA allows system administrators to proactively detect security holes before they are exploited. However, a VA scanner can be intrusive to the network applications being scanned. Therefore, it is not completely safe to scan the production-mode network services frequently. Because FVM can provide a virtual environment that are identical to the host environment, a VA scanner can instead scans the network services in the virtual environment, without worrying about undesirable side effects on the host environment. In this chapter, we present the design and implementation of a *VA supporting system* based on the FVM framework.

5.2.1 Introduction

A VA scanner is a program that scans specified network server applications to automatically identify vulnerabilities. It communicates with the network server machine being scanned by probing for open ports, checking patch levels, examining registry settings, or simulating real attacks. After a scan is completed, the VA scanner generates a vulnerability report, which suggests system administrator to patch the applications or to change the configurations. A well-known problem with the VA scanners is their safety. In one study that includes 11 VA scanners, all the scanners caused adverse effects on the scanned network servers. One of the scanners

crashed at least five servers during an assessment run. According to Nessus documentation [77], every existing network-based vulnerability scanner has the risk of crashing the systems and services being scanned. To resolve this safety problem, most VA scanners provide "safe scan" or "non-intrusive check" options to avoid potential denial of service. However, such options sacrifices the scanning accuracy. Even worse, it turns out that "safe scan" may not be truly safe, as reported in [78].

In fact, it is not surprising that many VA scanners are intrusive, because VA scanning packets should behave like real attacks in order to identify any vulnerabilities. For example, some protocol implementations in the scanned applications do not handle errors well, and thus can crash the process on certain unexpected inputs. Moreover, if the vulnerability is due to memory errors, such as the buffer overflow vulnerability, a scanner has the chance to overflow the buffer, which leads to unpredictable program execution, such as program crash or undesirable state change.

To obtain accurate VA results using VA scanners without possibly crashing the production-mode network services or changing the system state, we can duplicate the network service applications in a VM and scan the VM environment instead. In this way, even if the duplicated applications are crashed or cause unexpected state changes during the VA scanning, the original production-mode network services are still functioning. The VM used by VA scanners should have the following attributes:

- High testing fidelity. The duplicated network applications in the VM should be as close as possible to the original applications in the production-mode environment. New patches or configuration changes should be easily synchronized to the VM environment. This ensures that running a VA scanner against a VM can correctly identify the vulnerabilities of the production-mode environment.

- Small overhead and resource requirement.

- Fast duplication. It is necessary to run VA scanners frequently to identify new vulnerabilities in time. Therefore, duplicating the production-mode network applications into a VM should be performed very fast.

The FVM virtualization layer provides OS-level VMs with the above attributes. Therefore, we developed a VA supporting system that can perform automatic and safe VA scanning based on the FVM framework.

5.2.2 Design and Implementation

The VA supporting system is called *Vulnerability Assessment Support Engine (VASE)*. Unlike Nessus and other VA scanners, VASE does not send out probe

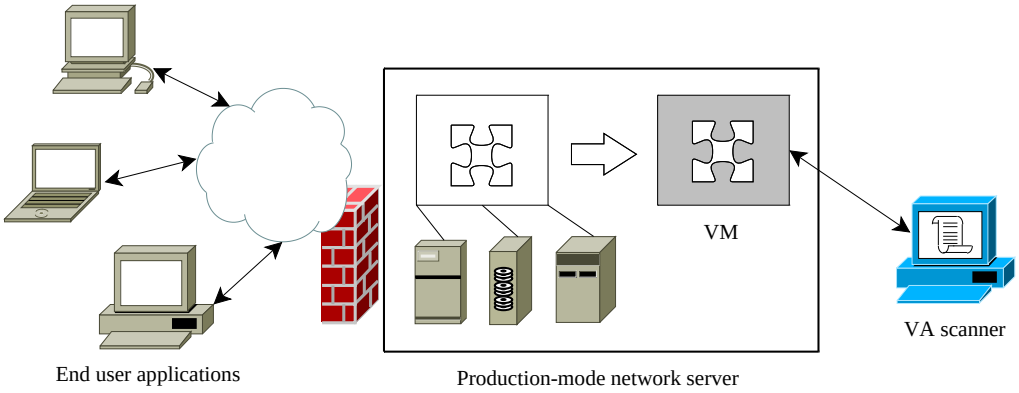

Figure 5.2: The architecture of VASE. It enables automatic and safe vulnerability assessment by running VA scanners against a VM environment that is identical to the production environment.

packets by itself to identify vulnerabilities in existing network servers. Instead, VASE is designed to automatically drive a VA scanner to scan the duplicated network service applications in a VM. The duplicated applications are identical to the production-mode applications on the host environment in every aspect. Furthermore, the scanning process does not change states on the host environment.

The system architecture of VASE is shown in Figure 5.2. To automate the vulnerability scanning process, VASE needs to create a VM on the production environment, and to start all network applications currently running on the production environment in the new VM in exactly the same way as they were started originally.

A vulnerability scanning process under VASE includes four phases. In the first phase, VASE enumerates all the network applications running on the production environment. VASE uses the *fport* tool from Foundstone to find out all the processes listening on a network port, and to obtain the full pathname of each process's executable image. These processes are usually running as daemon processes, which are controlled by the *Service Control Manager (SCM)*. VASE queries the SCM's database in the registry to decide if a server process is a daemon process. If the process image name is found in the database, VASE treats this process as a daemon process, and queries its configurations in the database. At the end of this enumeration phase, VASE generates a complete list of names and configurations of the network applications running on the production environment.

In the second phase, VASE creates a new VM and starts a VASE control process in the VM. The VASE control process installs and starts all the network applications identified in the first phase. These network applications run in the same VM context. In the third phase, VASE invokes a VA scanner to scan the VM where all the

duplicated network applications are running. The VA scanner generates a VA report after the scanning is completed. The final phase is cleanup. After the scanning process is completed, VASE stops all the processes in the VM, uninstalls daemon configurations associated with the VM, and finally terminates the VM.

5.2.3 Evaluation

We have developed a VASE prototype and measured its performance overhead, scanning fidelity and isolation effectiveness. We used three machines in the experiments, each of which has an AMD Athlon XP 2000+ CPU with 512MB memory. The first machine acts as a production-mode server. It runs Microsoft Windows 2000 server with service pack 4, Internet Information Service (IIS) 5.0, Microsoft SQL server 2000, MySQL server 4.1, and Apache web server 2.0.54. The second machine runs Redhat Linux 9.0 and Nessus 2.2.4, and serves as a VA scanner machine. The third machine is only used for performance measurement. It runs Redhat Linux 9.0, Apache benchmark for Apache web server, and Super Smack 1.3[79] used to benchmark MySQL server.

Performance Overhead

To evaluate the performance overhead of VASE, we collected the following four performance measurements on three test applications.

1. The performance of the test applications when they are running on the host environment. Under this condition, the virtualization overhead of FVM does not exist.

2. The same performance measurement as 1. except that a duplicated test application runs in a VM on the same physical machine. This measurement evaluates the impact of FVM's virtualization logic on the production-mode network applications.

3. The same performance measurement as 2. except that the duplicated test applications are being scanned by the Nessus scanner. This measurement evaluates the additional performance cost to the production-mode network applications when their duplicates in a VM are being scanned by VA scanners.

4. The performance of the duplicated applications running in a VM. This measurement evaluates the FVM's virtualization overhead.

The three test applications are Apache web server, MySQL server and WinRAR. We use Apache bench to benchmark the Apache web server. Apache bench

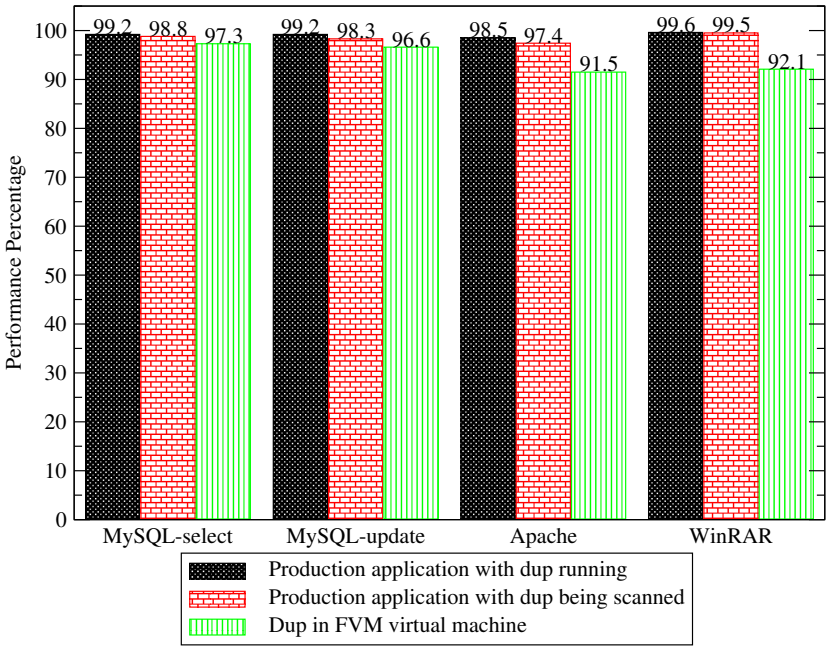

Figure 5.3: The performance overhead of virtualization and vulnerability assessment under VASE architecture. The performance impact on the production-mode network applications is less than 3%.

sets up 5 concurrent connections and sends 5,000 requests for the index.html in the document root directory. The performance metric is the number of requests served per second. We use Super Smack to benchmark the MySQL server. Super Smack sets up 10 concurrent connections and 10,000 requests for select operation (`select-key.smack` scripts) and update operation (`update-select.smack` scripts). The performance metric is the number of select or update operations served per second. Finally we measured WinRAR 3.42 to evaluate the performance impact of FVM on non-network applications. The performance metric for WinRAR is the number of seconds required to extract the Apache web server 2.0.54 package.

Figure 5.3 is the normalized performance of each test application under test case 2, 3 and 4, with respect to that under case 1. For example, 95% means that the performance is 95% of that when the application runs on the host environment without FVM. The performance result indicates that just running a duplicated application in a VM or even performing a VA scanning against such a duplicate has little impact on the performance of the production-mode network applications. The performance penalty is typically less than 3%. This result demonstrates that VASE's approach

of "duplicate and test" actually allows a network application to continue its service while being scanned. As for the FVM's virtualization overhead, it is negligible for the MySQL server, and about 9% for Apache and WinRAR.

Scanning Fidelity

To demonstrate that scanning a duplicated network application in a VM is as real as scanning the application running on the host environment directly, we start a few network applications, including Windows telnet service, MySQL server, Apache web server, eDonkey 2000 and KaZaA on the server machine, and then launch Nessus against each of them to generate a VA report. Among the tested applications, Nessus found a security hole in KaZaA, and a security warning in both telnet service and Apache web server. After that we duplicate all the applications into a VM and scan each of duplicated applications. We found the vulnerability reports from the two experiments are identical. Consequently, scanning duplicated applications running in a VM instead of the original production-mode applications has high fidelity on the vulnerability assessment result.

Isolation

To verify that all the file and registry changes during the vulnerability scanning are confined in the VM's workspace, we use snapshot tools similar to the isolation testing described in Chapter 4 to find out the differences before and after a vulnerability scanning. The result is that all the side effects on file system and registry state are successfully confined within the test VM's workspace. In addition, we found that vulnerability scanning indeed leaves a number of side effects to the file system state. For example, during one vulnerability scanning, the Apache web server changes its access log that records all the HTTP requests from Nessus, the MySQL server changes its database access log, and the eDonkey generates a number of .met files. Without VASE, these side effects, and possibly even worse side effects, may have affected the host machine directly.

5.3 Scalable Web Site Testing

5.3.1 Introduction

Malicious web sites on the Internet can exploit a web browser's vulnerability to install malware on a user's machine without user consent or knowledge, thus posing a serious threat to the web users. Some malicious sites use code obfuscation techniques which make signature-based detection difficult. A proactive approach

to protect users from these malicious sites is to periodically scan untrusted web sites and test their safety by monitoring the browser's runtime behavior. Once malicious web pages are identified, proper actions can be taken to discourage users from accessing them, such as blocking malicious URLs or shutting down their Internet connectivity.

A web site testing system automatically drives web browsers to visit a collection of untrusted sites and monitors any suspicious state changes on the testing machines. Because visiting malicious sites may cause permanent damage on a testing machine, each testing web browser should run on a separate virtual machine to isolate its side effects from the physical host machine. Whenever malicious state modifications are detected, or a certain number of URLs have been tested in a VM, the VM is terminated and a new VM is started to host the browser testing for other URLs.

The Strider HoneyMonkeys [66] is a large-scale web site testing system designed to identify malicious web sites that exploit browser vulnerabilities. It consists of a pipeline of "monkey programs" running Internet Explorer browsers on virtual machines. From a given input URL list, it drives browsers to visit each URL and detects browser exploits by monitoring modifications to file/registry state outside the browser sandbox. Upon detecting an exploit, the system destroys the infected VM and restarts a clean VM to continue the test. A crawler-based study of spyware [80] uses a similar testing system to detect spyware-infected executables on the web. In addition, the study also analyzes drive-by downloads that mount browser exploit attacks, and tests them with Internet Explorer as well as Firefox. A group in Google performed another analysis of web-based malware [81] using a similar approach. The SpyProxy [82] intercepts HTTP requests using an HTTP proxy, fetches the target web pages into a local cache, and checks the safety of the page before passing it to the client browser for rendering. Their malware detection procedure consists of both a static analysis of the web content and its dynamic analysis inside a virtual machine.

Given a large number of URLs to be tested, a web site testing system should launch a testing browser in a large number of concurrent VMs to improve scanning efficiency. In addition, the overhead of terminating and starting a VM should be very small because it is a frequent operation in web site testing. Existing web site testing systems [66, 80, 81, 82] use hardware abstraction layer (HAL)-based VM technology such as VMware and Xen, which is heavy-weight in that each VM runs its own instance of operating system. Typically, these testing systems deploy a cluster of physical machines each running a small number of VMs. The relatively high overhead of terminating and starting a HAL-based VM has seriously limit the overall web site scanning throughput. In contrast, FVM is designed to be light-weight in that concurrent VMs share most of the system resources, including the

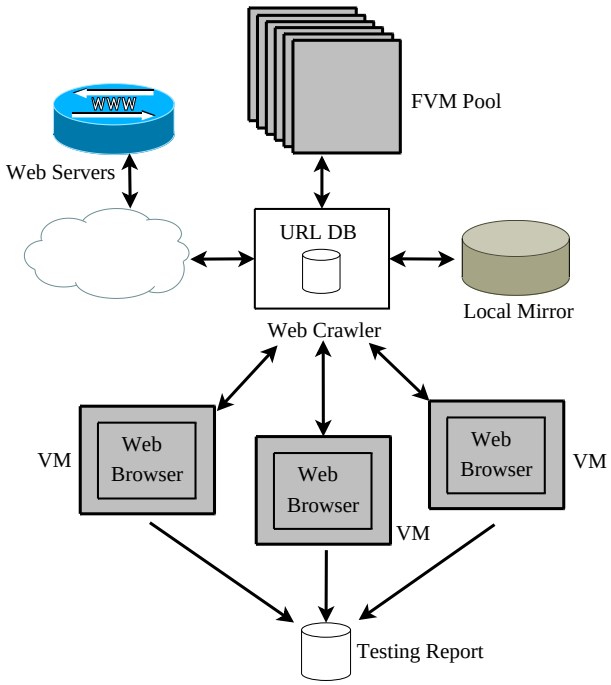

Figure 5.4: System architecture of the FVM-based web site testing system.

OS and shared libraries. As a result, FVM makes it possible to terminate and start a VM within seconds, thus significantly improving the throughput and scalability of a web site testing system.

To test a collection of untrusted web sites, we launch a separate web browser instance to access each such site. Each browser instance is created inside a separate VM, which can host one or multiple browser instances. To maximize the testing throughput, we create as many concurrent browser instances as possible on as many VMs as is allowed on a single physical machine. By monitoring the copy-on-write files and registry keys in a VM and the browser process's state, we can reliably detect all suspicious changes to a VM's state.

5.3.2 Design and Implementation

The system architecture of our web site testing system is shown in Figure 5.4. The input is a collection of URLs for untrusted web sites, and the output is a report of malicious URLs, and their suspicious state changes. We modified the same *WinHTTrack* [67] web crawler that has been used in the FVM's isolation testing. The crawler can scan remote sites directly, or mirror remote sites to a local machine

first and then scan them. Instead of running the web crawler directly in a testing VM, as in the isolation testing experiment, we now run the web crawler on the host environment. When crawling across the given URL list, it launches a separate *Internet Explorer* (IE) instance to visit each URL, associates the IE instance with an available VM in the FVM pool, and starts a worker thread to monitor the execution of the IE instance. Once the worker thread is initialized, the web crawler moves on to visit the next URL.

Each VM in FVM can host up to N IE instances each of which processes a separate URL. Therefore, the web crawler populates N IE instances within a VM before starting another new VM. To maximize the web site testing throughput, we start as many concurrent VMs as possible on a physical machine, as long as the total number of running IE instances across all VMs is below a threshold, which is determined by the kernel memory usage of the host machine.

To detect malicious state modification due to a visited URL, we need to know when the monitored IE instance completes rendering of the URL's page. We use a *Browser Helper Object* (BHO) to detect the completion of page rendering. A BHO is a DLL loaded into the address space of every IE instance as a plug-in. The BHO captures an IE instance's *DocumentComplete* event, and then signals the monitoring worker thread outside of the VM with a named *event*. We have modified the FVM driver to explicitly allow such a communication between every IE instance running inside VMs and the worker threads in the web crawler running outside VMs. Because malicious web content may carry a "time bomb" that triggers an actual attack at a later time, to detect such attacks, the worker thread can be set to wait for an additional T seconds after the page rendering is completed [66, 80].

After a downloaded page's rendering is completed and the associated waiting time expires, the worker thread terminates the IE instance by sending a *WM_CLOSE* message to all the windows owned by the IE instance. This message causes all the receiving windows to be closed, including all the pop-up windows. If the terminated IE instance is the last one running in a VM, the worker thread stops the VM, searches the VM's state for suspicious modifications, and finally deletes the VM.

The analysis module in our web site testing system monitors file/registry updates and process states in a VM environment to detect if any visited pages compromise the VM through browser exploits. For file updates, we monitor any executable files created outside of the IE's working directory, such as the cache and cookie directory. For registry updates, we monitor sensitive registry locations related to *Auto-Start Extensibility Points (ASEP)* [53]. ASEP is a set of OS and application configurations that enable auto-start of application programs without explicit user invocation. It is the common targets of infection for most malware programs. For process states, we monitor the creation of unknown processes and unexpected crash of IE instances. Because each IE instance is not asked to download and install any

software, or allowed to install plug-ins automatically, the above detection rules are able to recognize most browser exploit attacks.

Two modifications were made to the basic FVM framework to facilitate the development of the proposed web site testing system. First, several VM manipulation functions are made directly available to privileged user processes, including starting, stopping and deleting a VM, associating a new process with a VM, etc. To associate a new browser instance with a VM, we create the browser process in the *suspended* state, set up the association between the process and the VM, and then resume the browser process. Second, each VM is granted network access to the tested sites, and is configured to inform a process outside the VM when a certain event, such as when the rendering of a downloaded web page is completed, occurs.

5.3.3 Evaluation

We evaluate the effectiveness and testing throughput of our web site testing system by crawling real-world untrusted web sites. The testing system runs on a Dell Dimension 2400 with an Intel Pentium 4 2.8GHz CPU and 768MB RAM. The operating system is Windows 2000 Professional with all unnecessary system daemon processes disabled.

Effectiveness

We collect URLs of 237 untrusted web sites from McAfee's *SiteAdvisor* [83] as the input URLs to our testing system. These untrusted sites were known to contain adware, spyware or browser exploit programs. Because most existing browse exploits target at unpatched web browsers, we use Internet Explorer 5.0 running on unpatched Windows 2000 in the testing. The security setting of IE is not to allow untrusted ActiveX control and other plug-ins to be installed on the system. To crawl as many sites as possible, we set the web crawler to crawl a maximum of three links on each scanned page in the same domain. The testing system is configured to host one IE instance per VM. After the rendering of a downloaded page is completed, we wait for 20 seconds before terminating the IE instance to provide more time for malicious scripts to get activated.

In one testing, our system crawled and browsed 8988 web pages from the 237 untrusted web sites. We found 68 pages that causes malicious or suspicious state changes through unpatched IE 5.0, counting approximately 0.8% of the total visited web pages. Among the 68 pages, 40 pages download executables to the testing VM, 9 pages attempt to modify registries in the testing VM, and 22 pages crash the browser process. The file and registry modifications are outside the browser's working directory and registry key, and are thus considered as the malicious side

Suspicious State Changes	Responsible Web Sites
Create executables with known names under Windows system directory (kernel32.exe, kernel32.dll)	teen-biz.com; teenlovecam.com
Create executables with random names under Windows system directory (csndl.exe, cvhur.exe, cstbt.exe, psqzs.exe, dmkhb.exe, uiqml.exe)	teen-biz.com
Create executables with random names under Windows installation directory (3v96e1bt.exe, xil0s9uo.exe)	realsearch.cc
Create unknown executables under administrator's profile directory (tm.exe)	yournewindustry.net
Attempt to modify *Run* registry key	teen-biz.com
Crash browsers (Drwtsn32.exe process created)	bitdesign.co.yu; winneronline.ru; 2famouslyrics.com; ...

Table 5.1: *A set of suspicious system state modifications due to successful browser exploits, and the web sites responsible for them.*

effects of a successful browser exploit. Table 5.1 lists a set of suspicious system state modifications that our system detects and the web sites responsible for them.

Our system detects suspicious state changes and writes them into the testing report. After the testing is completed, all the running VMs are terminated, and all the VM's file and registry states are successfully removed from the host environment. Before a VM is terminated, we can also copy all the suspicious executables created in the VM into an isolated environment for further analysis.

Testing Throughput

To demonstrate the efficiency of the FVM-based web site testing system, we measure the web site testing throughput when the system configuration parameters are varied. To remove wide-area network access delay from the throughput measurement, we first run the unmodified *WinHTTrack* web crawler to bring the test web sites to a local machine, and then start the web site testing run against the local web pages. Because of memory limitation, the testing machine can support a maximum of 60 concurrent IE instances. To avoid thrashing, we launch up to 50 concurrent IE instances in our experiments. In addition, we vary two system parameters: number of seconds to wait for each URL visit, and number of URLs visited

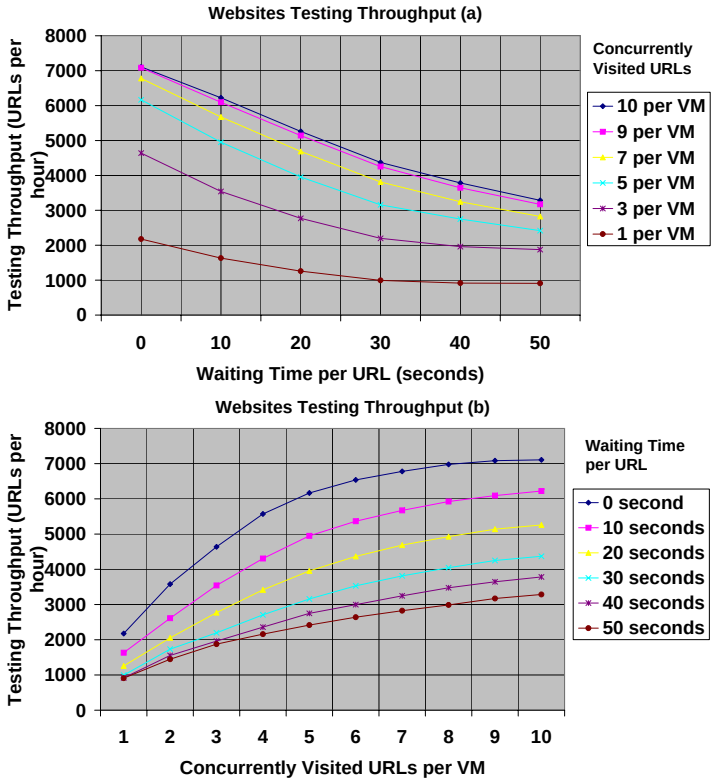

Figure 5.5: The throughput of web site testing when the number of seconds to wait for each URL visit and number of URLs visited per VM are varied.

per VM.

Figure 5.5 shows that the web site testing throughput increases when the waiting time for each URL visit decreases. When the system visits 10 URLs per VM with a waiting time of zero seconds, the throughput is 7,000 URLs per hour. As the waiting time is increased to 50 seconds, the throughput decreases to 3,400 URLs per hour. When the waiting time is short, each IE instance is terminated immediately after the rendering of a downloaded page is completed. Therefore, the IE process concurrency threshold (50) is never reached, and the web crawler can always creates new VMs to visit new URLs. However, as the waiting time increases, the number of concurrent IE instances and the number of worker threads in the crawler also increase. This in turn slows down the crawler's scanning speed and thus decreases the overall web site testing throughput. When the waiting time becomes long enough, many IE instances are running concurrently and the concurrency threshold (50) may be reached. Under this condition, the crawler spends most time on waiting for available VMs, and thus the throughput is also decreased as the waiting time is

longer.

As the number of URLs visited per VM increases, the web site testing throughput increases because the more URL visits per VM, the less frequently new VMs need to be started and the lower the impact of the VM start-up cost on the web site testing throughput. In our testing machine, restarting a VM in a clean state takes an average of 1.3 seconds under FVM. In comparison, rolling back a VM of VMware Workstation 5.0 on the same physical machine takes 30 to 50 seconds. The small VM start-up/shut-down cost of FVM significantly improves the throughput of our web site testing system.

5.4 Shared Binary Service for Application Deployment

A *shared binary service* is an application deployment architecture under which application binaries are centrally stored and managed, and are exported from a server to end user machines. With centralized management, this application deployment architecture greatly simplifies software patching/upgrade/repair and license control in a corporate environment without suffering performance penalty and scalability problem of the thin client computing model. Here we present the design and implementation of a binary server for Windows applications based on the FVM framework.

5.4.1 Application Deployment Architecture

Modern enterprises have adopted the following application deployment architectures to manage the application binaries and configurations on their end user machines: *local installation, thin client computing, OS steaming, network boot, application streaming* and *shared binary service.* In the *local installation* architecture, application program binaries are installed and executed on end user machines, and a network-wide application installation and patching tool is needed to reduce the application maintenance cost. In the *thin client computing* architecture, such as Microsoft *Windows Terminal Service* [84], application program binaries are installed on a central server and also executed on the server's CPU. An end user machine displays the results of application execution and replays user inputs via a special protocol such as *Remote Desktop Protocol* (RDP). Because of centralized binary installation and execution, this architecture lowers the overall application maintenance cost, but also incurs longer application latency and decreased runtime performance.

Network boot allows a diskless end user machine to boot from a kernel image stored on a remote server, and run the OS and applications on the machine's local CPU. Citrix's Ardence [85] supports network boot by implementing a remote virtual disk system. An end user machine boots from the network through *Preboot eXecution Environment* (PXE) and then accesses the virtual disk via a protocol called BXP. However, because of hardware dependencies, the virtual disk of each end user machine may be different from one another and require extensive customization. A similar application deployment architecture is *OS steaming* [86, 87], which configures an OS and a specific set of applications into a hardware-level virtual machine image and stores the image on a central server. An end user machine boots from one of such images and run it on a virtual machine monitor. Because a virtual machine image is isolated by the virtual machine monitor from the underlying machine hardware, in theory it has less hardware dependency. In practice, the hardware dependency is transferred to the virtual machine monitor, which is not necessarily the best party to deal with hardware dependencies because of its emphasis on minimal code base.

In the *application streaming* architecture [8, 23, 7], application binaries and configurations are bundled into self-contained packages, which are stored on a central server. An end user machine running a compatible OS can fetch these packages from the server and run them locally without requiring an installation process. Each such package runs in a virtual environment, and is cached on the local machine whenever possible. This architecture offers the advantages of both centralized application management and localized application execution on end user machines.

The *shared binary service* architecture is in fact similar to application streaming except that it does not require explicit application packaging, and that it allows resource sharing across application packages. More concretely, applications are installed on a shared binary server, and then exported to all end user machines through certain network interface. When the application is executed, accesses to configuration files, registry entries and libraries are redirected to the central binary server.

5.4.2 Overview of FVM-based Binary Server

The shared binary server architecture is widely used in the UNIX world. User machines typically mount binary files exported by a central binary server on local directories, and execute them directly on the local machine. In the Windows world, most application programs are not designed to be loaded from a shared repository and run locally. Instead, they are designed to run on the machine on which they are installed. Consequently, whenever an application program runs, it always tries to locate its operating environment, such as registries, configuration files and libraries

from the local machine.

To enable a Windows client machine to execute a program physically stored on a binary server, this program's accesses to its operating environment must be redirected from the local machine to the binary server. More specifically, when an application program accesses a system resource, the access should be redirected in the following ways: (1) when accessing application executables, libraries or configuration files, the access is redirected to the remote binary server; (2) when accessing a local file containing input/output data, the access should not be redirected.

Because FVM is designed to redirect the access of a process running in a VM, either to the host environment or to a VM's private workspace, we can easily support the access redirections on a binary server client machine based on the FVM framework. The basic approach is to associate a process started from an executable on the binary server with a special VM, whose redirection logic is modified in the way described above. To distinguish between accesses to local input/output data and those to application-specific configuration data, one could monitor an application's installation process and record its configuration files, DLLs and registry settings. The resulted application profile can then be sent to the client VM as a redirection criteria. However this process is cumbersome and error-prone. To avoid this profiling step, we use a simple redirection criteria. We notice that application installation usually updates only a few directories such as the program directory (*C:\Program Files*), the system directory (*C:\Windows or C:\Winnt*), the configuration directory for all users (C:*Documents and Settings\All users*), and usually modifies registry keys under *"HKLM\Software"*. Users rarely store their personal data in these locations. Therefore, the current shared binary server implementation simply redirects all accesses to these special directories and registry keys to the binary server. Testings with a variety of Windows applications suggest that this heuristic is reasonably reliable and effective.

5.4.3 Design and Implementation

The operating environment that a Windows application depends on includes its binary program, configuration files, registry entries, environment variables and COM/OLE components. When an end user machine executes a shared binary stored on a central binary server, the FVM layer on the end user machine properly redirects the shared binary's accesses to its operating environment to the central server. Because a VM share the same kernel of the end user machine, the current shared binary server does not support applications that require installation of kernel drivers. Figure 5.6 shows the system architecture of the proposed shared binary server for Windows-based end user machines.

Under the binary server architecture, users start an application by invoking the

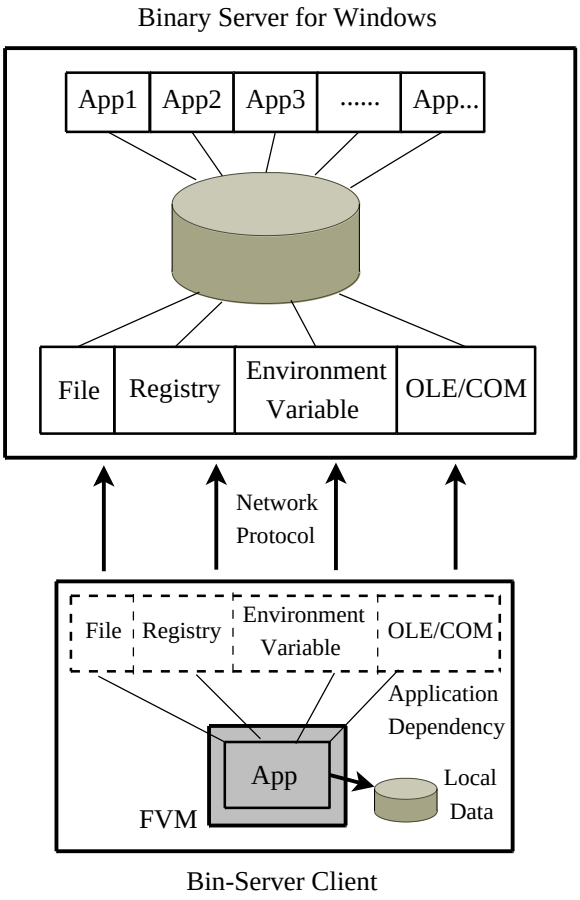

Figure 5.6: In the shared binary server architecture, all application programs are installed on the binary server. The FVM layer redirects a shared binary's accesses to its operating environment to the central server.

application's executable file, which is stored on the binary server and accessible to the client through remote file access/sharing protocol such as CIFS/SMB. We intercept process creations on the end user machine, and creates a VM to host the process if its executable image comes from the binary sever.

Because Windows supports remote file access through the *Common Internet File System (CIFS)*, redirecting file access is relatively straightforward. For example, when a process accesses one of its libraries using a local path, say "C:\Program Files\app.dll", FVM redirects the access to the binary server by renaming the system call argument to the UNC path "\\BinServ\C\Program Files\app.dll", where "BinServ" is the name of the binary server. In addition to application binaries, FVM can also redirect loading of Windows system DLLs to the binary server by intercepting system calls accessing memory-mapped DLL images. When system DLLs are loaded from the binary server, we require the end user machines to run the same OS kernel as the shared binary server. Additional administrative tasks on the binary server, such as setting up account for authentication, sharing binary server's disk partitions with read-only access, and disabling administrative shares, etc, are also required to implement this redirection mechanism. Discussion to these issues are out of the scope of this dissertation.

Unlike files, remote registry access on Windows is implemented completely at user space. Therefore, it is not possible to redirect registry accesses at the system call interface by renaming the registry-related system call arguments. Ideally, we should implement a registry access proxy in the FVM layer between the shared binary VM and the binary server to redirect the registry access. The current binary server prototype uses a simple alternative: we export registry entries under the *"HKLM\Software"* key on the binary server, and load these registry entries into the VM that is set up to run shared binaries when the VM is first created. Consequently, the shared binary VM has a local copy of all the registry entries associated with applications installed on the binary server, and any registry access from the shared binary VM is redirected to this local copy. We also periodically synchronize a shared binary VM's local registry entries copy with those on the central server to reflect newly installed applications.

In addition to files and registries, applications can install environment variables on the binary server. To allow an application running in a shared binary VM to access its environment variables that are set up at the installation time, the shared binary VM retrieves all the environment variables stored on the binary server and merges them with environment variables of the local environment.

Finally, the shared binary server prototype also correctly redirects accesses to *Component Object Model* (COM) components that shared binaries depend on. There are two types of COM objects. One is called *In-Process Object* (In-Proc), which is generally implemented as a DLL that an application process can load into

its address space. Redirecting an access to an In-Proc COM object is thus the same as redirecting accesses to DLLs. The other is called *Out-Of-Process Object* (Out-of-Proc), which is implemented as a stand-alone executable that runs as a separate process. An out-of-proc COM component allows multiple client processes to connect to it. When an application process running in a shared binary VM accesses an out-of-proc COM object, it should load the COM object from the binary server into the application, regardless of any locally installed COM objects. To implement this behavior, we assign a new *Class Identifiers (CLSID)*, a globally unique identifier, to each COM object accessed by applications running in a shared binary VM, and perform the necessary mapping between the old CLSID and the newly created CLSID during each COM object access. This mapping is maintained inside the FVM layer which forces the system to load COM objects from the binary server into the application process.

By redirecting file, registry and OLE/COM access to a remote binary server in the FVM layer, the binary server VM allows users to use Windows applications without installing them on the local machine. We have tested a few applications, such as LeapFtp and Winamp, which cannot be started through the file sharing interface unless running in the binary server VM. Please note that the shared binary service does not aim at reducing software license charges by installing less copies inside an organization. The application installed on the binary server should be software with proper license control mechanism.

5.4.4 Performance Evaluation

Because most of the performance overhead of the shared binary server architecture occurs at application start-up, we evaluate the performance of the FVM-based shared binary server prototype by measuring the start-up time of six interactive Windows applications in the following four different configurations:

- Local Installation and Execution (LIE): Applications are installed and executed on an end user machine.
- Shared Binary Service with Local Data (SBSLD): Applications are installed on a central server and executed on an end user machine with input/output files stored locally.
- Shared Binary Service with Remote Data (SBSRD): Applications are installed on a central server and executed on an end user machine with input/output files also stored on the central server.
- Thin Client Computing (TCC): Applications are installed and executed on a central server, with execution results displayed on an end user machine

through a Windows Terminal Service (WTS) session. Input/output files are stored on the central server as well.

We use a test harness program to launch an application under test using the *CreateProcess()* Win32 API, and monitors the application's initialization using the *WaitforInputIdle()* API. The start-up time of a test application corresponds to the elapsed time between the moments when these two API calls return. To measure the initialization time of a WTS session, we run a terminal service client ActiveX control program on an end user machine. When a WTS session is successfully established, this program receives a *Connected* event. The time between this event and the time when the program first contacts the terminal server is the WTS session's initialization time. We use two machines in this experiment. The client machine was an Intel Pentium-4 2.4GHz machine with 1GB memory running Windows 2000 server and the shared binary server machine was an Intel Pentium-4 2.4GHz machine with 256 MB memory running Windows 2000 server.

Figure 5.7 shows the start-up time comparison among these four configurations for the six test applications. The start-up overheads of the four configurations tested are in the following order: SBSRD $>$ SBSLD $>$ TCC $>$ LIE. In general, the amount of file/registry access over the network determines the magnitude of the initialization overhead. For LIE and TCC, the amount of file/registry access over the network is zero, and therefore their start-up times are smaller. Because TCC incurs an additional fixed cost (around 40 msec) of initializing a WTS session, its start-up time is longer than LIE's. Both SBSLD and SBSRD require access to the remote server for configuration files, registry entries, DLLs and COM/OLE components, and therefore incur a substantially higher start-up overhead. In the case of SBSRD, it incurs network access overhead even for input/output files and therefore takes longer to start-up than SBSLD.

The overhead order among the four configurations is different for WinWord. In this case, the start-up times of the four configurations are pretty close to each other, LIE: 463 msec, SBSLD: 479 msec, SBSRD: 494 msec and TCC: 517 msec. The reason that TCC exceeds SBSLD and SBSRD is because of its additional 40-msec WTS session initialization cost.

The start-up time of WinAmp is much higher than that of other test applications because the number of file access redirections is much higher (214) than others, as shown in Table 5.2. In other words, WinAmp requires access to 214 executable or configuration files, which are fetched from the binary server at the initialization time. Therefore, the execution of WinAmp using a binary server incurs a much larger start-up overhead than others. We believe a simple client-side caching mechanism may help significantly reduce this overhead. The large difference in SBSLD's and SBSRD's start-up times for WinAmp arises from the large input file used in the

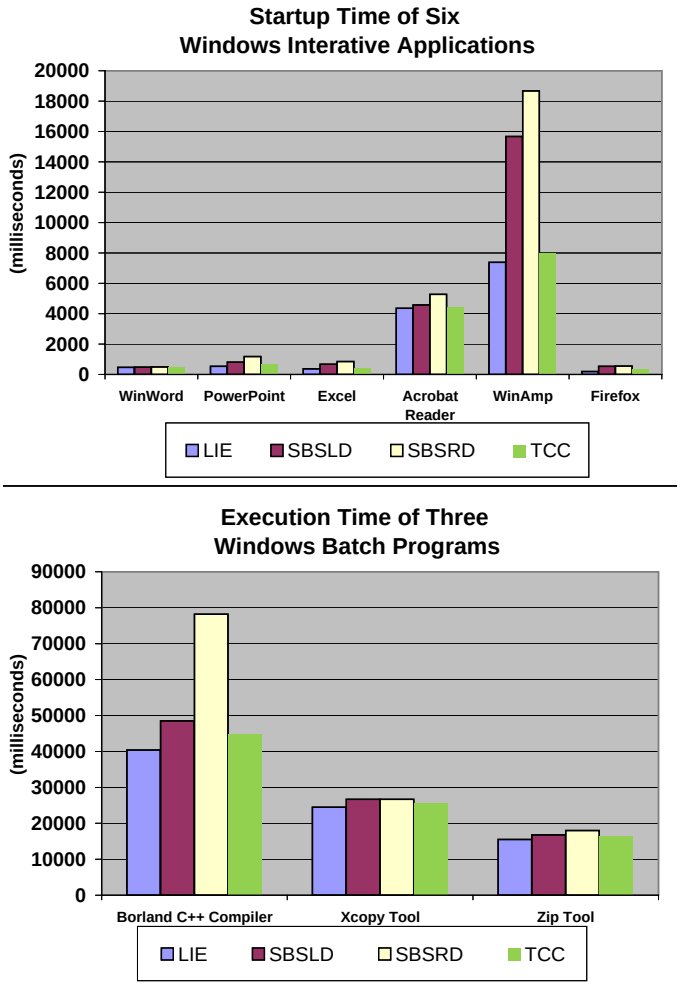

Figure 5.7: The start-up time of six interactive applications, and the execution time of three batch applications under four different test configurations.

Application	Registry Redirections	File Redirections
WinWord	996	62
PowerPoint	487	30
Excel	339	17
Acrobat Reader	152	55
WinAmp	397	214
Firefox	159	16

Table 5.2: *The number of registry and file access redirected to the binary server over the network during initialization for six interactive Windows applications.*

test of WinAmp, which is 5.2Mbytes in size.

In addition to interactive applications, we also measured the execution time of three batch programs in the above four configurations. The batch program's execution time in the four configurations follows the same order as the start-up time measurement for interactive applications, as shown in Figure 5.7.

5.5 Distributed *DOFS*

An enterprise has every incentive to protect its confidential documents from untrusted parties. However, information theft is difficult to detect and prevent, especially for information theft by insiders who have authorized access to stolen documents. We have developed a *Display-Only File Server* (DOFS), which physically isolates confidential documents from end users and redirects users to access these documents through terminal services in a transparent way. However, this thin-client computing design suffers from application compatibility and performance. In this chapter, we describe how to leverage the FVM framework to support a *distributed DOFS* system that distributes the computation to local CPUs.

5.5.1 Information Theft Prevention

Traditional solutions to prevent information theft are based on access controls and content filtering. Standard access control techniques provided by the OSes are not sufficient because they cannot prevent information theft by authorized users who have legitimate access to the stolen information. Content filtering monitors data transferred across an enterprise's network boundary to detect possible confidential information leakage. However, this technology is not effective with respect to encrypted packets and is therefore mainly for accidental leakage rather than malicious theft.

Digital Rights Management (DRM) [88, 89, 90] protects an enterprise intellectual property by encrypting a protected document and associating *fine-grained access rights* with the document. Typical rights include whether editing is allowed, when the rights expire, how the rights are transferred, etc. DRM systems tightly integrate encrypted documents with their access rights. When a DRMed document is accessed, the DRM client software decrypts the document, interprets the access rights, and authorizes proper document usage according to the specified rights. To effectively safeguard sensitive documents from unauthorized usage, the DRM client software still needs to integrate with tamper-resistent mechanisms to protect itself from motivated attackers. Because the high-level access rights are beyond the OS's

access control mechanism, DRM client software needs to be implemented as a plug-in for a standard document viewing application, such as WinWord, PowerPoint or AutoCAD, and therefore requires expensive customization.

Multilevel Security (MLS) systems label protected data and the accessing process, and ensure that no process may read data at a higher level, and that no process may write data to a lower level (the *Bell-LaPadula model* [91]). MLS systems are mainly used in military and government agencies, and are involved with more complicated access control policies. However, most commodity applications are not designed to assign or interpret security labels, and thus they have to be modified or rewritten to run on an MLS operating system. The Purple Penelope [92] is a MLS wrapper for Windows NT OS from the British Defense Evaluation and Research Agency. It allows unmodified applications to label files in a private file store, and to export files to a publicly shared file store. The label can be assigned to both files and clipboard data. Relabeling a labeled file follows a "trusted path" that requires the user's explicit confirmation.

A different solution to protect confidential documents against information theft is to prevent contents of confidential documents from physically residing outside specific servers, while allowing authorized users to access them as if they are stored locally. Sensitive files and clipboard data can be moved in the direction from end user machines to the servers, but not in the reverse direction. We have developed such a system called *Display-Only File Server* (DOFS) [12] using a thin client computing infrastructure.

5.5.2 Display-Only File Server (DOFS)

The display-only file server architecture stores confidential documents in remote file servers, and guarantees the following invariant: Once a document is checked into a DOFS server, the document content can never physically leave the server. Users authorized to access a document are granted a *ticket* that gives them the right to interact with that document but not the ability to access the document's bits. When a user clicks on a ticket on her local machine, the DOFS client invokes a proper application on the corresponding document in a *terminal service* session on the DOFS server, which displays the application's execution result on the user's desktop machine so the user can still access the document in the usual way. This architecture is named as "display-only file server" because its file servers can only export display but not file content to the users.

Figure5.8 is the software architecture of the DOFS system. The DOFS server consists of a *DOFS manager*, a *DOFS file protection driver*, and a *DOFS firewall*. The Server File System Driver (Server FSD) is Windows' *Common Internet File System* (CIFS) server, which allows a Windows server to share files to end users.

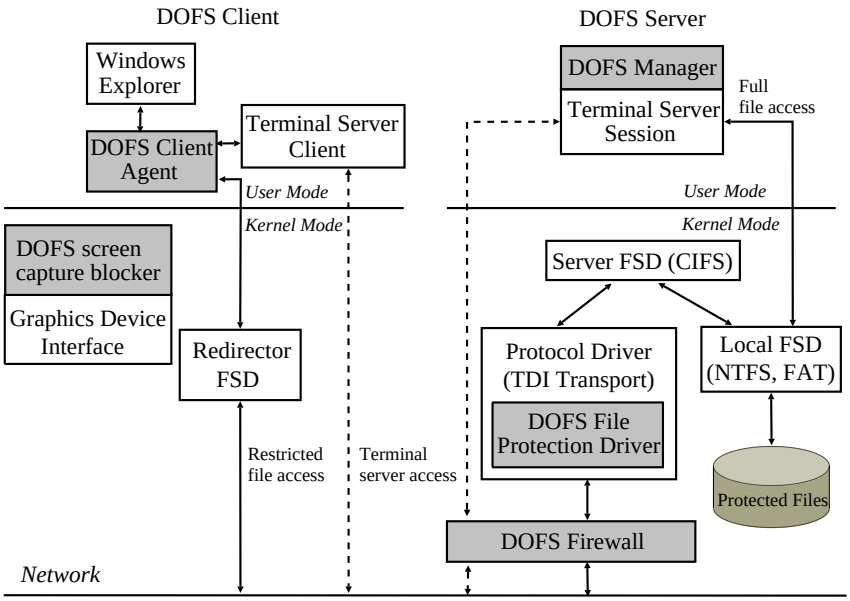

Figure 5.8: In the DOFS architecture, confidential documents are stored in a central DOFS server, and are rendered by processes running in a terminal session on the DOFS server.

We use the DOFS file protection driver to intercept the file operation requests to the server FSD. The DOFS file protection driver only allows client machines to copy files to the server file system or list the files. When a client machine reads a file from the DOFS server, the file protection driver returns a DOFS link similar to Windows file shortcut to the client. Therefore, the file protection driver guarantees that no real file contents from the DOFS server can leave the server. The DOFS firewall further restricts the protected files from leaving the server through other channels such as FTP and HTTP.

The DOFS client consists of a *DOFS client agent* and a *DOFS screen capture blocker*. The DOFS client agent communicates with the *Windows Explorer* process. Whenever the Windows Explorer process accesses a file stored on the DOFS server or accesses a DOFS link, the DOFS client agent takes control and launches a terminal service session to access the file on the DOFS server. After the terminal server client connects to the terminal server, the DOFS client agent sends a request to the DOFS manager to open the file through the terminal server session. The DOFS client agent can also allow clipboard data changes to be synchronized to a DOFS server application, thus enabling copy-and-paste from the DOFS client to the DOFS server. The screen capture blocker prevents client machines from taking a screen shot of the terminal service sessions used for DOFS file access.

A major weakness with the DOFS architecture is its reliance on the thin client computing architecture, which has both compatibility and scalability issues for certain applications. One idea is to distribute the DOFS server's execution environment to the clients so that document viewing applications physically run on the client machines but logically belong to the central DOFS server. More concretely, we distribute the centralized DOFS architecture by running a DOFS-VM on each client machine, and ensuring that each such DOFS-VM is administratively part of the central DOFS server. Because FVM provides an execution environment that is isolated from the rest of the host machine, we use the FVM framework to create and control the proposed DOFS-VM.

5.5.3 Design and Implementation of D-DOFS

The distributed DOFS architecture differs from the original DOFS in that the contents of confidential documents will physically come to the end user machine when users access them. To prevent the protected contents from being leaked onto the end user machine, the distributed DOFS should satisfy the following four requirements:

- Contents of confidential documents are encrypted on the fly when they leave the DOFS server, and are decrypted by the FVM layer. As a result, processes running outside of a DOFS-VM cannot get the decrypted content. Even if an attacker inserts a kernel module below the FVM layer, she still cannot decrypt the encrypted content.

- Processes running in a DOFS-VM can read from the local host environment, but any write operations to files and registry entries are redirected to the VM's workspace, which physically reside on the central DOFS server. In particular, file contents being written are encrypted by the FVM layer again before passing down to the lower-level driver or the network.

- Communication channels between the DOFS-VM and the host environment are largely blocked. The VM has no network access to other machines except to the central DOFS server. Processes outside of the VM are not allowed to observe the state of processes running in this VM, using approaches like reading the process address space, accessing clipboard data filled by VM processes, or installing global hooks in this VM.

- Processes running in a DOFS-VM, and higher-level drivers above the FVM layer can see the decrypted content. Therefore, we must prevent attackers at

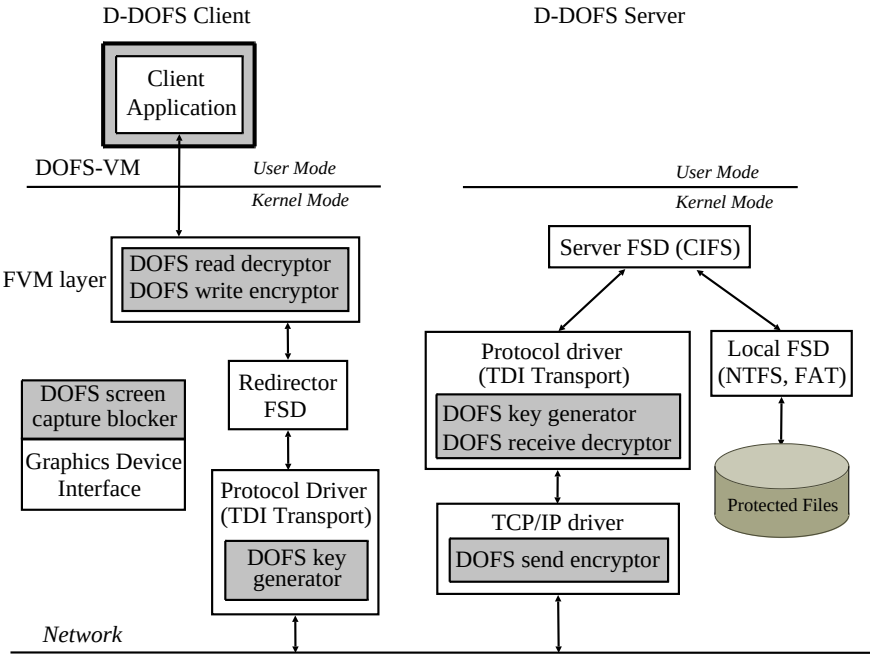

Figure 5.9: In the distributed DOFS architecture, confidential documents are stored in a central DOFS server, and are rendered by processes running in a special DOFS-VM on the user's machine, which is considered a logical extension of the DOFS server.

the two levels from using interfaces not confined by the FVM layer, such as a new system call, to capture the file content.

We assume that the FVM layer is not tampered by attackers on end user machines, and the key exchange and key storage are secure. Therefore, the above D-DOFS architecture design based on the FVM framework can effectively protect confidential document contents against information theft in an untrusted end user environment.

The system architecture of the distributed DOFS is shown in Figure 5.9. When an application running in a DOFS-VM reads a confidential file from a distributed DOFS server, the server first encrypts the requested file to prevent it from being leaked on the network. The Windows server's File System Driver (FSD) uses the *TCP_SendData* API in the TCP/IP driver to send file contents back to the requesting client machine. We built a *Transport Driver Interface* (TDI) filter driver called *DOFS send encryptor* to intercept this API and encrypt the outgoing file contents

using keys created by the *DOFS key generator*. At the client side, after FVM receives a file, it invokes the *DOFS read decryptor* to decrypt the data before forwarding it to the application. When an application running in a DOFS-VM writes data to a file, FVM invokes the *DOFS write encryptor* to encrypt the data. The server FSD uses the *TDI_Event_Receive* API to receive data from a client machine, and invokes the *DOFS receive decryptor* to decrypt the data. The encryption and decryption key used in a file access is generated on the fly. When an application at the client side needs to open or create a file on the DOFS server, the DOFS key generator on both the client and the DOFS server invoke the *Diffie-Hellman* key exchange algorithm [93] to generate a key for encrypting and decrypting the file.

To prevent an application running in the DOFS-VM from saving the decrypted data to the local machine, whenever the DOFS-VM is created to host the application accessing a confidential document on a DOFS server, the VM's workspace directory for files are initialized to a temporary directory on the DOFS server. This mechanism allows an application to use local configuration files or temporary files, but forces the application to save any file contents to the remote DOFS server instead of the local machine.

The FVM layer is modified to intercept *NtReadFile* and *NtWriteFile* in order to perform file decryption and encryption. As a result, lower-level kernel modules below the FVM layer only see encrypted file content. However, if an attacker intercepts the two system calls with a driver sitting on top of the FVM layer, she can access all decrypted data and directly create I/O request packet to write the decrypted data into the local host machine. To ensure no other drivers can intercept the two system calls after the FVM layer is loaded, each time before we perform file encryption and decryption, we verify if the system call table entries point to our intercepting functions inside the FVM layer. If the two system call table entries point to functions in other drivers, FVM will not perform the file encryption and decryption.

A malicious application running in the DOFS-VM can also save the decrypted data locally with support of certain malicious kernel drivers. An attacker can install a new system call with a kernel driver, or install a malicious kernel driver that communicates with the malicious application in the DOFS-VM. Therefore, once the malicious application gets decrypted data, it can use the new system call or communicate with the malicious driver to save the decrypted data locally, bypassing the sandbox of the FVM layer. There are two possible solutions to this type of attack. The first solution is to ensure that only certified applications can run in a DOFS-VM. Assuming the checksum of each certified binary is stored in a database, each time when a DOFS-VM launches an application, it verifies the checksums of all the needed binary files against the checksums stored in the database. However, this approach is expensive in maintaining the checksum database. Currently we use

the second approach, in which FVM intercepts the system call dispatch routine, and rejects any system calls that are not defined in the original Windows OS. FVM also prohibits applications from communicating with unknown kernel modules using the *NtDeviceIoControlFile* system call.

5.5.4 Performance Evaluation

To evaluate the performance overhead of applications running under the distributed DOFS architecture, we measure the start-up time of six interactive Windows applications in the following four different configurations:

- Local execution on remote files: applications run on the testing machine and open protected files on a remote server.
- D-DOFS execution on plain remote file: applications run in a DOFS VM on the testing machine and open remote protected files.
- D-DOFS execution on encrypted remote file: same as the second configuration except that the remote file is encrypted at the server and decrypted at the client.
- Execution under original DOFS. In this configuration, we install the DOFS server program on the testing machine, and measure the application start-up time from another machine where the DOFS client program is installed. The measured time is the start-up time of applications running in a terminal service session on the testing machine, plus the Terminal Server's connection time and the communication latency between the DOFS client and the DOFS server. This configuration ensures that applications run on the same CPU as the other three configurations. However, the original DOFS architecture requires the protected files reside on the DOFS server, so we copy the protected files onto the testing machine.

We use a Dell Dimension 2400 machine with an Intel Pentium-4 2.66GHz CPU and 1GB memory as the testing machine, and a Dell PowerEdge 600SC with an Intel Pentium-4 2.4GHz CPU and 256 MB memory as the remote server hosting protected files. We run the same set of applications as in the evaluation of the shared binary server application on a set of protected files. The measurement results are shown in Figure 5.10.

The distributed DOFS architecture introduces a small additional overhead (below 15%) to the applications' start-up time, which comes from FVM's virtualization

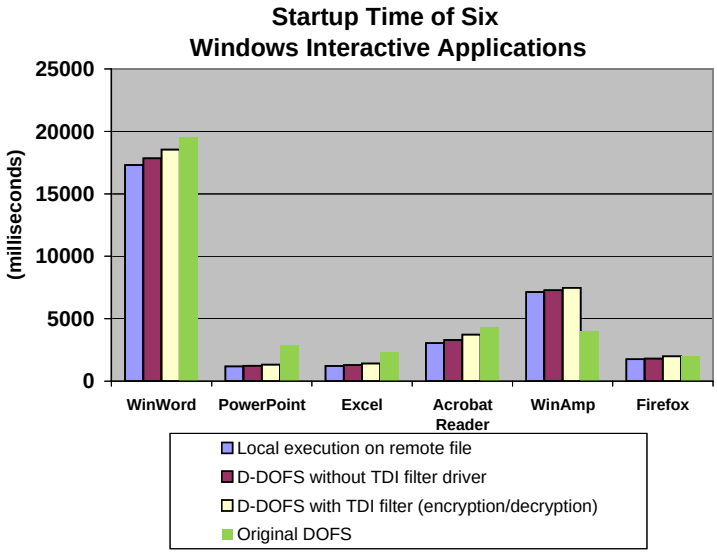

Figure 5.10: The start-up time of six interactive applications that operate on protected files under four different configurations. The performance overhead under the D-DOFS architecture is below 15%.

overhead and on-the-fly file encryption/decryption. The latter usually has an additional overhead below 8%. The start-up time under the original DOFS architecture is typically the largest among the four configurations, because of the overhead of communications between the DOFS client and the DOFS server. The only exception is WinAmp, where a large audio file of 10MB is used in the test. Because the audio file is placed on the same machine as the WinAmp application in the "original DOFS execution" configuration, the overhead of transmitting the large file over the network only shows up in the other three configurations and substantially increases their start-up times.

Chapter 6

Conclusion

6.1 OS-level Virtualization on Windows

OS-level virtualization technologies place the virtualization layer at the OS level by intercepting OS system calls, system libraries or I/O requests. User processes running on top of the layer are partitioned into different groups, each of which represents a logical VM and is given an isolated execution environment. The main idea to present multiple VMs on the same OS kernel is namespace virtualization, which assigns every VM its local namespace by renaming the target of an access request. To minimize each VM's resource requirement and to improve the scalability, a VM can share a base execution environment from the host machine, including files and configurations. The VM then evolves with a copy-on-write mechanism from the base environment. Because the VM environment provides one-way isolation, untrusted processes in the VM can never compromise the integrity of the host machine or other VMs. To achieve strong isolations in a VM, communications to other VMs or device drivers are blocked according to a configuration policy. Resource constraints are also associated with every VM to prevent service denial to the host machine or other VMs. Because OS-level VMs are running on a single OS kernel, they usually have smaller execution latencies, smaller resource requirement and higher scalability than most hardware-level virtual machines.

The *Feather-weight Virtual Machine (FVM)* is an OS-level virtualization implementation on Microsoft Windows OS. The main technical challenges in building FVM are the following: (1) There are many different types of OS objects that should be virtualized, such as kernel object and window management. We must identify and virtualize most if not all of them in order to achieve a reasonably accepted isolation. (2) There are various communication interfaces that should be confined under the scope of a VM, such as window message, hooks and daemon process

management. Without this step, different VMs can still communicate with each other without being isolated. (3) The Windows system call interface is not directly used by Win32 applications, and some system calls are even undocumented. We have implemented the FVM prototype on Windows 2000 and XP. We expect it to be extended to recent 32-bit Windows OS with small changes. The isolation testing and performance measurement demonstrate that processes running in a VM can be well isolated from other VMs and the host environment, and the performance overhead is typically less than 15%. As a result, the FVM architecture can be used as a flexible platform for security-related applications that require frequent invocation and termination of a large number of "playground" VMs.

A major contribution of the FVM architecture is that the FVM layer identifies and virtualizes most system components on the Windows OS, and provides a fully workable platform that supports a large number of OS-level VMs on a single physical machine. Each VM is effectively isolated from other VMs and the host environment. To achieve strong isolations over a single Windows kernel, extensive researches and investigations have been performed to recognize various communication channels on Windows. We hope the dissertation can provide researchers and developers a clear picture of how a comprehensive OS-level VM architecture is accomplished on the Windows platform, and promote more novel system development on this platform. The strong isolations in FVM enables the FVM framework to be used as a powerful foundation for many security-related applications. In addition, the interception techniques used in FVM are applicable to other systems requiring an additional layer between the applications and the Windows OS kernel. Because Windows is the operating system used by majority users, we believe that applications based on FVM can create many practical solutions to address issues in system security, privacy and reliability.

6.2 FVM's Applications on Security and Reliability

We have applied the basic FVM framework to a few applications that aim at improving system security and reliability for both end users and system administrators. Today's enterprises and organizations always face the threat of malware infection and information theft. In addition, application deployment and maintenance in a distributed personal computing environment are often a costly task. To resolve these problems, we present the design and implementation of secure mobile code execution service, distributed *Display-Only File Server* and shared binary service for application deployment. The customization of the FVM framework to fit these applications are straightforward.

Vulnerability assessment proactively scans network services to identify vulnerabilities before they are exploited by the attackers. A well known problem with vulnerability scanning is that vulnerability scanners may cause adverse side effects to the network services being scanned. To resolve the safety concern, we developed the VASE system to scan duplicated network services in a VM, thus confining any side effects within disposable VMs.

Malware researchers can use crawler-based approach to scan the internet and hunt for malicious web sites. The maliciousness of a web site is determined by monitoring web browsers' behaviors inside a VM. Given the large number of test sites, the scalability and scanning throughput of a web site testing system become significant. Therefore, we developed a scalable web site testing system with high scanning throughput based on the FVM framework.

6.3 Future Work

6.3.1 Live Migration

One future work of the FVM framework is to study how to correctly implement checkpoint/restart of a VM in order to support live migration. Live migration helps to move an active VM from one physical machine to another, and to maintain the active states of processes in the VM during the migration. There are two basic requirements in migrating a set of processes in a VM: (1) certain resource names used by processes in a VM should be virtual names, such as a process ID, a thread ID, or an object handle. FVM is responsible for maintaining the mappings between the virtual name and the physical name, and update the mappings on the destination machine of the migration. (2) state of processes should be captured as much as possible, including thread context, heap, static data and kernel states. Typically, states stored in the process address space are easy to be captured. However, every process has certain states stored in the kernel, such as opened file objects. How to completely recover these kernel states on the destination machine remains a challenge. One approach is to log the complete set of system calls on the source machine, and to replay the system calls in the log on the destination machine.

When a VM is checkpointed, we stop all processes in the VM, save the virtual-physical mappings, and capture process states. In addition, file and registry in the VM's workspace, and the VM's configurations, are also saved. On another physical machine that has an identical operating system and base environment, the VM can be restarted by creating a VM with the same workspace and configurations, creating the same set of processes in suspended mode, resetting all the process states, creating new virtual-physical mappings and finally starting all the processes.

6.3.2 Portable Computing Environment

As introduced in Chapter 2, a portable computing environment allows users to carry their applications and data in a portable storage device, and to use the included applications and data on any PC with a virtualization layer installed. We plan to apply the FVM framework to support such an application. In fact, the architecture of this application is similar to the architecture of the shared binary service and the distributed DOFS described in Chapter 5. The only difference is that the application environment and protected personal data are located on a removable hard drive instead of on a network drive.

Bibliography

[1] Kevin Lawton, Bryce Denney, N. David Guarneri, Volker Ruppert, and Christophe Bothamy. Bochs user manual. http://bochs.sourceforge.net/doc/docbook/user/index.html.

[2] VMware. Vmware products. http://www.vmware.com/products/home.html.

[3] Microsoft. Microsoft virtual pc 2007. http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx.

[4] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, 2000.

[5] Herbert Potzl. Linux-vserver technology. http://linux-vserver.org/Linux-VServer-Paper, 2004.

[6] Sun Microsystems. Solaris containers: Server virtualization and manageability. http://www.sun.com/software/whitepapers/solaris10/grid_containers.pdf, September 2004.

[7] Bowen Alpern, Joshua Auerbach, Vasanth Bala, Thomas Frauenhofer, Todd Mummert, and Michael Pigott. Pds: A virtual execution environment for software deployment. In *Proceedings of the 1st International Conference on Virtual Execution Environments*, 2005.

[8] Microsoft. Microsoft softgrid application virtualization. http://www.microsoft.com/systemcenter/softgrid/default.mspx.

[9] SWsoft. Virtuozzo server virtualization. http://www.swsoft.com/en/products/virtuozzo.

[10] Yang Yu, Fanglu Guo, Susanta Nanda, Lap chung Lam, and Tzi cker Chiueh. A feather-weight virtual machine for windows applications. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, June 2006.

[11] Fanglu Guo, Yang Yu, and Tzi cker Chiueh. Automated and safe vulnerability assessment. In *Proceedings of the 21th Annual Computer Security Applications Conference*, December 2005.

[12] Yang Yu and Tzi cker Chiueh. Display-only file server: A solution against information theft due to insider attack. In *Proceedings of 4th ACM Workshop on Digital Rights Management*, December 2004.

[13] Microsoft. Virtual pc for mac. http://www.microsoft.com/mac/products/virtualpc/virtualpc.aspx?pid=virtualpc.

[14] Jeremy Sugarman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[15] Intel. Intel virtualization technology. http://www.intel.com/technology/platform-technology/virtualization/index.htm.

[16] AMD. Introducing amd virtualization. http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_14287,00.html.

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

[18] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[19] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2001.

[20] OpenVZ. Unique features of openvz. http://openvz.org/documentation/tech/features.

[21] Thinstall. Application virtualization: A technical overview of the thinstall application virtualization platform. http://thinstall.com/assets/docs/ThinstallWP-ApplicVirtualization4a.pdf.

[22] Altiris Inc. Altiris software virtualization solution. http://www.altiris.com/upload/wp_svs.pdf, 2006.

[23] AppStream. Appstream technology overview. http://www.appstream.com/products-technology.html.

[24] Citrix. Citrix presentation server - the standard for delivering windows applications at the lowest costłanywhere. http://www.citrix.com/English/ps2/products/product.asp?contentID=186.

[25] Trigence. An introduction to trigence ae. http://www.trigence.com/products/index.html.

[26] Ann Ernst. Meiosys: Application virtualization and stateful application relocation. http://www.virtual-strategy.com/article/articleview/680/1/2, 2005.

[27] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[28] Shaya Potter, Jason Nieh, and Dinesh Subhraveti. Secure isolation and migration of untrusted legacy applications. Columbia University Technical Report CUCS-005-04, 2004.

[29] Ricardo A. Baratto, Shaya Potter, Gong Su, and Jason Nieh. Mobidesk: Mobile virtual desktop computing. In *Proceedings of the 10th ACM Conference on Mobile Computing and Networking*, 2004.

[30] Johny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin. A transparent checkpoint facility on nt. In *Proceedings of 2nd USENIX Windows NT Symposium*, August 1998.

[31] Tom Boyd and Partha Dasgupta. Process migration: A generalized approach using a virtualizing operating system. In *Proceeding of the 22nd International Conference on Distributed Computing Systems*, July 2002.

[32] Zhenkai Liang, V.N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of 19th Annual Computer Security Applications Conference*, December 2003.

[33] Weiqing Sun, Zhenkai Liang, R. Sekar, and V.N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of 12th Annual Network and Distributed System Security Symposium*, 2005.

[34] Doo San Sim and V. N. Venkatakrishnan. Suez: A distributed safe executionenvironment for system administration trials. In *Proceedings of 20th Large Installation System Administration Conference*, December 2006.

[35] GreenBorder. Greenborder. http://www.greenborder.com.

[36] Robert Balzer. Safe email, safe office, and safe web browser. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, 2003.

[37] Keith Brown. Security in longhorn: Focus on least privilege. http://msdn2.microsoft.com/en-us/library/Aa480194.aspx, 2004.

[38] Mojopac. Mojopac - take your pc in your pocket. http://www.mojopac.com/portal/content/splash.jsp.

[39] Ceedo. Ceedo products. http://www.ceedo.com.

[40] moka5. Moka5 livepc features. http://www.moka5.com/products/features.html.

[41] VMware. Take control of virtual desktops across the enterprise. http://www.vmware.com/products/ace.

[42] U3 smart drive. Bring the power of portable software to your usb flash drive. http://www.u3.com/smart/default.aspx.

[43] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R Ganger. Metadata efficiency in a comprehensive versioning file system. In *Proceedings of USENIX Conference on File and Storage Technologies*, April 2003.

[44] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, , and Erez Zadok. A versatile and user-oriented versioning file system. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2004.

[45] Ningning Zhu and Tzi-Cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, June 2003.

[46] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. In *Proceeding of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[47] MSDN Library. About transactional ntfs. http://msdn2.microsoft.com/en-us/library/aa363764.aspx, 2007.

[48] Wine. Wine user guide. http://winehq.org/site/docs/wineusr-guide/index.

[49] Interop Systems. Windows interoperatibility with unix and linux systems. http://www.interix.com.

[50] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[51] Wei Hua, Jim Ohlund, and Barry Butterklee. Unraveling the mysteries of writing a winsock 2 layered service provider. http://www.microsoft.com/msj/0599/LayeredService/LayeredService.aspx, 1999.

[52] skape and Skywing. Bypassing patchguard on windows x64. http://www.uninformed.org/?v=3&a=3&t=pdf, December 2005.

[53] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. Gatekeeper: Monitoring auto-start extensibility points (aseps) for spyware management. In *Proceedings of 18th Large Installation System Administration Conference*, November 2004.

[54] Gary Nebbett. *Windows NT/2000 Native API Reference*. New Riders Publishing, 2000.

[55] Joao B. D. Cabrera, Lundy Lewis, and Raman K. Mehra. Detection and classification of intrusions and faults using sequences of system calls. In *ACM SIGMOD Record, Volume 30, Issue 4*, December 2001.

[56] Cristina Abad, Jed Taylor, Cigdem Sengul, William Yurcik, Yuanyuan Zhou, and Ken Rowe. Log correlation for intrusion detection: A proof of concept. In *Proceedings of 19th Annual Computer Security Applications Conference*, December 2003.

[57] Emilie Lundin Barse and Erland Jonsson. Extracting attack manifestations to determine log data requirements for intrusion detection. In *Proceedings of 20th Annual Computer Security Applications Conference*, December 2004.

[58] Norman. Norman sandbox whitepaper. http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf, 2003.

[59] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. In *IEEE Security and Privacy, vol. 5, no. 2*, March 2007.

[60] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. In *Journal in Computer Virology, vol. 2, no. 1*, August 2006.

[61] David Zimmer. Sysanalyzer overview. http://labs.idefense.com/software/malcode.php, 2007.

[62] BindView. Strace for nt. http://www.bindview.com/Services/RAZOR/Utilities/Windows/strace_readme.cfm.

[63] Rohitab.com. Api monitor. http://www.rohitab.com/apimonitor/index.html.

[64] Jacob R. Lorch and Alan Jay Smith. The vtrace tool: Building a system tracer for windows nt and windows 2000. http://msdn.microsoft.com/msdnmag/issues/1000/VTrace.

[65] Symantec. Threat explorer. http://www.symantec.com/business/security_response/threatexplorer/threats.jsp.

[66] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, , and Sam King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of 13th Annual Network and Distributed System Security Symposium*, February 2006.

[67] Xavier Roche. Httrack website copier. http://www.httrack.com.

[68] Inc Prism Microsystems. Whatchanged for windows. http://www.prismmicrosys.com/whatchanged/index.htm.

[69] Roger A. Grimes. *Malicious Mobile Code - Virus Protection for Windows*, chapter 1. O'Reilly, 2001.

[70] Andrew Conry-Murray. Product focus: Behavior-blocking stops unknown malicious code. *http://mirage.cs.ucr.edu/mobilecode/resources_files/behavior.pdf*, June 2002.

[71] Lap chung Lam and Tzi cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Proceedings of Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[72] Wei Li, Lap chung Lam, and Tzi cker Chiueh. Automatic application-specific sandboxing for win32/x86 binaries. In *Proceedings of Program Analysis for Security and Safety Workshop*, July 2006.

[73] Tzi cker Chiueh, Harish Sankaran, and Anindya Neogi. Spout: A transparent distributed execution engine for java applets. *IEEE Journal of Selected Areas in Communications*, 20(7), September 2002.

[74] Lap chung Lam, Yang Yu, and Tzi cker Chiueh. Secure mobile code execution service. In *Proceedings of 20th Large Installation System Administration Conference*, December 2006.

[75] Dirk Balfanz and Daniel R. Simon. Windowbox: A simple security model for the connected desktop. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium*, August 2000.

[76] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. A safety-oriented platform for web applications. In *Proceedings of 2006 IEEE Symposium on Security and Privacy*, May 2006.

[77] Harry Anderson. Introduction to nessus. *http://www.securityfocus.com/infocus/1741*, October 2003.

[78] Kevin Novak. Va scanners pinpoint your weak spots. http://www.networkcomputing.com/1412/1412f2.html, 2003.

[79] Tony Bourke. Super smack. http://vegan.net/tony/supersmack/.

[80] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A crawler-based study of spyware on the web. In *Proceeding of the 13th Annual Network and Distributed System Security Symposium*, February 2006.

[81] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser analysis of web-based malware. In *Proceeding of the first Workshop on Hot Topics in Understanding Botnets*, April 2007.

[82] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. Spyproxy: Execution-based detection of malicious web content. In *Proceeding of the 16th USENIX Security Symposium*, August 2007.

[83] McAfee. Mcafee siteadvisor. http://www.siteadvisor.com/analysis/reviewercentral.

[84] Microsoft Corporation. Technical overview of windows server 2003 terminal services. http://download.microsoft.com/download/2/8/1/281f4d94-ee89-4b21-9f9e-9accef44a743/TerminalServerOverview.doc, January 2005.

[85] Citrix Ardence. Software-streaming platform. http://www.ardence.com/enterprise/products.aspx?id=56.

[86] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of 17th Large Installation Systems Administration Conference*, October 2003.

[87] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The collective: A cache-based system management architecture. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, May 2005.

[88] Microsoft Corporation. Windows rights management services. http://www.microsoft.com/windowsserver2003/technologies/rightsmgmt.

[89] Authentica. Authentica secure mail, authentica secure mobile mail, and authentica secure documents. http://software.emc.com/microsites/regional/authentica.

[90] Liquid Machines. Liquid machines document control. http://www.liquidmachines.com.

[91] David Elliott Bell. Looking back at the bell-la padula model. In *Proceedings of 21st Annual Computer Security Applications Conference*, December 2005.

[92] Bryony Pomeroy and Simon Wiseman. Private desktops and shared store. In *Proceedings of 14th Annual Computer Security Applications Conference*, December 1998.

[93] RSA Laboratories. Diffie-hellman key agreement protocol. http://www.rsa.com/rsalabs/node.asp?id=2248.

Appendix - FVM Source Code Organization

The current FVM prototype works on Windows 2000 and Windows XP operating system, and has been tested with many popular interactive applications, such as Microsoft Office, Adobe Acrobat and Internet Explorer, and a few applications running as daemon processes, such as Apache web server and MySQL server. The FVM's source code can be divided into three main components: the FVM kernel driver, the FVM user daemon/library, and the FVM user interface.

FVM Kernel Driver

The FVM kernel driver's code is under the *fvmDrv* folder. The driver code provides VM-related data structures and utility functions, and implements virtualization to file, registry and kernel object. It is built with *Driver Development Kit (DDK)* or the recent *Windows Driver Kit (WDK)*. The generated driver binary *hooksys.sys* should be copied to *\SystemRoot\system32\drivers*.

FVM User Daemon/Library

The user-level component includes the following modules:

- The FVM daemon under the *UserHook\fvmserv* folder. It implements the background daemon process that is responsible for loading the FVM driver and loading the user-level interception DLLs. The generated binary is *fvmserv.exe*.
- The Win32 API interception DLL under the *UserHook\fvmdll* folder. It intercepts Win32 API calls related to window and message to confine such IPCs in the scope of a VM. It also implements virtualization to daemon processes. The generated binary is *fvmdll.dll*.

- The window title renaming DLL under the *UserHook\fvmwin* folder. This DLL appends a VM name to the title text of a window running in a VM. The generated binary is *fvmwin.dll*.

- The Layered Service Provider (LSP) DLL under the *UserHook\lsp* folder. The code is based on the sample code provided by [51], and is compiled using the *nmake* command. It intercepts a few Winsock APIs to isolate the network subsystem in FVM. The generated binaries are *inst_lsp.exe* and *lsp.dll* The *lsp.dll* should be copied to *\SystemRoot\system32*.

FVM User Interface

The FVM user interface's code include the FVM management console under the *FVM_UI\fvm* folder, and the FVM shell under the *FVM_UI\fvmshell* folder. The management console's code implements a management GUI to support all the VM operations such as creating or deleting a VM, and the FVM shell's code provides a simple file browser in each VM, which is used to browse a VM's file system or launch application programs. The generated binaries are *fvm.exe* and *fvmshell.exe*, respectively. To ensure they can be launched successfully on a test machine, a few runtime libraries of the Visual C++ compiler should be copied to the same folder as the two binaries.

FVM Usage

All the generated FVM binaries, unless explicitly specified, should be placed under one folder on the physical machine. Users can start FVM in the following three steps:

1. Run "fvmserv.exe -i" to install and start the FVM service and driver ("fvmserv.exe -r" to stop and remove);

2. Run "inst_lsp.exe" to load the LSP DLL (run it a second time to unload);

3. Run "fvm.exe" to start the FVM management console.

After the FVM management console is started, we can perform various FVM operations. In a started VM, we can use the FVM shell to browse the file system, to open files, to create application shortcuts, or to launch application programs.