Applications of a Feather-weight Virtual Machine

Yang Yu Hariharan Kolam_govindarajan Lap-Chung Lam Tzi-cker Chiueh
Computer Science Department, Stony Brook University
{yyu, kghari, lclam, chiueh}@cs.sunysb.edu

Abstract

A *Feather-weight Virtual Machine* (FVM) is an OS-level virtualization technology that enables multiple isolated execution environments to exist on a single Windows kernel. The key design goal of FVM is efficient resource sharing among VMs so as to minimize VM startup/shutdown cost and scale to a larger number of concurrent VM instances. As a result, FVM provides an effective platform for fault-tolerant and intrusion-tolerant applications that require frequent invocation and termination of *dispensable* VMs. This paper presents three complete applications of the FVM technology: scalable web site testing; shared binary service for application deployment and distributed *Display-Only File Server* (DOFS). To identify malicious web sites that exploit browser vulnerabilities, we use a web crawler to access untrusted sites, render their pages in multiple browsers each running in a separate VM, and monitor their execution behaviors. To allow Windows-based end user machines to share binaries that are stored, managed and patched on a central location, we run shared binaries in a special VM on the end user machine whose runtime environment is imported from the central binary server. To protect confidential files in a file server against information theft by insiders, we ensure that file viewing/editing programs run in a VM, which grants file content display but prevents file content from being saved on the host machine. In this paper, we show how to customize the generic FVM framework to accommodate the needs of the three applications, and present experimental results that demonstrate their performance and effectiveness.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection; K.6.3 [*Management of Computing and Information Systems*]: Software Management

General Terms Security, Management

Keywords virtual machine, web crawler, browser exploit, binary server, information theft

1. Introduction

Virtual machine is a software abstraction that provides multiple execution environments on a single physical machine. A virtualization layer can be placed at different levels along the machine stack. Hardware-level virtual machines emulate the hardware and allow each VM to have its own operating system. In contrast, OS-level virtual machines partition the OS kernel and allow each VM to run a set of applications isolated from other VMs.

Feather-weight Virtual Machine (FVM) is an OS-level virtualization technology for Windows machines. It is designed to efficiently share resources of a physical machine among VMs running on top of it. FVM intercepts accesses to system resources from processes running inside a VM, and renames the target resources so that one VM cannot name resources owned by another VM, let alone compromise them. To enforce state isolation, when a VM attempts to modify certain resource, FVM makes a copy of that resource in the VM's private workspace through copy-on-write. Each VM in FVM has minimal physical resource requirement because it shares most of its resources with the base host environment. This design greatly reduces the VM's startup/shutdown time and is more scalable in terms of the number of concurrent VMs that can be supported on a physical machine. As a result, FVM provides an excellent platform for fault-tolerant and intrusion-tolerant applications that require frequent invocation and termination of "dispensable" VMs.

We have developed the FVM prototype on Windows 2000/XP and applied it to protect an end user machine from malicious mobile code by running web browsers, email clients and downloaded programs in a VM [1]. Another application of the FVM framework is an automated and safe vulnerability assessment system called *VASE* [2], which allows vulnerability scanning to run in a high-fidelity and safe manner by cloning production-mode network applications into a VM and running vulnerability scanners against this VM. In this paper, we present three new applications of FVM: *scalable web site testing*, *shared binary service for application deployment*, and *distributed Display-Only File Server*.

In a web site testing system, which proactively scans untrusted web sites to determine if they contain malicious content, we need a virtual machine technology to isolate potential malignant side effects of browser attacks of malicious web sites from the physical machine. FVM's small VM startup and shutdown overhead is a perfect match for this application and has been shown to significantly improve the throughput of a web site testing system.

In the shared binary service architecture, which requires an end user machine to fetch all its application binaries from a central server, we need to provide applications that are installed on a shared binary server but executed on a client machine. The application's execution environment, including registry settings, configuration files, DLLs and COM objects, also comes from 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) [3] is an information protection technology that guarantees bits of confidential documents never leave the protected file server after being checked in. Because DOFS is based on the thin-client computing architecture, it has both application compatibility and scalability problems. Dis-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'08, March 5–7, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-796-4/08/03...$5.00

tributed DOFS, which attempts to extend the control of a 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 eliminates the compatibility/scalability problems of centralized DOFS while retaining its effectiveness in protecting confidential documents from information theft.

2. Related Work

Virtual Machine Technology Virtual machine technology has been implemented at multiple levels of abstraction. Hardware-level virtual machines emulate the entire instruction set architecture [4] or the hardware abstraction layer [5, 6] to run guest operating systems with zero or small modifications. In contrast, OS-level virtual machines [7, 8, 9, 10] partition system resources on a single OS kernel so as to support multiple execution environments that are isolated from one another. Because these isolated execution environments are designed to share as much system resources as possible, OS-level VMs typically have smaller startup/shutdown overhead and larger scalability.

Web Site Testing The Strider HoneyMonkey [11] is a large-scale web site testing system to identify malicious web sites that exploit browser vulnerabilities. It consists of a pipeline of “monkey programs” running Internet Explorer browsers in virtual machines. From a given URL list, it drives browsers to visit each URL and detects browser exploits by monitoring unexpected state changes. Upon detection of an exploit, the system destroys the infected VM and restarts a clean VM to continue the test. A crawler-based study on spyware [12] uses a similar testing framework 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 and Firefox. A group in Google performed another analysis to web-based malware [13] using a similar approach. The SpyProxy [14] intercepts HTTP requests with an HTTP proxy, fetches the target web pages into a local cache, and checks the safety of the pages before passing them to the client browser for rendering. The malware detection procedure consists of both a static analysis to the web content and a dynamic analysis inside a VM.

All of these web site testing systems use hardware-level virtual machines to host web browsers. In contrast, we have developed a web site testing system based on FVM, an OS-level virtual machine technology. We will demonstrate that our system is more scalable and is a better fit for real-time browser exploit detection.

Application Deployment Architecture Modern enterprises have the following application deployment architectures to manage application binaries on end user machines: *local installation, thin client computing, OS steaming, network boot, application streaming* and *shared binary service.* In the *local installation* architecture, application binaries are installed and executed on end user machines, and a network-wide application installation/patching tool is needed to reduce the application management cost. In the *thin client computing* architecture, such as Microsoft’s *Windows Terminal Service* [15], application binaries are installed on a central server and executed on the server CPU. An end user machine displays the result of application executions and replays user inputs via a special protocol such as *Remote Desktop Protocol* (RDP). Due to centralized binary installation and execution, this architecture lowers the overall application maintenance cost, but also incurs larger application runtime latency and higher network traffic load.

Network boot allows a diskless end user machine to boot from a kernel image stored on a remote server, and then run the OS and application code on the machine’s local CPU. Citrix’s Ardence [16] 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 thus require extensive customization. A similar architecture is *OS steaming* [17, 18], which configures an OS and a specific set of applications into a hardware-level virtual machine image stored on a central server. An end user machine runs one of these VM images on a *Virtual Machine Monitor* (VMM). Because a VM image is isolated by the VMM from the underlying physical hardware, in theory it has less hardware dependencies. In practice, the hardware dependencies are transferred to the VMM, 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 [19, 20, 21, 22], application binaries and configurations are bundled into self-contained packages, which are stored and maintained centrally. An end user machine running a compatible OS fetches these packages from the server and runs them directly without local installations. 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, which is widely used in the UNIX world, is similar to application streaming except that it does not require explicit application packaging, and it allows resource sharing among packages. More concretely, applications are installed on a shared binary server, and then exported to all end user machines through a standard network file sharing interface. When an application is executed, accesses to binaries, configuration files, registry settings are redirected to the central binary server.

Information Theft Prevention Existing solutions to the information theft problem are based on access control and content filtering. Standard access control from the OS is 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 weak with respect to encrypted packets and is therefore mainly for accidental leakage rather than malicious theft.

Digital Rights Management (DRM) [23, 24, 25] protects an enterprise’s intellectual property by encrypting a protected document and associating *fine-grained* access rights with the document. Typical rights include whether modifications are 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. Because of this tight integration, DRM software needs to be implemented as a plug-in for standard document viewing/editing programs, such as Microsoft Office and AutoCAD, and therefore requires expensive customization.

3. FVM Architecture

FVM is designed to facilitate resource sharing among VMs in order to minimize the virtualization overhead and scale to a large number of concurrent VMs, while providing strong isolation between VMs. FVM virtualizes system resources at the OS system call interface, as shown in Figure 1. It virtualizes access requests to system resources such as files and registries by name re-mapping and copy-on-write. Much like virtual memory protection, processes running in one VM cannot interfere with those in another VM because they

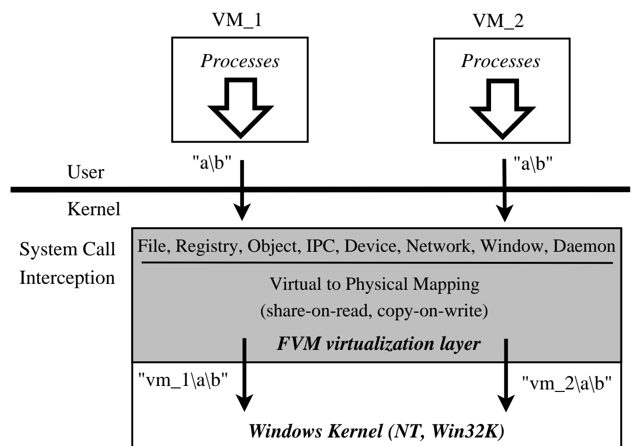

Figure 1. FVM supports isolated VM instances on a single OS kernel by re-mapping access to system resources at the OS system call interface.

cannot even name system resources owned by other VMs. In the FVM architecture, all VMs share the same kernel image, and start with exactly the same execution environment as the underlying host machine. Consequently, the startup cost of a new VM is minimal. Because FVM implements state isolation by copy-on-write, each VM only requires a small amount of system resources. As a result, FVM can easily support a larger number of concurrent VM instances than HAL-based virtualization technologies.

FVM intercepts each access to system resources by processes running in a VM at the system call interface and decides whether to pass through, redirect or block the system call, depending on whether the process is allowed to access the target resource and whether the access is read or write. For a read access to which the issuing process is entitled, FVM simply passes the access through to the kernel. For a write access, FVM makes a copy of the target resource in the corresponding VM's private workspace if such a copy does not exist, renames the access target, and then passes the access through to the kernel. To support strong isolation, processes in one VM are not visible to other VMs and therefore cannot be targets of inter-process communications; in addition, processes in a VM cannot tamper with the kernel state, such as loading/unloading a kernel driver, or accessing raw hardware devices such as network interface or physical memory directly. FVM implements these restrictions by blocking related system calls. FVM's virtualization to system resources includes the following:

File virtualization covers normal files, directory files and other NTFS structures such as alternate data streams and hard links. It allows a VM to leverage files from the host machine while maintaining copy-on-write versions in the VM's private workspace. It also moderates device accesses according to a configurable policy.

Registry virtualization covers Windows registry, which is a database where system and application configurations are stored. The registry database is loaded into system memory and accessed through system calls. Registry virtualization applies copy-on-write to handle modifications to registry entries in a way similar to file virtualization.

Kernel object virtualization covers named objects used for inter-process synchronization or communication, such as event, semaphore, mutex, etc. It isolates kernel objects associated with different VMs, and enables concurrent execution of multiple instances of the same program (such as WinWord and PowerPoint) in multiple VMs.

Network stack virtualization enables different VM instances on the same host to have different IP addresses, and allows multiple network server applications running in different VMs to use the same port number.

Interprocess communication confinement restricts the scope of common IPC mechanisms on Windows to the same VM, which include named pipe, mailslot, shared memory, local procedure call, socket, window message, clipboard, user-level hooks, etc.

Daemon process virtualization covers daemon processes (or services), which are managed by the *Service Control Manager* (SCM) on Windows. It enables daemon processes installed in one VM to be accessible to only that VM. However, some daemon processes such as SCM itself cannot be virtualized this way because they are integral components of Windows.

To prevent processes in a VM from exhausting resources of the physical machine and leading to denial-of-service, FVM imposes on each VM a set of constraints about its resource usage and network access, such as disk space, registry size, memory usage, scheduling priority, number of concurrent processes, etc. FVM uses the Windows *job object* mechanism to enforce some of these constraints, and periodically checks certain resource usage to enforce others by itself.

After processes running in a VM are terminated and the VM is stopped, FVM allows a user to either commit a VM's private state to the host environment, or discard the VM's state by simply deleting the VM. Committing a VM overwrites files and registries on the host environment with the VM's private versions. To protect the integrity of the host environment, FVM monitors sensitive file and registry locations that are common attack targets of malware programs, and aborts the commit operation if the VM contains such suspicious state changes.

Beyond the basic FVM prototype described in [1], we have made several important enhancements. First, the performance of file virtualization is greatly improved by keeping names of copy-on-write files in a VM's private workspace in the kernel memory. Whenever a file is copied from the host environment to a VM's private workspace, the full path of the file is deposited into a binary tree, with each node representing a component on the full path. As a result, FVM can decide where to redirect a file access request by looking up the binary tree instead of making system calls to check if the target file is already in the associated VM's private workspace. This optimization reduces the execution time of I/O-intensive applications running in a VM by up to 10%. Second, we have exported a set of FVM APIs, such as starting/stopping a VM, associating a new process with a VM, etc., for new applications that require direct manipulation of VMs. Finally, we have designed and implemented three new applications on FVM: scalable web site testing; shared binary service for deploying Windows applications, and distributed *Display-Only File Server* (DOFS) for information theft prevention. These new applications are described in detail in the next three sections.

4. Scalable Web Site Testing

4.1 Overview

Malicious web sites on the Internet can exploit a web browser's vulnerability to install malware on a user's machine without the user's consent or knowledge, thus posing a serious threat to web users. A proactive approach to protect users from these malicious sites is to periodically scan untrusted web sites and test their safety. 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 suspicious state changes on the testing machines. Because visiting malicious sites may cause permanent damage on a testing machine, each testing

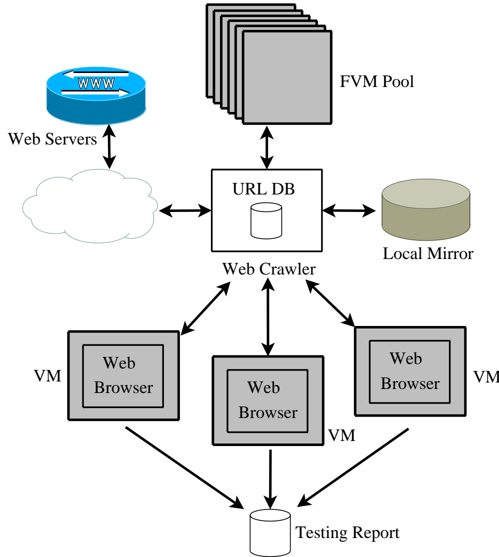

Figure 2. An automated and scalable web site testing system based on the FVM framework.

web browser should run on a virtual machine to isolate any side effects from the physical machine. When malicious changes are detected in a VM, or certain number of URLs have been tested in a VM, the VM will be terminated and a new VM is started to host new browser instances testing for other URLs.

Given a large number of URLs to be tested, a web site testing system should launch testing browsers in a large number of concurrent VMs. In addition, the overhead of restarting a clean VM should be very small because it is a frequent operation in web site testing. Existing web site testing systems [11, 12, 13, 14] use hardware-level VM technologies such as VMware and Xen, which are heavy-weight in that each VM runs its own instance of the guest OS. Typically, these testing systems need to deploy a cluster of physical machines each running a small number of VMs. The relatively high overhead of restarting a hardware-level 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 on a single OS kernel. As a result, FVM makes it possible to terminate and restart a clean VM within seconds, thus significantly improving the throughput and scalability of a web site testing system.

To test a collection of untrusted web sites under the FVM framework, we launch a separate web browser instance to access each URL in a separate VM. 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 entries in the testing VM's private workspace and the state of the browser process, we can reliably detect suspicious changes to the VM's state.

4.2 Design and Implementation

The system architecture of our FVM-based web site testing system is shown in Figure 2. The input is a collection of URLs of untrusted web sites, and the output is a report of malicious URLs and their suspicious state changes. The web crawler is based on an open-source website copier called *WinHTTrack* [26], which can scan the input URLs recursively and mirror the scanned sites on a local machine. The crawler can scan remote sites directly, or mirror the remote sites to a local machine and then scan them. We modified the web crawler to launch a separate *Internet Explorer* (IE) instance to visit each URL. The crawler associates each new IE instance with an available VM in the FVM pool, and starts a worker thread to monitor the execution of the IE instance. After the worker thread has been started, the web crawler moves on to visit the next URL.

Each VM in our testing system 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's operating system.

To detect malicious state changes in visiting a 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 an IE instance and the worker thread in the web crawler. 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 [11, 12].

After the page 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 changes, and finally deletes the VM.

The analysis module in our web site testing system monitors file/registry updates and process states in a VM 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)* [27]. 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 by 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 are 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, such as 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 (e.g. completion of page rendering) occurs.

4.3 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

Suspicious State Changes	Responsible Web Sites
executables with known names under Windows system directory (kernel32.exe, kernel32.dll)	teen-biz.com; teenlovecam.com
executables with random names under Windows system directory (csndl.exe, cvhur.exe, cstbt.exe, psqzs.exe, dmkhb.exe, uiqml.exe)	teen-biz.com
executables with random names under Windows installation directory (3v96e1bt.exe, xil0s9uo.exe)	realsearch.cc
unknown executables under administrator's profile directory (tm.exe)	yournewindustry.net
attempt to modify *Run* registry key	teen-biz.com
browser crashes (drwtsn32.exe process created)	bitdesign.co.yu; winneronline.ru; 2famouslyrics.com; ...

Table 1. *A set of suspicious state changes due to successful browser exploits and the web sites responsible for them.*

is Windows 2000 Professional with all unnecessary system daemon processes disabled.

4.3.1 Effectiveness

We collect URLs of 237 untrusted web sites from McAfee's *SiteAdvisor* [28] as the input URLs to our testing system. These untrusted sites were suspected to contain adware, spyware or browser exploit. 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. 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 additional 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 caused suspicious state changes in the VM, counting approximately 0.8% of the total visited web pages. Among the 68 pages, 40 pages downloaded executables to the testing VM, 9 pages attempted to modify registries in the testing VM, and 22 pages crashed the browser process. The file and registry modifications are outside the browser's *sandbox* directory and registry key, and are thus considered as malicious side effects of a successful browser exploit. Table 1 lists a set of suspicious state changes that our system detects and the web sites responsible for them.

4.3.2 Testing Throughput

We measure the throughput of our web site testing system under various configuration parameters. To remove the wide-area network access delay from the throughput measurement result, 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 mirror. Because of memory limitation, the testing machine can support a maximum of 60 concurrent IE instances. To avoid thrashing, we limits the maximum number of concurrent IE instances across all VMs to be 50 in this experiment. In addition

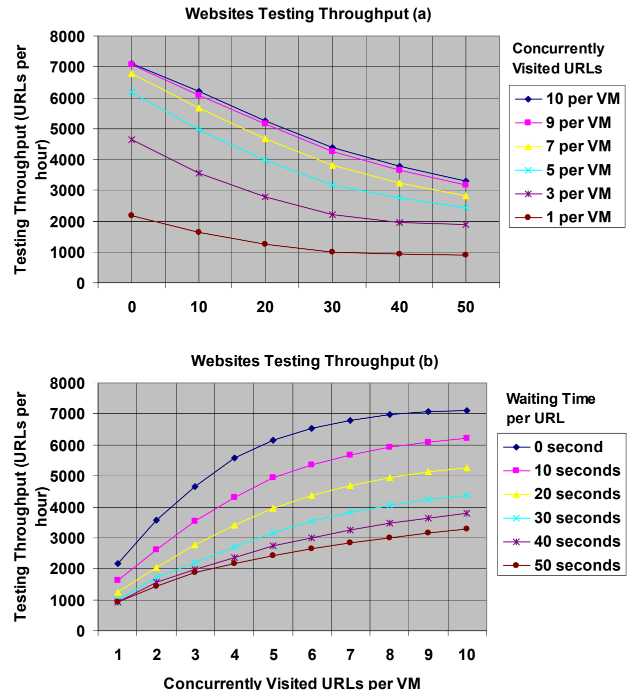

Figure 3. The throughput of our web site testing system when the number of seconds to wait for each URL visit and the number of URLs visited per VM are varied.

we vary two system parameters, number of seconds to wait for each URL visit, and number of URLs visited per VM.

Figure 3 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 small, 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 start new IE instances to visit new URLs without waiting. 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 testing throughput.

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 startup cost on the 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 to a clean state on the same physical machine takes 30 to 50 seconds. As a result, the small VM startup/shutdown cost of FVM significantly improves the throughput of a web site testing system.

5. Shared Binary Service

5.1 Overview

The shared binary server architecture is widely used in the UNIX world. User machines typically mount binary files exported by a central binary server onto local directories, and execute them directly on the local machine. In the Windows world, application pro-

grams are normally not designed to be loaded from a shared repository to run locally. Instead, they are designed to run on the machine where they are installed. Consequently, whenever a Windows application runs, it always tries to locate its execution environment, such as registries settings and libraries from the local machine. In contrast, in the shared binary server architecture, application binaries are installed on a central server, and when they run on an end user machine, they depend on the execution environment on the central server rather than the end user machine.

To enable an end user machine to execute a program physically stored on a binary server, this program's access to its execution environment must be redirected from the local machine to the binary server. More specifically, when an application program accesses certain resources, the access should be redirected in the following ways: (1) when accessing application binaries 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 go to the local machine without redirection. Because the basic FVM framework already intercepts file-related and registry-related system calls, it is straightforward to modify it to support the above redirection rules. With this redirection mechanism, users can run applications stored on a central binary server as if they are installed locally.

To distinguish between accesses to local input/output data and accesses to application binaries or configurations, one could monitor an application's installation process and record its installed files and registry settings. However this process takes time to complete. To avoid this profiling step, we use a simple redirection criteria. We notice that application installations usually update 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 design simply redirects all accesses to these special directories and registry keys to the binary server. Testing with a variety of Windows applications suggests that this heuristic is reasonably effective.

5.2 Design and Implementation

The execution environment that a Windows application depends on includes its binary and configuration files, registry settings, environment variables, COM/OLE components, etc. When an end user machine launches a shared program binary stored on a central binary server, the FVM layer on the end user machine can properly redirect the program's accesses to its execution environment to the binary server. To avoid potential conflicts between applications installed on the same binary server, each application can also be installed on a separate VM on the binary server. Because FVM instances run on the same OS kernel, the current shared binary server does not support applications with kernel driver components. Figure 4 shows the system architecture of the proposed shared binary server for Windows-based end user machines.

Because Windows supports remote file access through its *Common Internet File System* (CIFS) interface, it is straightforward to redirect file accesses to the binary server. For example, when a process attempts to access 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 a UNC path "\\BinServ\C\Program Files\app.dll", where "BinServ" is the name of the binary server. In addition to application binary and configuration files, FVM can also redirect loading of Windows system DLLs to the binary server by intercepting system calls accessing memory-mapped DLL images. Because of system DLL reloading, we require the end user machines to run the same OS kernel as the shared binary server.

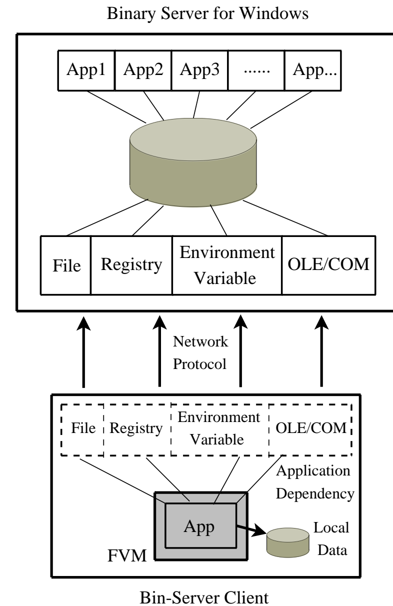

Figure 4. In the shared binary server architecture, all application programs are installed on the binary server. When users launch a program from a local machine, FVM redirects the program's accesses to its execution environment to the binary server.

Unlike files, remote registry access on Windows is implemented completely at the user level. Therefore, it is not possible to redirect registry accesses at the system call interface by renaming the registry-related system call arguments. The current binary server prototype uses a simple alternative: we export registry settings under the "HKLM\Software" key on the binary server, and load these settings into the VM that is set up to run shared binaries when the VM is first started. Consequently, the shared binary VM has a local copy of all the registry settings associated with applications installed on the binary server, and accesses to these registry settings from the shared binary VM are redirected to this local registry copy. We also periodically synchronize the local registry copy with the registry on the binary server to update settings of newly installed applications.

To allow an application running in a shared binary VM to access its environment variables set up at the installation time, the shared binary VM retrieves all the applications' environment variables stored on the binary server and merges them with local environment variables when the VM is started.

Our shared binary server prototype also correctly redirect accesses to *Component Object Model* (COM) components that shared binaries depend on. There are two types of COM objects. One is the *In-Process Object* (In-Proc), which is generally implemented as a DLL that an application process can load into its address space. Redirecting accesses to an In-Proc COM object is thus the same as redirecting accesses to DLLs. The other is the *Out-of-Proc Object* (Out-of-Proc), which is implemented as a stand-alone executable that runs in a separate process. An Out-of-Proc COM object 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 contact an instance of the Out-of-Proc object running in the same VM. To enforce 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.

5.3 Performance Evaluation

Because most of the performance overhead of the shared binary server architecture occurs at the application startup time, we evaluate the performance of the FVM-based binary server prototype by measuring the startup time of six interactive Windows applications in the following four 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 in a Windows Terminal Service (WTS) session. Input/output files are also stored on the central server.

We use a test harness program that launches an application program with the *CreateProcess()* call, and monitors the application's initialization with the *WaitforInputIdle()* call. The startup time of an application corresponds to the elapsed time between the moments when these two API calls return. To measure the setup 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 setup time. We use two machines in this experiment. The client machine is an Intel Pentium-4 2.4GHz machine with 1GB memory running Windows 2000 server and the shared binary server machine is an Intel Pentium-4 2.4GHz machine with 256 MB memory running Windows 2000 server.

Figure 5 shows the startup time comparison among these four configurations for the six test applications. The startup overheads of the four configurations 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 startup overhead. For LIE and TCC, the amount of file/registry access over the network is zero, and therefore their startup times are smaller. Because TCC incurs an additional fixed cost (around 40 msec) of setting up a WTS session, its startup time is longer than that of LIE. Both SBSLD and SBSRD require access to the remote server for configuration files, registry settings, DLLs and COM/OLE components, and therefore incur a substantially higher startup 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 the application of WinWord. In this case, the startup 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 fixed 40-msec WTS session setup cost.

The startup time of WinAmp is much higher than that of other test applications because the number of file access redirections overh the network is much higher (214) than others, as shown in Table 2. In other words, WinAmp requires accesses 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 startup overhead than other test applications. We believe a simple client-side caching mechanism can help to significantly reduce this overhead. The large difference between SBSLD's and SBSRD's startup time for WinAmp arises from the large input file (a 5.2MB file) used in the test.

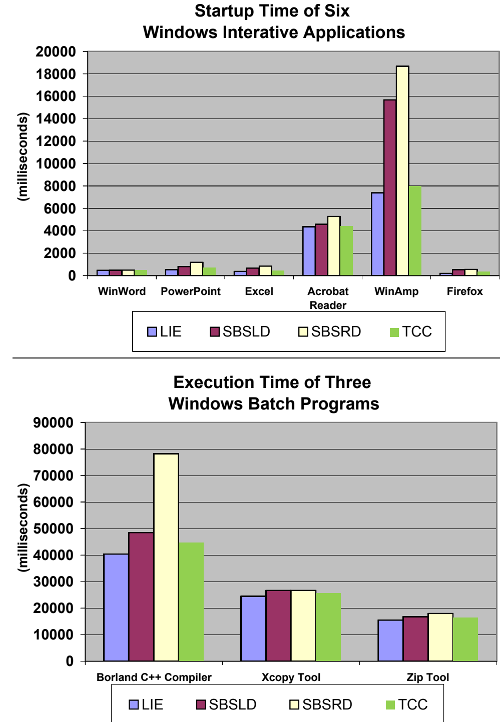

Figure 5. The startup time of six interactive applications, and the execution time of three batch applications under four different test configurations.

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 2. *The number of registry and file access redirected to the binary server during the initialization time for six interactive Windows applications.*

In addition to interactive applications, we also measured the execution time of three batch programs using the same four configurations. These batch programs' execution times in the four configurations follow the same order as the startup time measurement for interactive applications, as shown in Figure 5.

6. Distributed *DOFS*

6.1 Overview

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 the stolen documents. One solution to this problem is to prevent contents of confidential documents from physically residing outside protected servers, while allowing authorized users to access them as if they are stored locally. We have developed a *Display-Only File Server* (DOFS) that stores confidential documents in a central file server, and guarantees the fol-

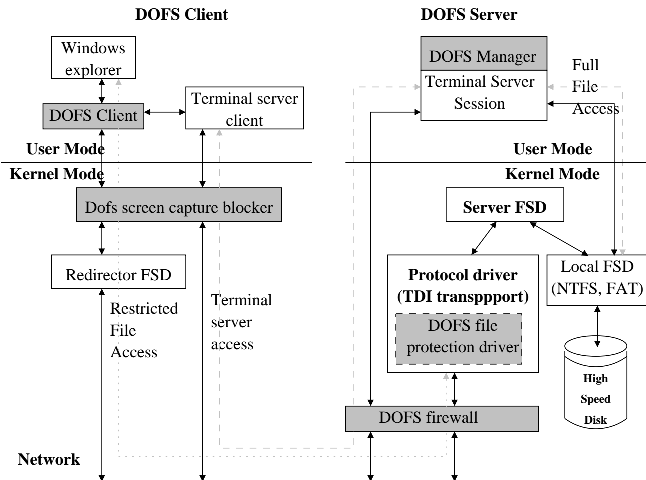

Figure 6. Confidential documents are stored in a central DOFS server. A DOFS client machine is only allowed to copy files to and list files on the server. Once a file is copied to the DOFS server, it cannot leave the server anymore. If a user accesses a file mounted from the central server, the DOFS client software transparently launches a terminal server session to the DOFS server, and opens the requested file in this terminal server session.

lowing invariant: once a document is checked into a DOFS server, the document content can never physically leave the server. Users authorized to access the document are granted a *ticket* that gives them the right to interact with the document but not the ability to access the document's bits. When a user clicks on a ticket on the 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 machine so the user can still access the document in the usual way.

Figure 6 shows the software architecture of the original DOFS. The DOFS server software consists of a DOFS Manager, a DOFS file protection driver, and a DOFS firewall. The server File System Driver (Server FSD) allows a Windows machine to share files with other machines. We use the DOFS file protection driver to intercept file access requests to the server FSD. The DOFS file protection driver only allows a client machine to copy files to the server file system or to list files on the server. When a client machine reads a file from a DOFS server, the file protection driver only returns a DOFS link similar to a Windows file shortcut to the client. Therefore, the file protection driver guarantees that no real file contents can leave the DOFS server through the file sharing interface. The DOFS firewall further prevents files from leaving the server through other interfaces such as HTTP and FTP.

The DOFS client software consists of a DOFS client module and a DOFS screen capture blocker. The DOFS client module interacts with the Windows explorer process. Whenever the Windows explorer process attempts to open a file stored on a DOFS server, the DOFS client module takes control and launches a new terminal service session to access the corresponding file on the server. The screen capture blocker prevents a client machine from taking screen shots of the terminal service session used for DOFS file access.

A major flaw with the original DOFS is its reliance on the thin-client computing architecture, which has both compatibility and scalability issues on certain Windows applications. One idea is to distribute the DOFS server's execution environment to the client machines so that document viewing/editing applications physically run on the client machines but logically belong to the central DOFS

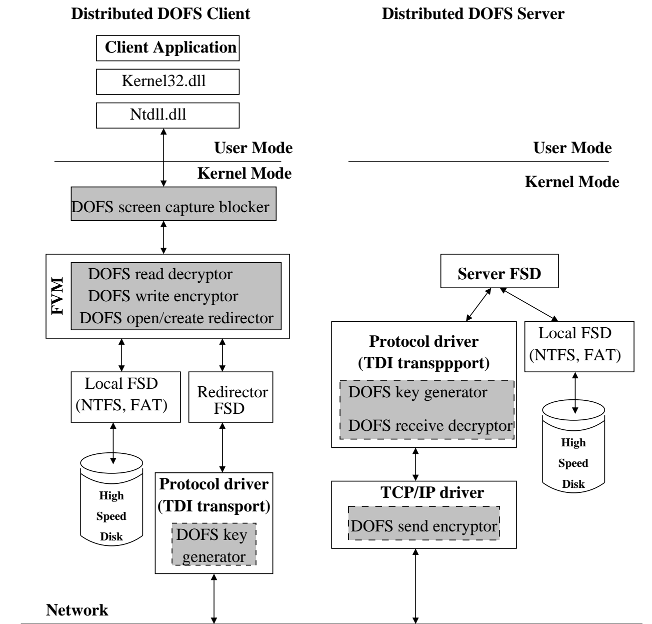

Figure 7. In the distributed DOFS architecture, confidential documents are stored in a central DOFS server, and are rendered by processes running in a special DOFS-VM on the user's machine. The DOFS-VM is considered as a logical extension of the DOFS server.

server. More concretely, we need to distribute the previously 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.

The difference between the shared binary server architecture and the distributed DOFS architecture is that, in the former setup the application binaries and execution environments come from a central server, but in the latter case application binaries and most of their execution environments are from the local clients. To enforce the invariant that bits of confidential documents never leave the DOFS server, which is now extended to include DOFS-VMs running on client machines, the distributed DOFS needs to intercept file I/O request for all processes running in the DOFS-VMs. The DOFS-VM is specially configured in the following ways:

- Processes running in the DOFS-VM can physically retrieve confidential documents stored in a DOFS server.
- Modifications to files and registry entries by processes in the DOFS-VM are redirected to the VM's private workspace, which resides on the central DOFS server rather than on the client machine.
- Most communication channels between the DOFS-VM and its host machine are blocked. The VM has no network access to other machines except to the central DOFS server. Processes outside of this VM are not allowed to look into the state of processes running in this VM in any way, such as reading the process address space or installing global hooks in this VM.

6.2 Design and Implementation

The system architecture of the distributed DOFS is shown in Figure 7. When a client machine reads a file from a distributed DOFS server, the server first encrypts the requested file to prevent it from being leaked during its transmission over the network. The Windows server's file system driver uses the `TCP_SendData` API from the TCP/IP driver to send file contents back to the requesting

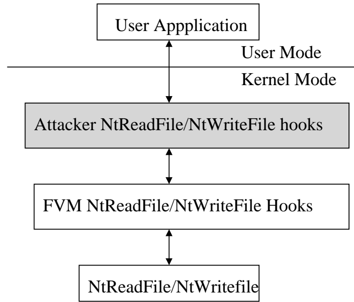

Figure 8. If an attacker can intercept the NtReadFile and NtWriteFile system calls before the FVM layer does, she can save the decrypted data in the local machine.

client machine. We have built a Transport Driver Interface (TDI) driver called *DOFS send encryptor* to intercept this API and to encrypt the outgoing file contents using keys created by the *DOFS key generator*. On the client side, after the FVM layer 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 on the DOFS server, FVM invokes the *DOFS write encryptor* to encrypt the data. The server FSD uses the `tdi_event_receive driver` API to receive data from a client machine, and invokes the *DOFS receive decryptor* to decrypt the data before saving it to the disk.

To prevent an application running in the DOFS-VM from saving the decrypted data to the local machine, FVM intercepts the `NtOpenFile` and `NtCreateFile` system calls to stop such write operations. More concretely, FVM only allows an application in the DOFS-VM to open local files for read. When an application opens a local file for write, we copy the target file to the central server and open the file copy on the server. When an application creates a new file, we also create it on the central server. This mechanism allows an application to use its local configuration or temporary files while preventing it from storing data of confidential documents locally. The encryption and decryption key used in a file access is generated on the fly. When an application on 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 [29] to generate a key to encrypt or decrypt the file.

FVM intercepts the system calls `NtReadFile` and `NtWriteFile` to perform file decryption and encryption. If an attacker intercepts these two system calls before FVM does, as shown in Figure 8, she can access all the decrypted data. To ensure our system call interceptions are always before any other hooks, each time before we perform file encryption or decryption, we check if the system call table contains our hook functions. If the system call table entries for `NtReadFile` and `NtWriteFile` are not our hook functions, the file encryption or decryption operations will fail.

An attacker can also add a new system call or load a malicious kernel driver to bypass the FVM's restriction so that a malicious application running in the DOFS-VM can save the decrypted data locally by directly invoking the new system call or communicating with the malicious driver. There are two possible solutions to this type of attacks. 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. Currently we use the second approach, in which FVM intercepts the system call dispatch routine, and rejects all system calls that are not defined in the original Windows kernel.

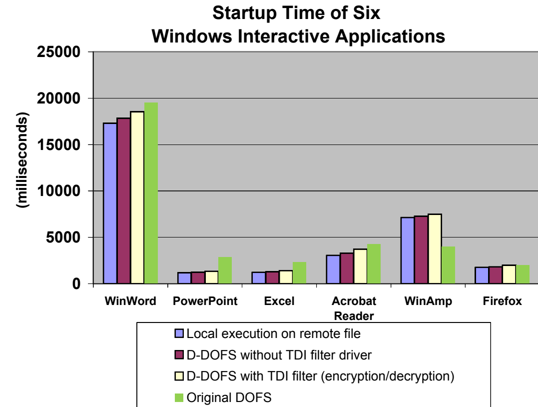

Figure 9. The startup time of six interactive applications operating on protected files under four different configurations. The performance overhead under D-DOFS is below 15%.

FVM also restricts applications in the DOFS-VM from making the `NtDeviceIoControlFile` system call, which is used by user applications to communicate with kernel drivers. Currently, we only allow user applications running in the DOFS-VM to communicate with a few well-known Windows system drivers, such as the TCP/IP drivers.

6.3 Performance Evaluation

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

- Local execution on remote files: applications run on the test machine and open protected files on a remote server.
- D-DOFS execution on plain-text remote files: applications run under FVM on the test machine and open remote protected files.
- D-DOFS execution on encrypted remote files: same as the second configuration except that the remote files are encrypted on the server and decrypted on the client on the fly.
- Execution under original DOFS. In this configuration, we install the DOFS server program on the test machine, and measure the application startup time from another machine where the DOFS client program is installed.

We use a Dell Dimension 2400 machine with an Intel Pentium-4 2.66GHz CPU and 1GB memory as the test 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 FVM-based shared binary server prototype on a set of protected files. The measurement results are shown in Figure 9.

The distributed DOFS architecture introduces a small additional overhead (below 15%) to an application's startup time, which comes from the FVM's resource virtualization and on-the-fly file encryption/decryption. The latter usually has an additional overhead up to 8%. The startup time under the original DOFS architecture is typically the largest among the four configurations, because of the overhead of communication between the DOFS client machine and the DOFS server using the RDP protocol. The only exception is the WinAmp application, 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 startup times.

7. Conclusion

Feather-weight Virtual Machine (FVM) allows a single machine to host multiple isolated execution environments on a single Windows kernel. By name space virtualization and copy-on-write, FVM enables multiple VMs to efficiently share resources without interfering with one another. FVM's minimal VM startup/shutdown cost and large scalability make it an excellent platform for building applications that require frequent invocation and termination of "dispensable" VMs.

This paper describes the design, implementation and evaluation of three new applications of FVM: scalable web site testing; shared binary service for application deployment, and distributed Display-Only File Server. In web site testing, we crawl untrusted web sites and launch web browsers in VMs to visit potentially malicious web pages. Suspicious state changes in a VM indicates a browser exploit. The FVM-based web site testing system has successfully identified a set of web sites exploiting browser vulnerabilities, with a testing throughput of up to 7,000 URLs per hour, the highest throughput ever reported in the literature. In shared binary service, we install application programs on a central server and allow end user machines to run them locally. This allows all end user machines to share application binaries, and transparently run the applications in a VM that presents an execution environment the same as that on the central server. We have successfully leveraged FVM to implement a shared binary service prototype that exhibits moderate performance overhead. In the distributed DOFS architecture, we protect confidential files on a file server against information theft by running file viewing/editing programs in a VM, which redirects all the write operations back to the central file server. Confidential files are encrypted and decrypted on the fly as they transmit over the network to ensure that end user machines never have the plain-text file content. The distributed DOFS prototype introduces a small startup overhead for interactive Windows applications, typically below 15%.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] VMware. VMware Products. http://www.vmware.com/products/home.html.

[6] 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.

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

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

[9] Sun Microsystems. Solaris Containers: Server Virtualization and Manageability. http://www.sun.com/software/whitepapers/solaris10/grid_containers.pdf, September 2004.

[10] SWsoft. Virtuozzo Server Virtualization. http://www.swsoft.com/en/products/virtuozzo.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Citrix Ardence. Software-Streaming Platform. http://www.ardence.com/enterprise/products.aspx?id=56.

[17] 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.

[18] 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.

[19] Microsoft. Microsoft SoftGrid Application Virtualization. http://www.microsoft.com/systemcenter/softgrid/default.mspx.

[20] AppStream. AppStream Technology Overview. http://www.appstream.com/products-technology.html.

[21] 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.

[22] Thinstall. Application Virtualization: A Technical Overview of the Thinstall Application Virtualization Platform. http://thinstall.com/assets/docs/ThinstallWP-ApplicVirtualization4a.pdf.

[23] Microsoft Corporation. Windows Rights Management Services. http://www.microsoft.com/windowsserver2003/technologies/rightsmgmt.

[24] Authentica. Authentica Secure Mail, Authentica Secure Mobile Mail, and Authentica Secure Documents. http://software.emc.com/microsites/regional/authentica.

[25] Liquid Machines. Liquid Machines Document Control. http://www.liquidmachines.com.

[26] Xavier Roche. HTTrack Website Copier. http://www.httrack.com.

[27] 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.

[28] McAfee. McAfee SiteAdvisor. http://www.siteadvisor.com/analysis/reviewercentral.

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