A Survey on Virtualization Technologies

Susanta Nanda   Tzi-cker Chiueh
{susanta,chiueh}@cs.sunysb.edu
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

Abstract

Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and others. Virtualization technologies find important applications over a wide range of areas such as server consolidation, secure computing platforms, supporting multiple operating systems, kernel debugging and development, system migration, etc, resulting in widespread usage. Most of them present similar operating environments to the end user; however, they tend to vary widely in their levels of abstraction they operate at and the underlying architecture. This paper surveys a wide range of virtualization technologies, analyzes their architecture and implementation, and proposes a taxonomy to categorize them on the basis of their abstraction levels. The paper identifies the following abstraction levels: instruction set level, hardware abstraction layer (HAL) level, operating system level, library level and application level virtual machines. It studies examples from each of the categories and provides relative comparisons. It also gives a broader perpective of the virtualization technologies and gives an insight that can be extended to accommodate future virtualization technologies under this taxonomy. The paper proposes the concept of an extremely lightweight technology, which we call as *Featherweight Virtual Machine* (FVM), that can be used to ”try out” untrusted programs in a realistic environment without causing any permanent damage to the system. Finally, it demonstrates FVM's effectiveness by applying it to two applications: secure mobile code execution and automatic clean uninstall of Windows programs.

1 Introduction

Virtual machine concept was in existence since 1960s when it was first developed by IBM to provide concurrent, interactive access to a mainframe computer. Each virtual machine (VM) used to be an instance of the physical machine that gave users an illusion of accessing the physical machine directly. It was an elegant and transparent way to enable time-sharing and resource-sharing on the highly expensive hardware. Each

VM was a fully protected and isolated copy of the underlying system. Users could execute, develop, and test applications without ever having to fear causing a crash to systems used by other users on the same computer. Virtualization was thus used to reduce the hardware acquisition cost and improving the productivity by letting more number of users work on it simultaneously. As hardware got cheaper and multiprocessing operating systems emerged, VMs were almost extinct in 1970s and 1980s. With the emergence of wide varieties of PC based hardware and operating systems in 1990s, the virtualization ideas were in demand again. The main use for VMs then was to enable execution of a range of applications, originally targeted for different hardware and OSes, on a given machine. The trend is contuing even now.

”Virtuality” differs from ”reality” only in the formal world, while possessing a similar essence or effect. In the computer world, a *virtual environment* is perceived the same as that of a *real environment* by application programs and the rest of the world, though the underlying mechanisms are *formally* different. More often than not, the virtual environment (or virtual machine) presents a misleading image of a machine (or resource) that has more (or less) capability compared to the physical machine (or resource) underneath for various reasons. A typical computer system already uses many such technologies. One such example is the virtual memory implementation in any modern operating system that lets a process use memory typically much more than the amount of physical memory its computer has to offer. This (virtual memory) also enables the same physical memory to be shared among hundereds of processes. Similarly, multitasking can be thought of as another example where a single CPU is partitioned in a time-shared manner to present some sort of a virtual CPU to each task. In a different setting, a cluster of medium-speed processors can be grouped together to present a single *virtualized* processor that has a very high clock speed. There are lots and lots of examples in today’s world that exploit such methods. The umbrella of technologies that help build such virtualized objects can be said to achieve tasks that have one common phenomenon, `virtualization`.

With the increase in applications of virtualization concepts across a wide range of areas in computer science, the girth of the definition has been increasing even more. However, just for the discussions in this paper, we use the following relaxed definition: *”Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others”*. Although virtualization can, in general, mean both partitioning as well as aggregation, for the purposes of this paper, we shall concentrate on only patitioning problems (as these are much more prevalent). A virtualization layer, thus, provides infrastructural support using the lower-level resources to create multiple `virtual machines` that are independent of and isolated from each other. Sometimes, such a virtualization layer is also called `Virtual Machine Monitor`(VMM). Although traditionally VMM is used to mean a virtualization layer right on top of the hardware and below the operating system, we might use it to represent a generic layer in many cases. There can be innumerous reasons how virtualization can be useful in practical scenarios, a few of which are the following:

- **Server Consolidation:** To consolidate workloads of multiple under-utilized machines to fewer machines to save on hardware, management, and administration of the infrastructure

- **Application consolidation:** A legacy application might require newer hardware and/or operating systems. Fulfilment of the need of such legacy applications could be served well by virtualizing the newer hardware and providing its access to others.

- **Sandboxing:** Virtual machines are useful to provide secure, isolated environments (sandboxes) for running foreign or less-trusted applications. Virtualization technology can, thus, help build secure computing platforms.
- **Multiple execution environments:** Virtualization can be used to create mutiple execution environments (in all possible ways) and can increase the QoS by guaranteeing specified amount of resources.
- **Virtual hardware:** It can provide the hardware one never had, e.g. Virtual SCSI drives, Virtual ethernet adapters, virtual ethernet switches and hubs, and so on.
- **Multiple simultaneous OS:** It can provide the facility of having multiple simultaneous operating systems that can run many different kind of applications.
- **Debugging:** It can help debug complicated software such as an operating system or a device driver by letting the user execute them on an emulated PC with full software controls.
- **Software Migration:** Eases the migration of software and thus helps mobility.
- **Appliances:** Lets one package an application with the related operating environment as an appliance.
- **Testing/QA:** Helps produce arbitrary test scenarios that are hard to produce in reality and thus eases the testing of software.

Accepting the reality we must admit, machines were never designed with the aim to support virtualization. Every computer exposes only one ”bare” machine interface; hence, would support only one instance of an operating system kernel. For example, only one software component can be in control of the processor at a time and be able to execute a privilged instruction[1]. Anything that needs to execute a privileged instruction, e.g. an I/O instruction, would need the help of the currently booted kernel. In such a scenario, the unprivileged software would trap into the kernel when it tries to execute an instruction that requires privilege and the kernel executes the instruction. This technique is often used to virtualize a processor.

In general, a virtualizable processor architecture is defined as *”an architecture that allows any instruction inspecting or modifying machine state to be trapped when executed in any but the most privileged mode”*. This provides the basis for the isolation of an entity (read *virtual machine*) from the rest of the machine. Processors include instructions that can affect the state of a machine, such as I/O instructions, or instructions to modify or manipulate segment registers, processor control registers, flags, etc. These are called ”sensitive” instructions. These instructions can affect the underlying virtualization layer and rest of the machine and thus must be trapped for a correct virtualization implementation. The job of the virtualization layer (e.g. the virtual machine monitor) is to remember the machine state for each of these independent entities and update the state, when required, only to the set that represents the particular entity. However, the world is not so simple; the most popular architecture, x86, is not virtualizable. It contains instructions that, when executed in a lower-privileged mode, fails silently rather than causing a trap. So virtualizing such architectures are more challenging than it seems.

[1]Instructions that are allowed to be executed only when the processor is in the highest privilege mode

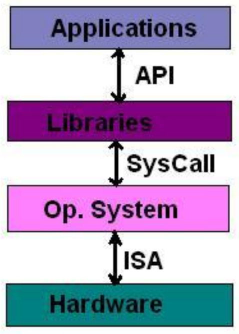


Figure 1: Machine stack showing virtualization opportunities.

Architectures aside, there are many other problems that make virtualization difficult. Since a virtualization layer (e.g. a Virtual Machine Monitor) has little knowledge regarding what goes on inside a virtual machine, it is typically hard for it to know what not to do. For example, a page fault exception caused by a guest OS inside one virtual machine should not be handled by the virtualization layer, and rather, be left to the guest OS to handle by itself. Similarly, a virtualization layer in the OS level should not handle any system call issued by one of the processes in a virtual machine; rather, should be left to its guest OS kernel to handle. Optimizations are also hard to achieve in the virtualization software as it does not know when a virtual machine does not need some resource. For example, it is hard for the VMM to know if the guest OS inside one of its virtual machine (VM) instances is running idle thread and is wasting processor cycles that can be allocated to other VMs for better performance.

Conceptually a virtual machine represents an operating environment for a set of user-level applications, which includes libraries, system call interface/service, system configurations, daemon processes, and file system state. There can be several levels of abstraction where virtualization can take place: instruction set level, hardware abstraction layer (HAL), OS level (system call interface), user-level library interface, or in the application level (depicted in Figure 1). Whatever may be the level of abstraction, the general phenomenon still remains the same; it partitions the lower-level resources using some novel techniques to map to multiple higher level VMs transparently.

Virtualization at the instruction set architecture (ISA) level is all about instruction set emulation. Emulation is the technique of interpreting the instructions completely in software. For example, an x86 emulator on Sparc processor can execute any x86 application, thus giving the illusion to the application as if it is a real x86 processor. To achieve this, however, an emulator would have to be able to translate the guest ISA (x86 here) to the host's ISA (Sparc).

The functionality and abstraction level of a HAL level virtual machine lies between a real machine and an emulator. A virtual machine is an environment created by a VMM, which is the virtualization software lying between the bare hardware and the OS and gives the OS a virtualized view of all the hardware. A VMM can create multiple virtual machines (VMs) on a single machine. While an emulator provides a complete layer between the operating system or applications and the hardware, a VMM manages one or more VMs where every VM provides facilities to an OS or application to believe as if it runs in a normal environment and directly on the hardware.

Virtualization at the OS level work at on top of or as a module in OS to provide a virtualized system call interface. Since system call invocation is the only way of communication from user-space to kernel-space, it should be possible for the virtualization software to totally control what the user-space processes can do by managing this interface.

Most applications use the APIs exported by user-level libraries rather than direct system calls for the implementation of their logic. Since most systems provide well-documented APIs and well-defined ways to hook them, such an interface becomes another candidate for virtualization. Virtualization at the library interface is possible by controlling the communication link between the applications and the rest of the system through the API hooks. This can, in turn, choose to expose a different implementation altogether using the same set of API and still have a running system. WINE [1] does a similar thing to support Windows applications on top of Unix/X.

Virtualization at the application level is a little different. This is not a case of inserting the virtualization layer in the middle; rather, it implements a virtualization layer as an application that eventually creates a virtual machine. The created VM could be as simple as a language interpreter or as complex as JVM.

All these virtualization technologies, however, differ significantly in terms of performance, flexibility, ease of use, resource consumption, and scalability; hence, differ in their usage scenarios as well. For example, instruction set emulators (operate at the ISA level) tend to possess very high latencies which makes it impractical to use them on a regular basis, unlike commercial virtual machines that operate at HAL level. However, they are very useful for debugging and learning purposes as every component is implemented in software that is fully under user's control. Commercal VMs, like VMware, give the flexibity of using different OSes or different versions of the same OS on the same machine by presenting a complete machine interface; this demands a much higher amount of resources. When this flexibility is not necessary, OS level virtual machines are more useful. Although they expose the OS same as that of the underlying one in all of its virtual machines, the resource requirement is much lower, performance much better, and manipulations (e.g. creation) much faster. However, it compromizes on the level of isolation as all the VMs use the same kernel and can potentially affect the whole system. Library-level virtualization technologies are extremely lightweight and can even help build a different subsystem altogether under the same infrastructure (e.g. Win32 on Unix). Application- level virtualization find applications in mobile computing and building trusted computing infrastructures. However, being in the application-level, these suffer from extra overhead.

There are innumerous examples where virtualization concept is used in various abstraction models. This paper, however, concentrates only the virtualization concepts involved in building virtual machines. For this reason, concepts involved in, say, a virtual file system or virtual memory are overlooked. It makes an attempt to capture and study a good number of such developments and come up with a taxonomy for them. Section 2 through 6 discuss virtualization technologies at various abstraction levels by studying some example cases. After these background studies, we identify a potential class in the taxonomy that has not been tapped and try to concretize the idea with a project proposal, **Featherweight Virtual Machine**. We discuss the current implementation status and a couple of its applications. All these are discussed in section 7. Section 8 concludes with the summary and some future directions.

2 Virtualization at the Instruction Set Architecture Level

Virtualization at the instruction set architecture level is implemented by emulating an instruction set architecture completely in software. A typical computer consists of processors, memory chips, buses, hard drives, disk controllers, timers, multiple I/O devices, and so on. An emulator tries to execute instructions issued by the guest machine (the virtual machine that is being emulated) by translating them to a set of native instructions and then executing them on the the available hardware. These instructions would include those typical of a processor (add, sub, jmp, etc on x86), and the I/O specific instructions for the devices (IN/OUT for example). For an emulator to successfully emulate a real computer, it has to be able to emulate everything that a real computer does that includes reading ROM chips, rebooting, switching it on, etc.

Although this virtual machine architecture works fine in terms of simplicity and robustness, it has its own pros and cons. On the positive side, the architecture provides ease of implementation while dealing with multiple platforms. As the emulator works by translating instructions from the guest platform to instructions of the host platform, it accomodates easily when the guest platform's architecture changes as long as there exists a way of accomplishing the same task through instructions available on the host platform. In this way, it enforces no stringent binding between the guest and the host platforms. It can easily provide infrastructure through which one can create virtual machines based on, say x86 on platforms such as x86, Sparc, Alpha, etc. However, the architectural portability comes at a price of performance. Since every instruction issued by the emulated computer needs to be interpreted in software, the performance penalty involved is significant. We take three such examples to illustrate this in detail.

2.1 Bochs

Bochs [2] is an open-source x86 PC emulator written in C++ by a group of people lead by Kevil Lawton. It is a highly portable emulator that can be run on most popular platforms that include x86, PowerPC, Alpha, Sun, and MIPS. It can be compiled to emulate most of the versions of x86 machines including 386, 486, Pentium, Pentium Pro or AMD64 CPU, including optional MMX, SSE, SSE2, and 3DNow instructions. Bochs interprets every instruction from power-up to reboot, emulates the Intel x86 CPU, a custom BIOS, and has device models for all of the standard PC peripherals: keyboard, mouse, VGA card/monitor, disks, timer chips, network card, etc. Since Bochs simulates the whole PC environment, the software running in the simulation thinks as if it is running on a real machine (hence virtual machine) and in this way, supports execution of unmodified legacy software (e.g. Operating Systems) on its virtual machines without any difficulty. No matter what the host platform is, Bochs always simulates x86 software, and thus incurs the extra overhead of instruction translation. Commercial emulators (covered in the next section) can achieve high emulation speed using a technique called ”virtualization”. However, they lose the property of portability to non-x86 platforms.

To achieve anything ”interesting” in the simulated machine, Bochs needs to interact with the operating system on the host platform. When a key is pressed in the Bochs virtual machine, a key event goes into the device model for the keyboard. When the Bochs virtual machine needs to read from the simulated hard disk, Bochs' device model for the hard disk reads from a disk image file on the host machine. When the Bochs virtual machine sends a network packet to the local network, Bochs' virtual ethernet card uses the host platform's network card to send the packet to the real world. These interactions between Bochs and the host operating system can be complicated, and in some cases be specific to the host platform.

Although Bochs is too slow a system to be used as a virtual machine technology in practice, it has several important applications that are hard to achieve using commercial emulators. It can have important use in letting people run applications in a second operating system. For example, it lets people run Windows software on a *non-x86* workstation or on an x86-Unix box. Being an open source, it can be extensively used for debugging new operating systems. For example, if your boot code for a new operating system does not seem to work, Bochs can be used to go through the memory content, CPU registers, and other relevant information to fix the bug. Writing a new device driver, understanding how the hardware devices work and interact with others, are made easy through Bochs. In industry, it is used to support legacy applications on modern hardware, and as a reference model when testing new x86-compatible hardware.

2.2 Crusoe

Transmeta's VLIW based Crusoe [3] processor comes with a dynamic x86 emulator, called ”code morphing engine”, and can execute any x86 based application on top of it. Although the initial intent was to create a simpler, smaller, and less power consuming chip, which Crusoe is, there were few compiler writers to target this new processor. Thus, with some additional hardware support, extensive caching, and other optimizations Crusoe was released with an x86 emulator on top of it. It uses 16MB system memory for use as a ”translation cache”, that stores recent results of the x86 to VLIW instruction translations for future use. The Crusoe is designed to handle the x86 ISA's precise exception semantics without constraining speculative scheduling. This is accomplished by shadowing all registers holding the x86 state. For example, if a division by zero occurs, it rolls back the effect of all the out of order and aggressively loaded instructions by copying the processor states from the shadow registers. Alias hardware also helps the Crusoe rearrange code so that data can be loaded optimally. All these techniques greatly enhance Crusoe's performance.

2.3 QEMU

QEMU [4] is a fast processor emulator that uses a portable dynamic translator. It supports two operating modes: user space only, and full system emulation. In the earlier mode, QEMU can launch Linux processes compiled for one CPU on another CPU, or for cross-compilation and cross-debugging. In the later mode, it can emulate a full system that includes a processor and several peripheral devices. It supports emulation of a number of processor architectures that includes x86, ARM, PowerPC, and Sparc, unlike Bochs that is closed tied with the x86 architecture. Like Crusoe, it uses a dynamic translation to native code for reasonable speed. In addition, its features include support for self-modifying code and precise exceptions. Both full software MMU and simulation through `mmap()` system call on the host are supported.

During dynamic translation, it converts a piece of encountered code to the host instruction set. The basic idea is to split every x86 instruction (e.g.) into fewer simpler instructions. Each simple instruction is implemented by a piece of C code and then a compile time tool takes the corresponding object file to a dynamic code generator which concatenates the simple instructions to build a function. More such tricks enable QEMU to be relatively easily portable and simple while achieving high performances. Like Crusoe, it also uses a 16MB translation cache and flushes to empty when it gets filled. It uses a basic block as a translation unit. Self-modifying code is a special challenge in x86 emulation because no instruction cache invalidation is signaled by the application when code is modified. When translated code is generated for a basic block, the corresponding host page is write protected if it is not already read-only. Then, if a write access is done to the page, Linux raises a SEGV signal. QEMU, at this point, invalidates all the translated code in the page and enables write accesses to the page, to support self-modifying code. It uses basic block chaining to accelerate most common sequences.

2.4 BIRD

BIRD [5] is an interpretation engine for x86 binaries that currently supports only x86 as the host ISA and aims to extend for other architectures as well. It exploits the similarity between the architectures and tries to execute as many instructions as possible on the native hardware. All other instructions are supported through software emulation. Apart from interpretation, it provides tools for binary analysis as well as binary rewriting that are useful in eliminating security vulnerabilities and code optimizations. It combines static as well as dynamic analysis and translation techniques for efficient emulation of x86-based programs.

Dynamo [6] is another project that has a similar aim. It uses a cache to store the so-called ”hot-traces” (sequences of frequently executed instructions), e.g. a block in a *for loop*, optimizes and executes them natively on the hardware to improve its performance.

3 Virtualization at the Hardware Abstraction Layer

Virtualization at the HAL exploits the similarity in architectures of the guest and host platforms to cut down the interpretation latency. Most of the today’s world’s commercial PC emulators use this virtualization technique on popular x86 platforms to make it efficient and its use, viable and practical. Virtualization technique helps map the virtual resources to physical resources and use the native hardware for computations in the virtual machine. When the emulated machine needs to talk to critical physical resources, the simulator takes over and multiplexes appropriately.

For such a virtualization technology to work correctly, the VM must be able to trap every privileged instruction execution and pass it to the underlying VMM to be taken care of. This is because, in a VMM environment, multiple VMs may each have an OS running that wants to issue privileged instructions and get the CPU’s attention. When a trap occurs during privileged instruction execution, rather than generating an exception and crashing, the instruction is sent to the VMM. This allows the VMM to take complete control of the machine and keep each VM isolated. The VMM then either executes the instruction on the processor, or emulates the results and returns them to the VM. However, the most popular platform, x86, is not fully-virtualizable, i.e. certain supervisor (privileged) instructions fail *silently* rather than causing a convenient trap when executed with insufficient privileges. Thus, the virtualization technique must have some workaround to pass control to the VMM when a faulting instruction executes. Most commercial emulators use techniques like *code scanning* and *dynamic instruction rewriting* to overcome such issues. In this section, we shall explore the techniques used by a number of commercial PC emulators to create correct and efficient virtualized machines and some of their features and shortcomings.

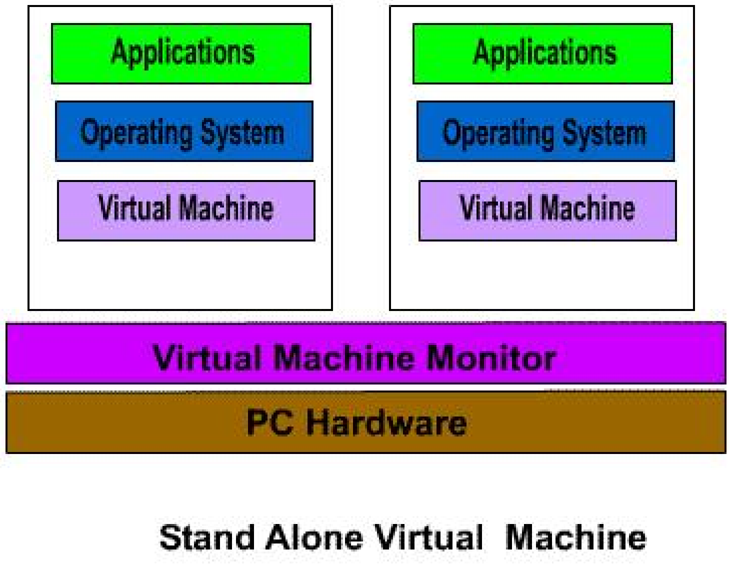


Figure 2: Stand-alone VM implementation.

3.1 VMWare

VMware [7] is an industrial strength virtual machine company with three levels of VM products: VMware Workstation, VMware GSX Server, and VMware ESX server. In this paper, we shall concentrate on the VMware Workstation product for normal PC users that is very common and a few features and the diffrences with the other ones.

VMware's VMMs can be *standalone* or *hosted*. Figure 2 shows an architecture of a standalone Virtual Machine Monitor. A standalone VMM is basically a software layer on the base hardware that lets users create one or more VMs. These are similar to operating systems, require device drivers for each hardware device, and are typically limited in hardware support. Such VMMs are typically used in servers, VMware ESX server being a prime example of such an architecture. A hosted VMM, however, runs as an application on an existing host operating system as shown in Figure 3. It can take advantage of the host operating system for memory management, processor scheduling, hardware drivers, and resource management [8]. VMware Workstation group of products use this hosted virtual machine architecture.

VMware products are targeted towards x86-based workstations and servers. Thus, it has to deal with the complications that arise as x86 is not a fully-virtualizable architecture. VMware deals with this problem by using a patent-pending technology that dynamically rewrites portions of the hosted machine code to insert traps wherever VMM intervention is required [9]. Although it solves the problem, it adds some overhead due to the translation and execution costs. VMware tries to reduce the cost by caching the results and reusing them wherever possible. Nevertheless, it again adds some caching cost that is hard to avoid.

To understand how VMware workstation is installed and run, it helps to look at the way IA32 platform works. On the intel architecture, the protection mechanism provides four privilege levels, 0 through 3

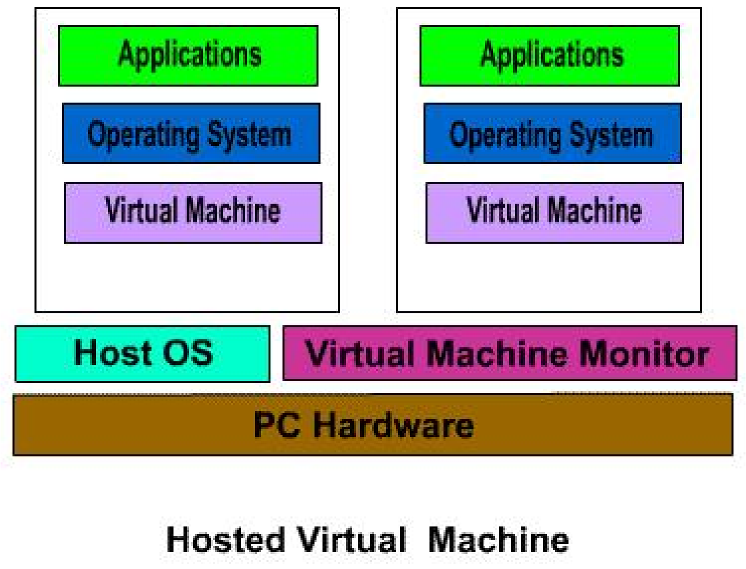


Figure 3: Hosted VM implementation.

(shown in Figure 4). These levels are also called *rings*. These protection rings exist only in *Protected Mode*[2]. According to Intel, ring 0 is meant for operating systems and kernel services, ring 1 and 2 for device dribers, and ring 3 for applications. However, in practice, most operating systems along with the device drivers run completely in ring 0 and applications in ring 3. Privileged instructions are allowed only in ring 0, and cause protection violation if executed anywhere else.

VMware Workstation has three components: the *VMX driver* and *VMM* installed in ring 0, and the *VMware application* (VMApp) in ring 3. The VMX driver is installed within the operating system to gain the high privilege levels required by the virtual machine monitor. When executed, the VMApp loads the VMM into kernel memory with the help of VMX driver, giving it the highest privilege (ring 0). The host OS, at this point, knows about the VMX driver and the VMApp, but does not know about the VMM. The machine now has two worlds: the *host world* and the *VMM world*. The VMM world can communicate directly with the processor hardware or through the VMX driver to the host world. However, every switch to the host world would require all the hardware states to be saved and restored on return, which makes switching hit the performance. The architecture of the whole system can be seen in Figure 5.

When the guest OS or any of its applications run purely computational programs, they are executed directly through the VMM in the CPU. I/O instructions, being privileged ones, are trapped by the VMM and are executed in the host world by a world switch. The I/O operations requested in the VM are translated to high-level I/O related calls and are invoked through the VMApp in the host world, and the results are communicated back to the VMM world. An example is illustrated in the Figure 6 for the network `send` and `recv` operation. This makes the overall VM run slow for I/O intensive applications.

[2]Intel processors provide Protected, Real, and Virtual-86 Modes of operation. However, most of the modern OSes such as Windows 2000 and XP run pretty much in Protected Mode.

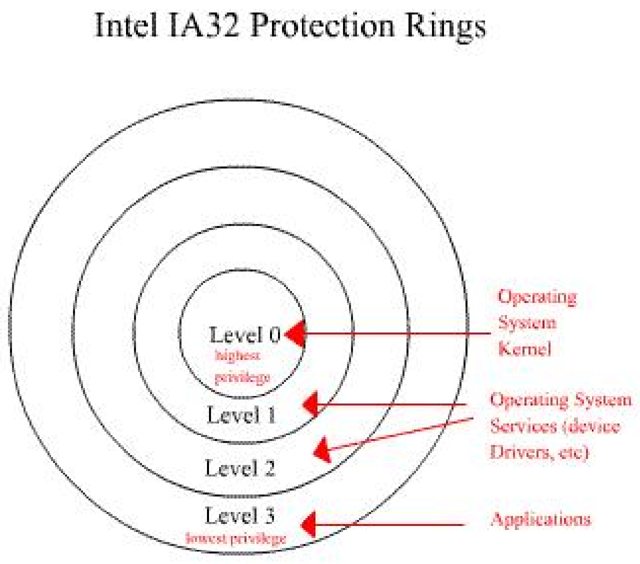


Figure 4: Intel x86 protection rings.

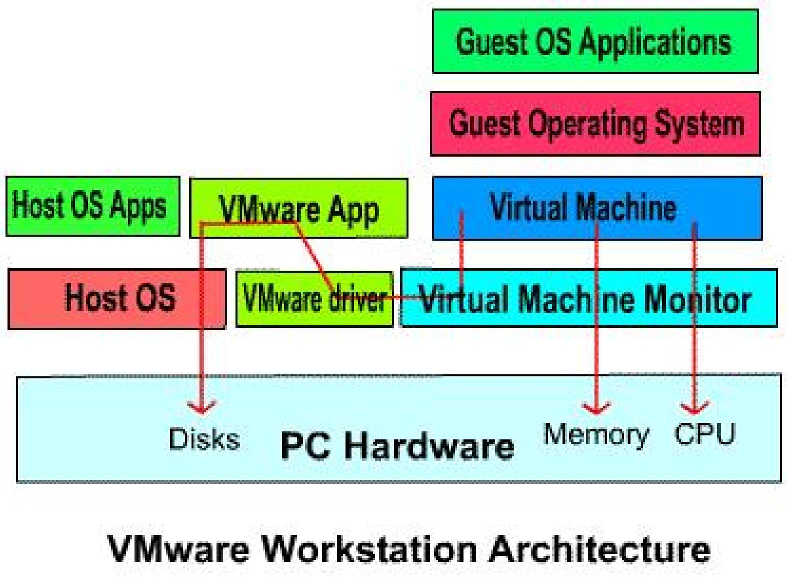


Figure 5: VM Workstation Architecture.

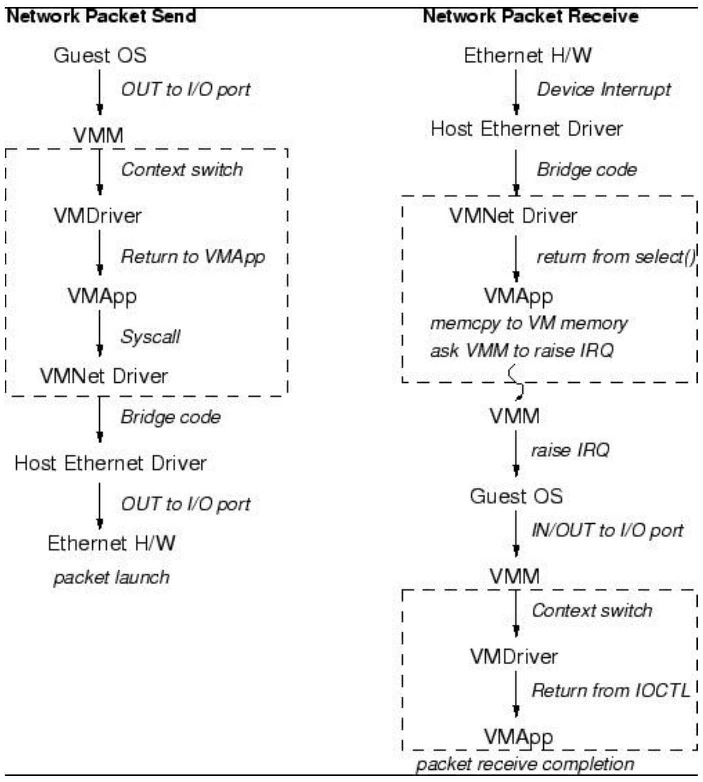


Figure 6: VMware Network Architecture.

ESX server product is installed on a bare machine without any operating system. It gives a console interface to create and configure VMs. The product typically finds use in server consolidation and web hosting. Since there is no host operating system, VMM has to handle all the I/O instructions, which necessitates the installation of all the hardware drivers and related software. It implements shadow versions of system structures such as page tables and maintains consistency with the virtual tables by trapping every instruction that attempts to update these structures [9]. Thus, there exists one extra level of mapping in the page table. The virtual pages are mapped to physical pages throught the guest operating system's page table. The physical page (often called *frame*) is then translated to the machine page by the VMM, which eventually is the correct page in physical memory. This helps the ESX server better manage the overall memory and improve the overall system performance. It uses various other techniques to increase the overall efficiency, and level of isolation to keep each VM independent from another, making it a reliable system for commercial deployment.

The newer versions of VMware Workstation come with some of the striking features. *Pointer integration* with the host desktop allows the use to move the mouse pointer seamlessly in and out of the VMware Application's display window like it happens with any other window-based application. *File sharing* allows the user to share files and folders between the host and the guest machines to help easy transfer of data from and to the virtual machines for backing up and other purposes. *Dynamic display resizing* lets the user dynamically resize the VMware Application's display window like any other window. This is not so trivial realizing the fact that every resize operation changes the screen resolution for the virtual machine.

VMware Workstation supports a variety of networking setups to help user connect to the network according to his/her convenience. It has the provision of a virtual ethernet hub that connects all the virtual ethernet adapters that exist in the VMs to create a LAN within the host computer. It also supports bridged networking through the external ethernet card connected to the real network outside the VM. Network address translation (NAT) is also supported. Among other features, it passes the host USB devices to the virtual machines that lets user connect and work with any USB device inside of a VM. However, VMware limits the maximum memory to be allocated across all the active VMs to be 4GB. Although at this time it does not really look like a limitation, it is for the world to see how things develop in the next few years.

3.2 Virtual PC

Microsoft's Virtual PC [10], recently acquired from Connectix, is a product very similar to what is offered by VMware Workstation. It is based on the Virtual Machine Monitor (VMM) architecture and lets the user create and configure one or more virtual machines. Apart from the features supported by VMware, it provides two distiguishing functionalities. It maintains an *undo disk* that lets the user easily undo some previous operations on the hard disks of a VM. This enables easy data recovery and might come handy in several circumstances. The other striking feature is *binary translation*, which it uses to provide x86 machines on Macintosh-based machines.

There are a number of shortcomings that the Virtual PC possess in terms of features when compared to VMware. Linux, FreeBSD, OpenBSD, Solaris, etc are not supported as guest OSes in Virtual PC. The Virtual PC VMs do not have support for SCSI devices, unlike VMware workstation, although some SCSI disks are recognized as IDEs by the VMs. It does not let user add or upgrade the hardware set for a VM. Once configured, it makes it impossible to change the hardware devices a VM possesses later on. Linux or other exotic operating systems are not available as host OS.

3.3 Denali

Although virtual machines provided by the likes of VMware Workstation and Microsoft Virtual PC are very efficient and practical to use supporting almost all the PC-like features with all the ease, due to design limitations it is difficult to create and use thousands of them simultaneously. The way virtualization is achieved in the VMM makes it difficult to scale it to high numbers. For example, the interrupt handling mechanism is hard to scale beyond a few active VMs, if multiplexed simultaneously. The same is the case for memory management, world switching, and so on. However, there might be legitimate reasons to have large numbers of active virtual machines for various purposes. The University of Washington's Denali project [11] tries to address this issue and come up with a new virtualization architecture to support thousands of simultaneous machines, which they call `Lightweight Virtual Machines`. Using a technique, called *paravirtualization*, it tries to increase the scalability and performance of the Virtual Machines without too much of implementation complexity.

The *paravirtualization* technique modifies the traditional virtualization architecture for new customized guest operating systems (unlike VMware Workstation, that supports legacy OSes) to obtain extra performance, and high scalability. This new architecture comes up with new interfaces for the customized guest operating systems. The paravirtualized architecture provides modified architectural features that makes the implementation of guest OSes simple yet versatile. *Virtual instructions*, equivalent to system calls in traditional architecture, expose rich and simple instructions for the upper layer by grouping and optimizing commonly used instructions. The new architecture exposes a set of *virtual registers* for ease of data transfer between the virtualization layer and the virtual machines. It also provides a simplified architectural interface to be exported by the virtual I/O devices. Among other things, it supports a modified interrupt delivery, and does not support the virtual memory concept. All these modifications are incorporated aiming at a simpler implementation with low overhead to make the overall system scalable and the VMs lighter.

3.4 Xen

The discussions so far have been concentrating on full virtualization, where applications and the operating system within a VM live in a complete virtual world with no knowledge of the real machine whatsoever. Although many a times this has been the goal of virtualization, there may be cases where an application or the operating system running within a VM might desire to see both the real as well as virtual resources and use the information to its benefit. For example, seeing both real and virtual time might help the guest OS better support time-sensitive tasks and come up with good round trip time (RTT) estimates for handling TCP timeouts. Likewise, seeing real machine addresses might help improve performance by using superpages [12] and page coloring [13]. This apart, a full virtualization is always tricky and cumbersome when implemented on x86 due to its inherent problem of not being a virtualizable architecture. X86, being an uncooperative machine architecture, makes the task of achieving high performance with a strong resource isolation in virtualization very difficult. In addition, completely hiding the effects of resource virtualization from guest OSes risks both correctness and performance.

All these along with issues like QoS, security, and denial of service motivate the researchers in University of Cambridge come up with a modified architecture for virtualization, called Xen [9]. Xen exports a paravirtualized architecture in each of its VMs to maximize performance and resource isolation yet maintaining the same application binary interface (ABI) as commodity operating systems. Although it does require the operating systems to be ported, the porting effort is kept as low as possible. It aims at supporting around a hundred VM instances within a single physical machine within a reasonable performance hit.

Although Denali [11] uses a paravirtualized architecture for more or less the same purposes, they both have diffrent targets. Denali is designed to supposed thousands of virtual machines running network services, the vast majority of which are small-scale and unpopular. Since the applications used in Denali are customized ones, the paravirtualization architecture does not have to guarantee the *ABI compatibility* and thus can elide certain architectural features from its VM interface. No support for x86 segmentation is one such example although ABIs in most of the OSes including Windows XP, Linux, and NetBSD export this feature. Denali VMs are designed with the aim of hosting a *single application, single-user unprotected guest OS* (e.g. Ilwaco) and thus does not have support for virtual memory which is a common feature in almost all the modern OSes. Denali VMM performs all the paging work to and from the disks that are vulnerable to thrashing should there be any malicious VM, while Xen relies on the guest OS to do all the paging. Denali virtualizes *namespaces*, whereas Xen believes *"secure access control"* within the hypervisor is sufficient to ensure protection while making physical resources directly accessible to guest OSes .

The paravirtualization exports a new virtual machine interface that aims at improving the performance and scalability as compared to the other commercial VMMs. A new *lightweight event mechanism* replaces the traditional hardware interrupts in the x86 architecture for both CPU as well as the device I/O. This greatly improves the performance and scales to great numbers. *Asynchronous I/O rings* are used for simple and efficient data transfers between the VMs and the hypervisor (Xen's VMM or simply Xen, in short). For security purposes, *descriptor tables* for exception handlers are registered with Xen by each of the VMs and aside the page faults, the handlers remain the same. Guest OS may install 'fast' handler for system calls, allowing direct calls from an application into its guest OS avoiding the indirection through Xen on every call.

For efficient page table and TLB management, Xen exists in a 64MB section of every address space. This avoids a TLB flush when entering and leaving the hypervisor. However, this also means a restricted segmentation that disallows installation of fully-privileged segment descriptors and the top end of the linear address space can not be overlapped. Guest OSes have the direct access to hardware page tables, however, updates are batched and validated by Xen. This allows Xen to implement a secure but efficient memory management technique when compared to VMware where every update to the page table is trapped by VMM and updated. In particular, when a new process is created through fork(), the number of updates is enormous which might hit the performance badly in VMware. Using batched updates, as in Xen, helps it a lot. Each guest OS is provided a timer interface and is aware of both 'real' and 'virtual' time. In this way, even though the hypervisor exports a modified VM interface as compared to a traditional x86 architecture, it tries to build a more robust architecture that preserves all the features that are of importance to application binaries yet keeping the porting effort for the guest OSes minimal.

3.5 Plex86

Plex86 [14] project works toward an open-source x86 simulator with virtualization. It uses the virtualization technique to improve the efficiency of a virtual machine such as Bochs. Virtualization is a technique that is used to take advantage of the hardware similarity of the guest and the host machine to allow large portions of the simulation to take place at the native hardware speed. When the simulated machine talks to the hardware, or enters certain privileged modes (such as the ”kernel mode”), the simulator takes control and simulates the code in software at a much slower speed, as the Bochs does. Thus, virtualization helps Plex86 run much faster compared to Bochs, however, the portability is lost. Another version of Plex86 is being developed just to create a partial virtual machine that is able to support Linux and which runs much faster than the full scale virtual machine.

3.6 User-mode Linux

User-mode Linux [15], or UML, is an open source project that lets the user run Linux on top of Linux. Basically, it gives a virtual machine on which a Linux version can execute as it does on a physical machine, and everything implemented in the user-level. Unlike previous ones that use the VMM right on the base hardware, this uses a different implementation being on top of the operating system and in the user-space. Implementation aside, the abstraction level still remains more or less similar to the previous ones. It lets the user configure virtual hardware resources that would be available for the guest Linux kernel. Since everything runs in the user-level, safety is assured. Its hardware support comes in the form of virtual devices that make use of the physical resources. Among the devices supported are, block devices, consoles, serial lines, network devices, SCSI devices, USB, Sound, and so on. The UML runs its own scheduler independent of the host scheduler, runs is own virtual memory system, and basically supports anything that is not hardware-specific. It also supports `SMP` and `highmem`. The virtual console driver implementation lets the user attach it to a number of interfaces available on the host: file descriptors, ptys, ttys, pts devices, and xterms.

Implementation of UML involves a port of the Linux kernel to the Linux system call interface rather than to a hardware interface. In this regard, the virtual machine and the guest Linux kernel are tighty coupled. Executing totally in the user space, the major challenge it faces is to be able to intercept the system calls in the virtual kernel, as they would naturally go to the real host kernel. Using the Linux `ptrace` facility to track system calls, it diverts the system calls made by processes running within the Virtual Machine to the user space kernel to execute them. Similarly, traps are implemented through Linux signals. Kernel and the processes within the VM share the same address space; and conflicts with process memory are avoided by placing the kernel text and data in areas that processes are not likely to use. Each process in the virtual machine gets its process in the host kernel. In order for the virtual kernel’s data to be shared across all the processes in the VM, its data segment is copied into a file, and the file is mapped shared to all the processes. Using such tricks, it implements, that too within a reasonable overhead, the user-space virtual machine.

3.7 Cooperative Linux

CoLinux [16], as it is often called, is a variation of User-mode Linux. It is a port of the Linux kernel that allows it to run as an unprivileged lightweight virtual machine in kernel mode, on top of another OS kernel.

It allows Linux to run under any OS that supports loading drivers, such as Windows, with some minor porting effort.

`Cooperative Linux` works in parallel with the host kernel. In such a setup, each kernel has its own complete CPU context and address spce, and decides when to give the control back to its partner. However, only one of the two kernels has the control on physical hardware (the host kernel), and the other (guest kernel) is provided only with virtual hardware abstraction. The only requirement on the host kernel is that it should allow to load the colinux portable driver to run in ring 0 and allocate memory. The colinux VM uses only one host process, called the `Super Process`, for itself and its processes. It uses the portable driver to switch the context to the host kernel and back as well as to load the kernel from a file during startup. Using a forwarding technique, colinux handles the interrupts by making use of its own code and the context switches appropriately. The overall performance is comparable to UML.

4 Virtualization at the OS level

Hardware-level virtual machines tend to possess properties like high degree of isolation (both from other VMs as well as from the underlying physical machine), acceptance of the concept (people see it as a normal machine, that they are used to), support for different OSes and applications without requiring to reboot or going through the complicated dual-boot setup procedure, low risk, and easy maintenance. Since a virtual machine at this level gives access to a raw computer, the user needs to spend a good amount of time installing and administering the virtual computer before he can start thinking to test or run his required application. This includes, operating system installation, application suites installation, network setup, and so on. If the required OS is same as the one on the physical machine (and the user is using the VM for security/sandboxing purposes, as a playground, or anything that warrants an extra machine), the user basically ends up duplicating most of the effort he/she has already invested in setting up the physical machine. In this section, we explore virtualization at a higher level in the machine stack (see Figure 1) that addresses this issue of minimizing the redundancy of the OS requirement in VMs described above. The virtual machines at this level share the hardware as well as the operating system on the physical machine and use a virtualization layer (similar to the VMMs in VMware) on top of the OS to present multiple independent and isolated machines to the user.

An operating environment for an application consists of the operating system, user-level libraries, other applications, some system specific data structures, a file system, and other environmental settings. If all of these are kept intact, an application would find it hard to notice any difference from that of a real environment. This is the key idea behind all the OS-level virtualization techniques, where virtualization layer above the OS produces a partition per virtual machine on demand that is a replica of the operating environment on the physical machine. With a careful partitioning and multiplexing technique, each VM can be able to export a full operating environment and fairly isolated from one another and from the underlying physical machine. In this section, we go though some examples that use this abstraction in their virtualizing software and the techniques they usually employ to achieve the target.

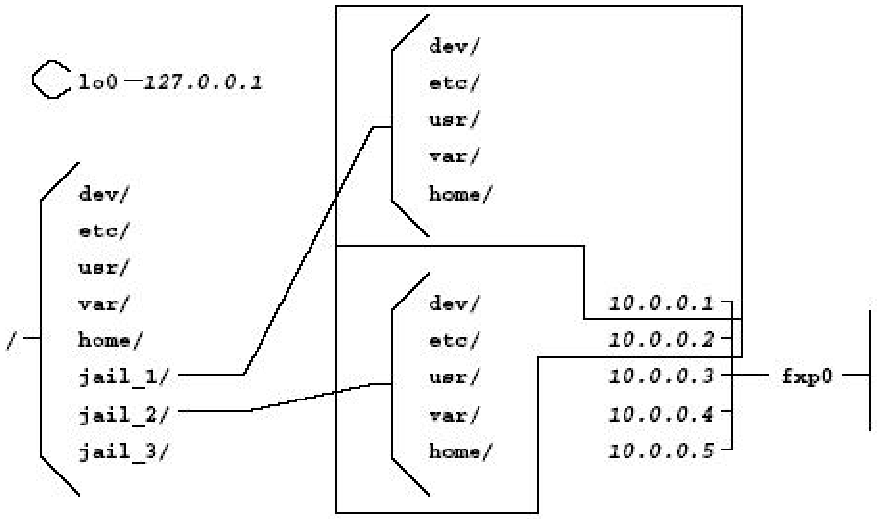


Figure 7: Machine with two configured jails.

4.1 Jail

"Jail" [17] is a FreeBSD based virtualization software that provides the ability to partition an operating system environment, while maintaining the simplicity of UNIX "root" model. The environment captured within a jail are typical system resources and data structures such as processes, file system, network resources, etc. In *Jail*, users with privilege find the scope of their requests to be limited to the *Jail*, allowing system administrators to delegate management capabilities to each virtual machine environment.

A process in a partition is referred to as "in jail". When the system is booted up after a fresh install, no processes will be in jail. When a process is placed in a jail, all of its descendants after the jail creation, along with itself, remain within the jail. A process *may not* belong to more than one jail. Jails are created by a privileged process when it invokes a special system call `jail(2)`. Each call to jail(2) creates a new jail; the only way for a new process to enter the jail is by inheriting access to the jail from another process already in that jail. Processes may never leave the jail they created, or were created in. Figure 7 shows a schematic diagram of a machine with two configured jails.

4.1.1 Approach

There are a number of restrictions involved in jail memberships. Access to only a subtree of the file system, the ability to bind network resources to a specific IP address, a curtailed ability to manipulate system resources and perform privileged operations, a limited ability to interact with other processes (to only processes inside the same jail) are examples of some restrictions. Jail uses the existing Unix `chroot(2)` facility to limit access to the file system name-space for the jailed processes. Each jail is configured to bind to a particular file system subtree at the time of creation. Files outside this subtree are unaddressable by the jailed processes and thus can not be manipulated. All the mechanisms of breaking out of `chroot(2)` are blocked.

As jailed processes can only be bound to a particular IP address (specific to the jail), attempts to bind to all IP addresses (INADDR_ANY) are redirected to the individual jail-specific IP address. Processes in the jail may not make use of any of the IP addresses other than that of the jail. This may restrict the kind of network services that may be offered in a jailed environment. Network functionalities of some of the privileged calls are restricted or totally disabled depending on its type. In particular, facilities that would allow ”spoofing” of IP addresses or disruptive traffic to be generated are all disabled.

Processes running without root privileges experience hardly any difference between a jailed and an unjailed environment as long as they try not to interact with other processes. However, the interactions with other processes are limited. They can not interact or even verify the existence of processes that lie outside of the jail. Connecting to those processes via debuggers, delivering signals, and being able to see them in the sysctl or process file system are also disallowed. Each jail, in theory, represents an abstract machine; hence the use of covert channels or communication mechanisms via accepted interfaces like sockets are not prevented. Processes running with root privileges face restrictions in terms of what they can achieve using the privilege calls. Privileged calls that try to create device nodes, for example, are not allowed inside a jail. These kind of calls result in an ”access error” when invoked. An attempt to bind to a reserved port on all available IP addresses would result in binding the port only to the jail’s IP address. All file system related system calls would succeed as long as the file is accessible through the jail file system name-space.

4.1.2 Implementation and Jail Management

The jail(2) system call is implemented as a non-optional system call in FreeBSD. The implementation of the system call is straightforward: a data structure is allocated and populated with the arguments provided. The data structure is attached to the current process’ struct proc, its reference count set to one and a call to the `chroot(2)` syscall implementation completes the task. Hooks in the code implementing process creation and destruction maintains the reference count on the data structure and free it when the reference count becomes zero. There is no way to attach a process to an existing jail if it was not created from its inside. Process visibility is controlled by modifying the way processes are reported by `procfs` file system and the `sysctl` tree. The code which manages `"protocol control blocks"` is modified to use just the jail-specific IP address and not the address provided by the process. Loop-back interface, or the 127.0.0.1 IP address, is also handled apropriately by replacing it with the jail’s IP address. Virtual terminal or the pty driver code is also modified to make sure multiple jails do not get access to a particular virtual terminal.

Though not enforced, the expected configuration creates a complete FreeBSD installation for each jail. This includes copies of all relevant system binaries, data files, and its own `/etc` directory. This maintains complete independence among the jails. However, to improve storage efficiency, a fair number of the binaries in the system tree may be deleted, as they are not relevant in a jail environment. This includes the kernel, boot loader, and related files, as well as hardware and network configuration tools. The jailed virtual machines are generally bound to an IP address configured using the normal IP alias mechanism; thus, these jail IP addresses are also accessible to host environment applications to use. If the accessibility of some host applications in the jail environment is not desirable, it is necessary to configure those applications to only listen on appropriate addresses.

4.2 Linux Kernel-mode Virtualization

The virtual environment system [18] for Linux is another similar work. It aims to allow administrators to run several independent application environments on a single computer with proper boundaries between them. This aims to help application hosting and improve system security.

The ”Virtual Environment” (VE) consists of an isolated group of processes with its own file system root, init, startup scripts, and so on. It also allows administration of the environment from the inside, like mounting file systems, changing network configurations, etc with the obvious restriction of keeping changes within the VE. Unlike jail, it provides more sophisticated access control measures. For example, it can provide a user with a *subset* of administrative rights inside the environment. It also tries to avoid the ugly and unnatural relationships between file system roots and IP addresses that exist in the ”jail” implementation and interface.

It uses a name-space separation method to implement the environment boundaries. All environment specific objects are marked by an identifier, VE-id, and the searching algorithms are modified to expose only objects with VE-id matching the VE-id of the calling process. Some example objects are *task_struct*, *sockets*, and *skbuf*. The very first process spawned inside a VE is handled specially. That becomes the ’init’ process for the VE, with PID 1, which handles the orphans. Network changes ensure the processes can only see IP packets destined to the IP address of the VE. The file system plans to allow combine/union mount of read-only template tree and a private VE tree.

4.3 Ensim

Ensim’s Virtual Private Server [19, 20] (VPS) employs a similar virtualization technique for server consolidation, cost reduction, and to increase the efficiency in the way web sites are managed and sold. Ensim VPS ”virtualizes” a server’s native operating system so that it can be partitioned into isolated computing environments called virtual private servers. These virtual private servers operate independently of each other, just like a dedicated server. To the operating system, an Ensim VPS appears to be an application, and to applications, it’s indistinguishable from the native operating system. As a result, an Ensim VPS appears and operates as a physical server for the users. Figure 8 shows the architecture of the Ensim VPS.

The implementation is much stronger than the previous ones as it also lets the administrator provision each VPS with a desired allocation of hardware resources such as memory, disk space, CPU, and network bandwidth. The virtualization technology also supports adjustment of the resources and, if required, movement of the VPs to another physical machine maintaining a total transparency. A centralized Ensim ServerXchange accomplishes cross machine transfers seamlessly.

5 Virtualization at the Programming Language Level

A traditional machine is one that executes a set of instructions as supported by the Instruction Set Architecture (ISA). With this abstraction, operating systems and user-level programs all execute on a machine

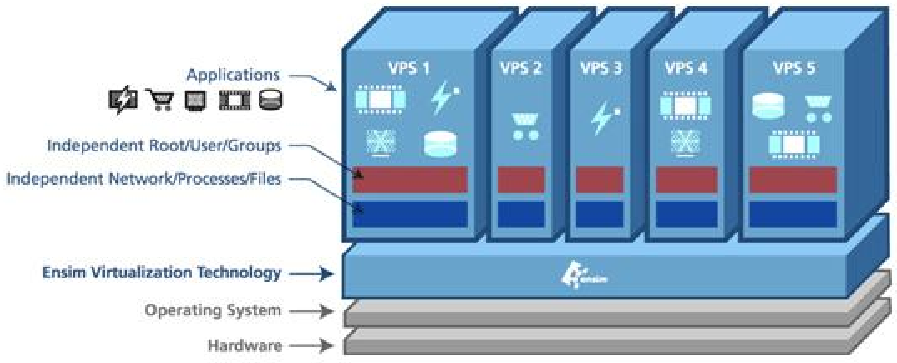

Applications
Independent Root/User/Groups
Independent Network/Processes/Files
Ensim Virtualization Technology
Operating System
Hardware
VPS 1
VPS 2
VPS 3
VPS 4
VPS 5

Figure 8: Ensim Virtual Private Server Architecture.

like applications for the machine. Hardware manipulations are dealt with either by special I/O instructions (I/O mapped), or by mapping a chunk of memory to the I/O and then manipulating the memory (memory mapped). So ultimately, it is a block or sequence of instructions that constitute the application. A new level of abstraction for virtual machine came to the limelight with the arrival of Java Virtual Machine (JVM). Following this, there were a few more, but the abstraction level was the same. The idea is to be able to create a virtual machine at the application-level than can behave like a machine to a set of applications, just like any other machine. It supports a new self-defined set of instructions (java byte codes for JVM). Such VMs pose little security threat to the system while letting the user play with it by running applications like he would on physical machines. Like a normal machine, it has to be able to provide an operating environment to its applications either by hosting a commercial operating system, or by coming up with its own environment. In this section, we explore a few such cases.

5.1 Java Virtual Machine

Chances are, everything you know about Java technology is only a few years old. There's a good reason for that: On May 23, 1998 the technology officially celebrated its third birthday. The motivation for the technology came way back in 1995 from the idea of designing a processor-independent language that could run on a wide range of platforms and appliances. The project started with a new language ”Oak” designed for the device *7 (called StarSeven) by James Gosling and later developed into what is now known as ”Java”. The back-end for such a language implements an interpreter of a new instruction set defined by the Java people, and is made available for all the popular platforms including Windows, Unix, Mac, and other OSes. In this way, a universal deployment of such a back-end made Java a platform-independent language, and the back-end so popular, that is called the Java Virtual Machine [21], or in short JVM.

JVM is a virtual machine that runs Java byte code. This code is most often generated by Java compilers, although the JVM has also been targeted by compilers of other languages. Programs intended to run on a JVM must be compiled into a standardized portable binary format comprised of Java byte code (synonymous to instructions), which typically comes in the form of `.class` files. This binary is then executed by the JVM runtime which carries out emulation of the JVM instruction set by interpreting it or by applying a just-in-time Compiler (JIT). The JVM, in addition to providing the instruction set interpreter, also provides the operating environment (that OS typically provides in a native system) for the Java byte codes. Thus the Java platform is a combination of a virtual machine along with an operating environment (JRE). The virtual machine is eventually implemented in some native language and can afford to be more flexible than traditional machines. It adds some extra computation to add features like byte code verification, structured exception handling (SEH), garbage collections, and so on. Being in the application layer, it has much more control over these implementation than system implementations, like SEH in Windows family of OSes.

JVM is a stack-based architecture and supports threads. Each thread has its own program counter and imaginary register set (registers supported by the Virtual Machine). These instructions, on the runtime, are mapped to a set of real instructions that are to be executed natively. Code verification also ensures that arbitrary bit patterns cannot get used as an address. Memory protection is achieved without the need for an MMU. Thus, JVM is an efficient way of getting memory protection on simple silicon that has no MMU. The JVM supports instructions like load/store, arithmatic, type conversion, object creation/manipulation, push/pop, branches, call/ret, and exception throws. However, more than the emulation of the byte code is the complication involved in getting a compatible and efficient implementation of the map of Java core API to the host OS.

The ”virtual hardware” of the Java Virtual Machine can be divided into four basic parts: the registers, the stack, the garbage-collected heap, and the method area. These parts are abstract, just like the machine they compose, but they must exist in some form in every JVM implementation. JVM supports addresses upto 4GB of memory with its 32-bit addressng scheme and uses 32-bit virtual registers. Depending on the particular JVM implementation, the stack, garbage-collected heap, and the method area reside at some well-defined places within this memory. JVM supports a small number of primitive data types: byte (8 bits), short (16 bits), int (32 bits), long (64 bits), float (32 bits), double (64 bits), char (16 bits), and object handle (32 bits). Apart from the program counter, it uses three registers to manage the stack: optop register, frame register, and vars register. These point to various places within the stack of the executing method.

Method area is similar to the text area in an x86 machine; it contains the byte code to be executed and the program counter always points to some byte in this area. The program counter is similar to PC and advances as execution proceeds. The stack is used to store the parameters and results of the methods, and to keep the state of each method invocation. The *vars* register point to the local variables section containing all the local variables in the method. *Frame* register points to the execution environment within the stack that maintains the operations of the stack. Finally, the *optop* register points to the top of the stack where the operands and results are placed.

Such a virtual machine architecture allows very fine-grained control over the actions that code within the machine is permitted to take. This allows safe execution of untrusted code from remote sources, a model used most famously by Java applets. Security, sandboxing, easy debugging, platform-independence are a few very important features of such a virtualization setup. Since all the hardware devices are below the JVM layer, it has access to everything in the system and virtually do everything that a normal application can do. Thus, Java programs are equally powerful when compared with any other programs with traditional languages such as C and c++.

5.2 Microsoft .NET CLI

The common language runtime (CLR) in the .NET framework is the microsoft specific implementation of the common language infrastructure [22] (CLI) specification. The CLR could be assumed to be Microsoft's equivalent of JVM. The CLI specification is an international standard for creating development and execution environments in which languages and libraries work together seamlessly. This, however, is much stronger than JVM as it integrates data and services into the framework too. It supports interfaces, classes, objects, as the Java framework does. It also defines a common library interface to all the languages that use the platform. Similar to JVM, the source-codes are translated to an intermediate representation by compiler, which eventually is run by the CLR.

5.3 Parrot

Parrot [23] is a virtual machine designed to execute bytecode for interpreted languages efficiently. Parrot will be the target for the Perl 6 compiler. There is already a partial Perl 6 compiler as well as compilers in various stages of completion for a wide range of other languages.

6 Virtualization at the Library Level

In almost all of the systems, applications are programmed using a set of APIs exported by a group of user-level library implementations. Such libraries are designed to hide the operating system related nitty-gritty details to keep it simpler for normal programmers. However, this gives a new opportunity to the virtualization community.

Examples discussed in this section work above the operating system layer and produce a different virtual environment, so much so that they can expose a different binary interface altother. In other words, virtualization techniques are used to implement a different application binary interface (ABI) and/or a different application programming interface (API) using the underlying system. Such techniques could well be said to do the work of `ABI/API emulation`. The following few sub-sections discuss this in detail.

6.1 WINE

Wine [1] is often used as a recursive acronym, standing for ”Wine Is Not an Emulator”. It is also known to be used for ”Windows Emulator”. In a way, both meanings are correct, only seen from different perspectives. The first meaning suggests Wine is not a virtual machine, it does not emulate a processor, and one is not supposed to install an operating system or device drivers on top of it; rather, Wine is an implementation of the Windows API, and can be used as a library to port Windows applications to Unix. The second meaning suggests that Wine does look like Windows to the Windows binaries (.EXE files) and emulates its behaviour very closely to that of Windows. In other words, Wine is a virtualization layer on top of X and Unix to export the Windows API/ABI, thus letting the Windows binaries run on top of it.

	DOS (.COM or .EXE)	**Win16 (NE)**	**Win32 (PE)**	WineLib
Multitasking	Only one application at a time (except for TSR)	Cooperative	Preemptive	Preemptive
Address space	1 MB of memory, where each application is loaded and unloaded	All 16-bit apps share a single address space, protected mode	Each app has its own address space. Requires MMU support from CPU	Each app has its own address space. Requires MMU support from CPU
Windows API	No Windows API but the DOS API (int 21h traps)	Calls the 16-bit Windows API	Calls the 32-bit Windows API	Calls the 32-bit Windows API, possibly the Unix APIs
Code (CPU level)	Only available on x86 in real mode. Code and data are in segmented forms, with 16-bit offsets. Processor is in real mode	Only available on IA-32, code and data are in segmented forms, with 16-bit offsets. Processor in protected mode	Available (with NT) on several CPUs, including IA-32. On IA-32 uses a flat memory model with 32 bit offsets (hence the 32 bit name)	Flat model, with 32-bit addresses
Multi-threading	Not available	Not available	Available	Available, but must use the Win32 APIs for threading and synchronization, not Unix

Table 1: Wine Executables

6.1.1 Executables

Wine's task is to run Windows executables under non-Windows operating systems. It supports a varierty of Windows executables:

- **DOS Executable:** The older programs that use the DOS format, e.g. .com, .exe (MZ execuables)
- **Windows NE:** Also called 16-bit executable, they were the native processes run in Windows 2.x and 3.x versions
- **Windows PE:** Indoduced in Windows 95 and later became the native formats for all the new Windows versions, it stands for Portable Executable, and still supports 16-bit applications. The word ”portable” says that the format remains the same across different CPU architectures although the code may be different.
- **WineLib Executable:** These are the applications written in Windows API, but compiled as Unix executables, particularly useful when a Windows application's sources are available that can benefit from the optimizations of the Wine implementation. Wine provides tools for such cases.

Table 1 summarizes the differences in these executables.

Each Win32 process is launched as a separate process by Wine, which eventually gets translated to a Unix process. However, due to their cooperative nature (unlike any modern OS' processes), Win16 tasks are handled differently. They are run as different intersynchronized Unix-threads in the same dedicated Wine process; this Wine process is commonly known as a WOW process (Windows on Windows), similar to NT. Synchronization among various Win16 tasks are done using a mutex variable; the running task holds it and passes to the other when it wishes to let that run (cooperative).

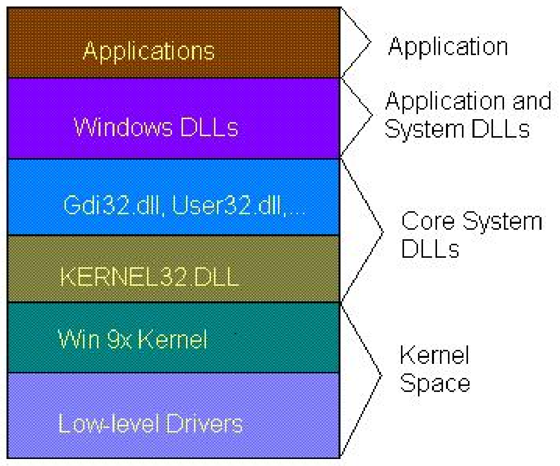


Figure 9: Windows 9x Architecture.

6.1.2 Windows Architectures

Figure 9 shows the architecture of the Windows 9x systems (e.g. Windows 95). Like a typical system, it builds user-level system-specific libraries to expose kernel features at the user-level. All the graphics related routines were designed to be in the user-level (GDI32.DLL). User32.DLL implements the core windows UI at the user-level. Applications have the free-hand of installing any library for its own use. Windows 9x family of systems does not have any concept of subsystems.

The design was substantially changed in the next series of releases (from NT onwards) to incorporate a native library (NTDLL.DLL) that hid several kernel and executive features from other system libraries. It presented the only way through which an application or any other libraries can get access to system services (equivalent to syscalls in Unix). This along with certain added features in the kernel improved the security of the system as a whole. It also planned to support multiple APIs by introducing a subsystem concept. Apart from Windows (that is implemented by Win32 subsystem), POSIX, OS/2 APIs were also supported. Figure 10 depicts this changed architecture in the NT family of systems that includes all the popular OSes like Windows NT, Windows 2000, Windows XP, and Windows 2003 server. Unlike Windows 9x that roots its architecture in 16 bit systems, Windows NT family is a true 32 bit system. The driver model is also different in these two architectures.

6.1.3 Wine Architecture

The Wine implementation targets the Windows NT applications and hence closely follows its architecture. The 16 bit support is implemented in a single 32 bit Windows executable (remember WOW), and not as a subsystem. Figure 11 shows the global picture that implements Windows on top of Unix.

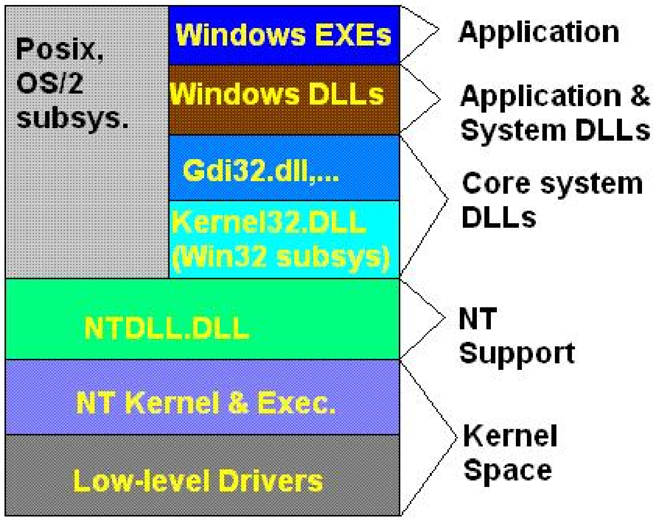


Figure 10: Windows NT Architecture.

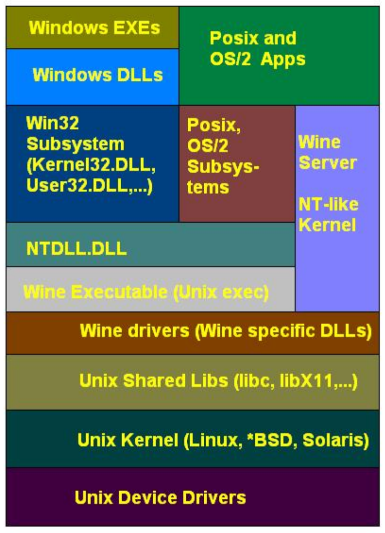

Windows EXEs
Windows DLLs
Posix and OS/2 Apps
Win32 Subsystem (Kernel32.DLL, User32.DLL,...)
Posix, OS/2 Subsys-tems
Wine Server
NT-like Kernel
NTDLL.DLL
Wine Executable (Unix exec)
Wine drivers (Wine specific DLLs)
Unix Shared Libs (libc, libX11,...)
Unix Kernel (Linux, *BSD, Solaris)
Unix Device Drivers

Figure 11: Wine Architecture.

Several design decisions that went into the Wine implementation are as follows:

- Wine closely follows the Windows NT architecture; so it is important to replace the NTDLL.DLL by its corresponding implementation in Wine (over Unix) that is correct and captures all the aspects of NT.
- In order to be able to provide the correct and full Windows API to the Wine user, the core subsystem DLLs, viz. User32.DLL, Kernel32.DLL, and Gdi32.DLL need to have their own complete implementation in Wine. Since there are instances where these subsystem DLLs use the kernel knowledge and features of NT, only NTDLL.DLL implementation is not sufficient for Wine.
- All other system- and user-level libraries are layered on top the above core DLLs and would be automatically taken care of, i.e. no Wine-specific implementation is required for such libraries.
- The `Wine server` provides the backbone for the implementation of the core DLLs. It mainly implementents inter-process synchronization and object sharing. It can be seen, from a functional point of view, as a NT kernel, although the APIs and protocols used between Wine's DLL and the Wine server are Wine specific
- Being on top of Unix, Wine uses the Unix device drivers to access the hardware devices on the machine. However, wherever necessary Wine has to provide a proxy driver (that looks like a Windows driver to the Windows libraries) to access the hardware that acts as a proxy to the real Unix driver. One such example is a graphics driver that uses the X11 or SDL driver as a proxy.

6.1.4 Wine Server

The `Wine server` is like a central management entity of the Wine implementation that helps implement the NT core. It provides Inter-Process Communication (IPC), synchronizaton, and process/thread management. When started, it creates a Unix socket for the current host in a well-defined location — all wine processes (which basically represent Windows processes) launched later connect to the Wine server using this socket. The message-queues that every thread in a Windows system possess, are implemented using this server. Every thread in each Wine process has its own request buffer, which is shared with the Wine server. When a thread needs to synchronize or communicate with any other thread or process, it fills out its request buffer, then writes a command code through the socket. The Wine server handles the command as appropriate, while the client thread waits for a reply. The synchronization primitives like *WaitFor* are handled by the server by making the thread wait until the condition has been satisfied.

Wine server is a separate Unix process that is built on a large `poll()` loop that alerts the server when some important event happens, such as the receipt of a command from client, fulfilment of some wait condition, etc. Because the Wine server needs to manage processes, threads, shared handles, synchronization, and any related issues, all the clients' Win32 objects are also managed by the Wine server, and the clients must send requests to the Wine server whenever they need to know any Win32 object handle's associated Unix file descriptor.

6.1.5 Loading an Executable

Loading a windows binary is not a tough task. But the real complexity lies in creating the proper load environment where the binary expects all those DLLs and entry points, it imports, to be present and function as expected.

DLLs are implementated by Wine as a Unix shared library and contains, including the DLL code, some more information like DLL resources and a DLL descriptor specific to Wine. When the DLL is instantiated, the descriptor is used to create an in-memory PE header that provides information regarding the DLL, such as its entry points, resources, sections, debug information, etc. The DLL descriptors are automatically registered when shared libraries are loaded by putting a call to the registration routine inside the shared library constructor.

When an application module wants to import a DLL, Wine does the following to search for it:

- It looks through its list of registered DLLs, i.e. loaded DLLs and the loaded shared libraries which have a registered DLL descriptor.
- If not found, Wine looks for it on the disk, building the shared library name from the DLL module name (by appending .so) in some environment specified directory.
- If still not found, it looks for real Windows .DLL file on disk, load it, and look through its imports etc.
- Failing all, it declares not found (an error condition).

The DLLs are mapped to the memory by traditional Unix `dlopen()` call. It relies on the dynamic loading features of the Unix shared libraries to relocate the DLLs if needed.

6.1.6 Wine/Windows DLLs

Wine provides full implementation specific to Wine for many Windows DLLs; these new DLLs are called as *Built-in DLLs*. For this reason, the true Windows DLLs, in this context, are often called *Native DLLs*. Native DLLs guarantee 100would maintain a virtually perfect and Windows 95-like look for window borders, dialog controls, etc. Using the built-in Wine version of this library, on the other hand, would produce a display that does not precisely mimic that of Windows 95. However, not every native DLL would work if its required features and imported DLLs are not fully available or not implemented in Wine. Sometimes, if the Wine built-in DLL provides some extra features that exploit Unix/X facilities, the build-in DLL might outperform the native ones. If, for a particular version, both the native and built-in versions are available, load oder decides which one to choose. By default, a fully implementd built-in version takes precedence over a native one.

6.1.7 Memory Management

Every Win32 process in Wine has its own dedicated native process on the host system, and therefore its own address space. The virtual memory layout in Windows (and therefore Wine), and Linux are shown in the table 2. Fortunately, they overlap in almost all the places, except the shared area in Win9x. This is taken care of by creating and managing a shared heap by Wine at the address 0x80000000.

Address	**Windows 9x**	**Windows NT**	**Linux**
00000000-7fffffff	User	User	User
80000000-bfffffff	Shared	User	User
c0000000-ffffffff	Kernel	Kernel	Kernel

Table 2: Memory Layout in Windows and Linux

All the DLLs, and the EXE itself are loaded as they would in Windows. Work is going on to handle the Ingo-Molnars 4G/4G VM split patch for Linux, which gives the Linux kernel its own address space of 4G — this forces a switch in the address space at every system call.

6.1.8 Processes and Threads

Windows provides a richer API to create, manage, and destroy threads and many thread-related primitives. But Linux, unfortunately does not have all these. Therefore, most of the thread realated APIs are implemented in the user-space and wherever possible, Win32 threads are mapped to native Linux pthreads for better performance.

To start a process, Wine first starts another executable to check the threading model of the underlying OS (Win9x or WinNT) of real executable and uses the corresponding Wine loader to start it. First, the ntdll.dll.so (the built-in NTDLL.DLL implementation) is loaded into memory using the standard dynamic library loader. Once loaded, ntdll does three things: creates a PEB and TEB (process/thread environment block), sets up the connection to the Wine server, and creates the process heap. This follows by the load of Kernel32 built-in dll and a wine specific entry point, __wine_kernel_init is called which handles the rest of the logic and never returns from the call. Following are the tasks accomplished by the __wine_kernel_init:

- Initialization of program arguments from Unix program arguments
- Lookup of the executable from the file system
- If the file is not found, then an error is printed and the Wine loader stops
- The PE module is loaded in memory using the Windows shared library mechanism
- A new stack is created, whose size is given in the PE header, and this stack is made one of the running thread
- With this new stack, ntdll.LdrInitializeThunk is called which performs the remaining initialization parts, including resolving all the DLL imports on the PE module, and doing the init of the TLS (thread local storage) slots
- Control is then passed to the EntryPoint of the PE module, which will let the executable run

For non-PE executables, Wine hands over the execution to a different executable, `winevdm` (VDM stands for Virtual DOS Machine). This sets up the 16 bit environment to run the executable. Any new 16 bit executables created will eventually be executed by the same winevdm. Thus sharing of address space, cooperative multitasking, managing the selectors for the 16 bit segments are all handled by winevdm.

6.1.9 Miscellaneous

Apart from the above-mentioned main components, there are several others that play important roles in the Wine implementation but are not covered in this report. Among them are: Structured Exception Handling (signals are translated to exceptions), special console handle, customized graphics drivers (implemented on top of X11 driver), synchronization, and messaging subsystem.

6.2 WABI

WABI [24], that stands for Windows Application Binary Interface, is the Sun's implementation with an aim similar to Wine. It lets users run Microsoft Windows applications on several UNIX operating environments that use X Window System. It also has a variation, called WabiServer, that lets multiple users connect remotely and use Wabi on a client computer. It works by translating Windows calls to X Window and Unix calls, and on RISC platforms, translating x86 instructions to RISC instructions. However, the implementations are not as extensive as Wine's and hence fails to correctly execute certain programs.

6.3 LxRun

LxRun [25] is a Linux x86 binary emulator for other x86 *IX machines, such as SCO OpenServer, SCO UnixWare, and Sun Solaris. It works be remapping system calls on the fly. As it turns out, there's not much difference between Linux and iBCS2 binaries. The main difference is the way in which system calls are handled. In Linux, an ”int 0x80” instruction is used, which jumps to the system-call-handling portion of the Linux kernel. On other systems, ”int 0x80” usually causes a SIGSEGV signal. Lxrun intercepts these signals and calls the native equivalent of the system call that the Linux program attempted. The result is that the Linux binary can be run (with the help of lxrun) with a small (usually negligible) performance penalty. All this is accomplished without modifying the kernel or the Linux binary.

6.4 Visual MainWin

Visual MainWin [26] for the UNIX and Linux platforms is an enterprise-class application-porting platform that enables software developers to develop C++ applications on Windows using Visual Studio and deploy them on UNIX and Linux operating systems. The product actually recompiles Windows source code with UNIX compilers to create native UNIX applications. The Visual MainWin Runtime consists of the Windows Runtime on UNIX and Core Services. Together, they enable Windows applications to execute natively on UNIX. Visual MainWin Core Services for UNIX and Linux provide functionality such as synchronization objects, threads and low-level graphic functionality utilized by the Windows Runtime on UNIX. Visual MainWin creates native UNIX binaries requiring no virtual machine with no emulation or mapping performed at runtime to improve performance.

7 Featherweight Virtual Machine

So far, there have been discussions on most of the techniques currently used in virtualization, and the taxonomy gives a reasonable exposure to various categories of them. Although, the taxonomy, by no means, is complete but definitely gives the direction at which in can further be extended to accommodate future innovations. After all the background survey and analysis, we propose an idea that fits in the category described in section 3. In this section, we talk about the issues that motivated us to come up with such a proposal. In addition, we discuss some of its features and applications that make it unique.

7.1 Motivation

Conceptually a virtual machine represents an operating environment for a set of user-level applications, which includes libraries, system call interface/service, system configurations, daemon processes, and file system state. Multiple virtual machines can run on a single physical machine through a virtualization software as discussed in the previous sections. The choice of abstraction-level at which virtualization is implemented, entails a different tradeoff among implementation complexity, run-time performance overhead, flexibility, and degree of isolation.

Many fault-tolerant and intrusion-tolerant systems require the ability to ”try out” untrusted programs in a realistic environment without leaving any permanent damage. Virtual machine technology is a perfect fit to meet this need. Because such trial operations need to be invoked frequently on a routine basis, creating, committing and terminating a virtual machine must be fully automatic and extremely low-overhead. However, none of the current virtual machine implementation address this issue. Although Denali (see section 3.3), and a few others are considered as *”lightweight”*, they still are not usable to solve this issue either because they do not support commodity software, or because the latency involved are higher than the tolerance limit. Thus, to satisfy this new requirement, we develop a new virtual machine architecture on the Windows environment called *feather-weight virtual machine* (FVM). As the name suggests, we aim at making the virtual machine as light as feather so that it takes time of the order of milliseconds to create, manipulate, and destroy such a machine without compromizing any functionality, correctness, or security aspects. In addition, we try to exhibit the beauty and usefulness of such a technology by coming up with two application areas that can benefit from this: secure mobile code execution, safe vulnerability testing.

The key idea of FVM is *name space virtualization*, which virtualizes machine resources at the system call interface using resource renaming technique. A typical operating system supports multiple types of name spaces for various system resources, for example, virtual/physical address space, file system hierarchy, process ID, network address, network protocol number, port number, etc. Through resource renaming, the name spaces visible to processes in one virtual machine are guaranteed to be disjoint from those visible to processes in another virtual machine. As a result, two FVM virtual machines never share any resources and therefore cannot interact with each other directly. FVM is more flexible and scalable, incurs less start-up overhead, and thus makes a more effective building block for cybersecurity applications that require constant spawning of new virtual machines on an interactive basis.

7.2 Related Work

The fundamental idea behind *virtualization* is to introduce an additional layer of indirection so that the a lower-level resource can be transparently mapped to multiple higher-level resources simultaneously. Virtual machine implementations like VMware [7], Virtual PC [10] (see section 3) are very effective in this regard, but incur significant initialization and management overhead. In contrast, FVM uses a much simpler approach for partitioning that has almost negligible management overhead. Light-weight virtual machines [9, 11, 27, 15] (discussed earlier) virtualizes only a subset of the computer hardware. Compared with light-weight virtual machines, FVM is even lighter because it virtualizes the system resource at the system call interface through renaming. Emulators like Bochs [2] are even worse due to their unusually high latency. All of the OS level virtualization utilities discussed in section 4 are very similar to FVM in the way they operate but are not supported on Windows.

The Alcatraz project [28] provides a virtual execution environment for Linux applications through system call interposition technology. It is designed specifically to sandbox untrusted applications that may need to access privileged files. However, it does not support general virtualization thus would not allow multiple virtual machines to co-exist on a single physical machine without interfering one another. Nor does it support virtualization of other types of system resources than files.

Softricity's SystemGuard [29] virtualizes the operating environment of a user-level application so that the application can be bundled together with its operating environment into a self-contained package for distribution. It uses a name space overlay technique to carefully group resources before presenting them to the application. Another system with similar goals and techniques is Jitit's Thinstall [30]. Windows terminal server [31] also solves the resource conflict problem among multiple terminal server sessions using the name space virtualization approach. The *object manager* in the Windows executive uses a hierarchical naming convention for all the system resources and every terminal server session gets a separate sub-tree in this hierarchy.

In some sense, FVM is equal to versioning of system resources plus visibility control. Various versioning file system projects [32, 33] attempt to efficiently maintain multiple versions of the same file. Most if not all of them use block-based versioning rather than file-based versioning to avoid unnecessary data copying. For simplicity, FVM uses a file-based approach, but it also provides more comprehensive versioning support.

7.3 FVM Architecture

7.3.1 Overview

When applying virtual machine technology to fault-tolerant and intrusion-tolerant systems, a common requirement is to start a new virtual machine to run a potentially damaging transaction in the original operating environment while hiding its side effects from the host machine. In other words, the new virtual machine should be able to access as many resources of the host machine as possible, but should never interfere with it. One can satisfy this requirement by setting up a new virtual machine, and copying the hosting machine's environment to the new virtual machine. However, this approach is impractical for existing heavy-weight virtual machine technology because its start-up overhead is too high. In contrast, the proposed feather-weight virtual machine (FVM) architecture is specifically designed to reduce the invocation latency of a new virtual machine and to scale to a large number of virtual machines by minimizing per-virtual-machine resource requirement.

By default, a new FVM virtual machine starts with the same operating environment as the host machine. Therefore, both the invocation overhead and the initial resource requirement are minimal, and applications in the new virtual machine can immediately run against the host machine's operating environment. However, to protect the host machine from a virtual machine, all the updates made by the virtual machine are only visible to its processes but not to those running in the host machine or other virtual machines. To effectively control visibility of updates, FVM performs resource renaming at the system call interface to ensure each virtual machine sees exactly what it is supposed to see. The insight behind this technique is that by controlling the name spaces visible to each virtual machine, it is possible that virtual machines can share as many common resources as possible without interfering with each other. This machine virtualization approach is called *name space virtualization*. Compared with name space virtualization, existing virtual machine technologies incur higher start-up overhead and resource requirement, because they operate at the hardware abstraction layer and thus require unnecessary duplication of common system resources, such as operating system and shared libraries.

On modern Windows operating systems (Windows NT and subsequent versions), the operating environment of a virtual machine is uniquely defined by the following system components: file system image, registry entries (for example, `HKEY_CURRENT_USER`), kernel objects (for example, spin locks), system services[3] (for example, `svchost.exe`), and kernel state. The first four components of a virtual machine each have a separate name space, and therefore can be easily separated through resource renaming. Accordingly, the state of an FVM virtual machine is defined by the following:

- A virtual machine ID,
- An set of environment variables,
- An IP address,
- A list of IDs of the processes that are currently running in it,
- A log of deleted files and registry entries, and optionally
- An initial image of files, registry entries, kernel objects, and system services.

Typically the initial image of a new virtual machine is the same as that of the host machine at the time of its creation. However, a user can choose to specify a special subset of the host machine's image as a new virtual machine's initial image. The virtual machine ID is used for resource renaming. Currently, the FVM architecture simply uses a virtual machine's ID as the prefix when converting a resource name in the virtual machine into a resource name in the host machine. For example, a resource in a virtual machine whose original name is `/a/b` is renamed to `/XYZ/a/b`, where `XYZ` is the ID of the virtual machine. The same renaming mechanism is used in the name space for files, registry entries, kernel objects, and system services. An FVM virtual machine's state also includes an IP address, because it allows the same port such as Port 80 to be used simultaneously in multiple virtual machines that co-exist in the same physical machine. When a socket is created in a virtual machine, it is automatically bound to the virtual machine's IP address. An

[3]On Windows, the term "system service" actually refers to what is commonly known as system calls. We use the term "system services" to mean "daemons."

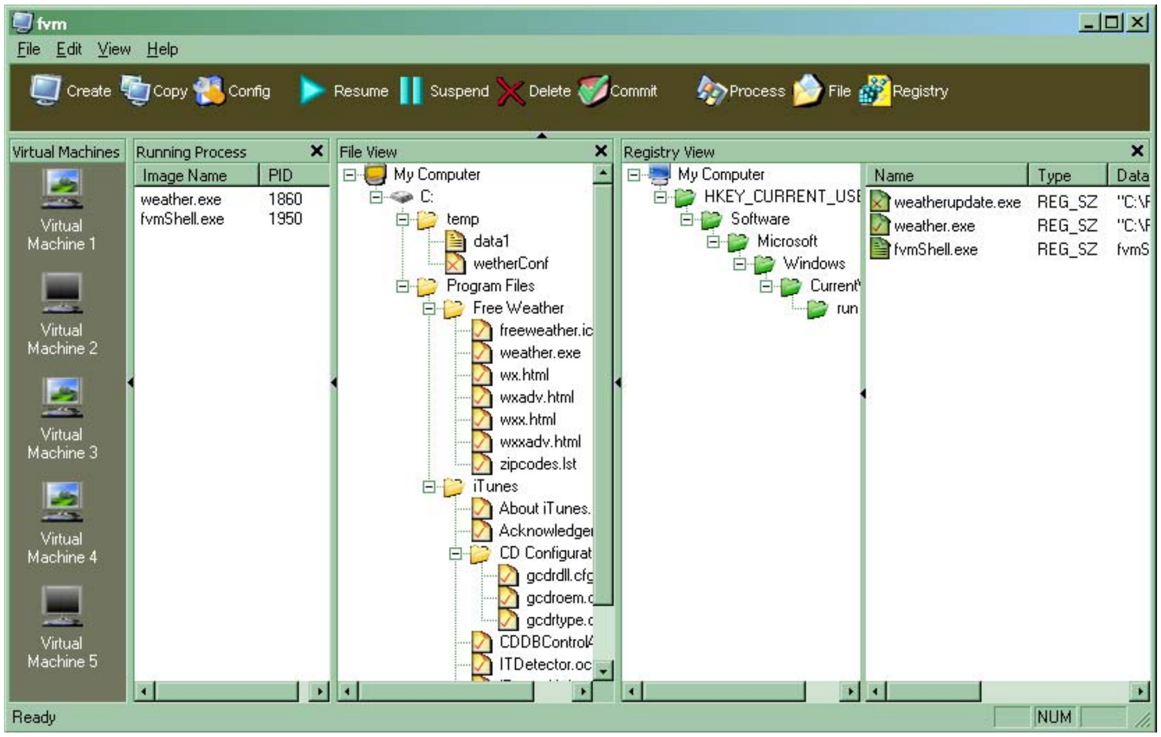


Image Name	PID
weather.exe	1860
fvmShell.exe	1950

Name	Type	Data
weatherupdate.exe	REG_SZ	"C:\F
weather.exe	REG_SZ	"C:\F
fvmShell.exe	REG_SZ	fvmS

Figure 12: Graphical user interface of FVM showing permitted operations and properties.

FVM virtual machine also keeps track of the set of active processes running in it in order to ensure that these processes see the image of the virtual machine and nothing else. Finally, to compute the ”delta” between an FVM virtual machine and the host machine, each FVM virtual machine also maintains a log of deleted files and registry entries. Files and registry entries that are added or modified by an FVM virtual machine are guaranteed to be in its private name spaces through a copy-on-write mechanism, and therefore do not need to be separately recorded.

FVM provides a suite of operations for users to manipulate virtual machines. *CreateVM* creates a new virtual machine whose initial image is a replica of the host machine at the time of creation. *CopyVM* creates a new virtual machine whose initial image is copied from another virtual machine. *ConfigureVM* creates a new virtual machine with an initial image that the user can configure explicitly. *DeleteVM* deletes a virtual machine completely. *SuspendVM* saves the virtual machine's state to a persistent file and renders it inactive. *ResumeVM* restarts a virtual machine from a file that stores the virtual machine's state. *CommitVM* merges the side effect of a virtual machine to the host machine and deletes the virtual machine. To help users decide whether to commit a virtual machine, FVM provides a GUI to display the side effects of a virtual machine, and to possibly commit a selective subset of them. A sample interface with all the operations and properties of a FVM is shown in Figure 12.

7.3.2 Registry Entry Virtualization

When an FVM virtual machine is created, by default it sees all the registry entries in the host machine. Whenever an application in a virtual machine opens a registry entry, FVM checks whether and where the registry entry exists. If a copy of the registry entry exists in the virtual machine itself, FVM uses this copy. If the registry entry exists only in the host machine, FVM duplicates the host machine's copy in the virtual machine if the registry entry is opened for "write," and uses the host machine's copy directly if it is opened for "read only." If an application in a virtual machine creates a new registry entry, it is always created in the virtual machine.

A registry entry in the Windows environment consists of a *key* and a *value*. Registry entries are organized in a hierarchical name space similar to standard UNIX file system. A registry entry can have both subkeys and values. Subkeys are similar to subdirectories while values are similar to files. To reduce implementation complexity, FVM embeds a virtual machine's registry entries in the host machine's registries using Windows' own registry subsystem. More concretely, FVM intercepts all registry-related system calls that use registry keys as arguments, and renames these registry key arguments by prepending a prefix to the path name of these keys. For each virtual machine, FVM creates a registry entry with the key `/HKEY_CURRENT_USER`. For example, for the virtual machine VM1, FVM creates the entry `/HKEY_CURRENT_USER/VM1` as its root, and all the registry entries in VM1 will be stored under this entry . Whenever any application from VM1 accesses a registry entry, FVM adds the prefix `/HKEY_CURRENT_USER/VM1` to the path name of the requested registry entry.

7.3.3 File System Virtualization

File system virtualization is based on exactly the same renaming technique as in registry entry renaming. However, in this case, the "copy on write" (COW) mechanism can be applied at the whole file or individual file block level. Although block-based COW is more efficient than file-based COW, it is also more complicated as it needs to duplicate some of the file system metadata. We plan to start with file-based COW in the first FVM prototype, measure its performance cost under standard workloads, and then decide whether it is worthwhile to pursue the more complex block-based alternative.

To implement file system virtualization, FVM intercepts system calls that use a file name as an input argument, such as create/open files, delete/rename files, query file attributes and traverse directory content. For such system calls, FVM transparently renames the input file name by prepending it with a virtual machine identifier of the issuing process. As a result, these system calls really operate on the renamed file name, as shown in Figure 13. Again, if an application in a virtual machine opens an existing file for write, the file is first copied to the virtual machine and accessed using the its renamed path name. FVM ensures that a copied file has the same attributes and directory structure as the file being copied.

Windows object virtualization is based on the same techniques used in registry entry and file system virtualization. However, the current prototype does not support it.

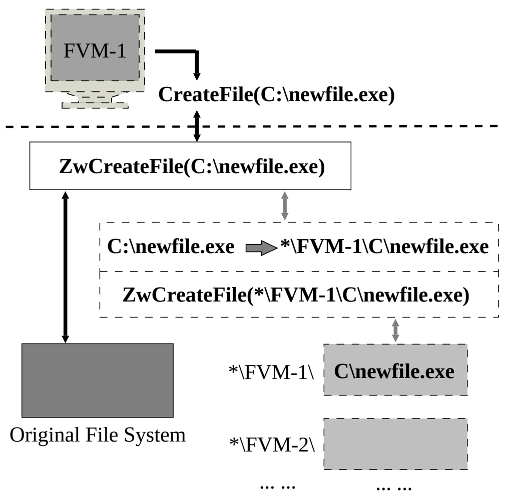


Figure 13: File system virtualization works by renaming file path name arguments at file open time in a way similar to registry entry renaming.

7.3.4 Service Virtualization

Services are to Windows what daemons are to UNIX. They run around the clock and are an integral part of an operating environment. To fully isolate one virtual machine from others, FVM needs to ensure each virtual machine has its own set of services so that sharing among virtual machines through common services is not possible. The Windows operating system provides a generic service called the service control manager (SCM), which is integrated within the kernel and is responsible for starting up and managing all the services. SCM manages the kernel's active services database, interacts with system services to exchange control or status information, and can start or stop a service, save service-related information in registry entries for persistence, etc. Because SCM itself cannot be duplicated in each virtual machine and it performs system state modification on behalf of other services, it is essential that FVM can accurately identify the service behind each of SCM's update operation so that it can attribute each update to the associated virtual machine.

Conceptually, the technique of resource renaming at the system call interface still applies here. However, there are complications. The `CreateService` call in the Win32 DLL, which is typically used to create a Win32 service, is not just a simple wrapper to a system call; rather it makes a series of system calls that cannot be easily recognized by the generic system call interception layer. To solve this problem, FVM employs a DLL hooking mechanism [34] to intercept the `CreateService` call and issues an artificial system call that just passes all the necessary information to the system call interception layer, including a service name and the name of the program image used to start the service. As usual, the DLL hook modifies the original service name by prepending the virtual machine ID to it before calling `CreateService`. The system call interceptor extracts the program image name and the renamed service name for future use. Later on, when SCM forks a service process using a program image name that matches one of the stored image name, FVM binds this service process to the virtual machine whose ID appears in the associated renamed service name, and all subsequent modifications made by this service process are automatically ascribed to the matching virtual machine. When SCM makes a system state modification, it does so by making a system call with the name of the associated service as an argument. Because service names are already renamed when `CreateService` is called, FVM can easily identify from the service name argument the virtual machine on behalf of which SCM performs this modification.

7.3.5 Mapping Processes to VMs

The central design issue in FVM is visibility control, which is essentially reduced to this: When a process issues a system call to read or modify a system resource, which virtual machine should serve as the context within which to interpret this system call? When a virtual machine is first created, FVM automatically starts its first process, which is either a command shell or file browser, and tags it with the virtual machine's ID. All the descendant processes of this initial process belong to the same virtual machine and thus are tagged with its virtual machine ID.

Because Windows does not maintain any process hierarchy, FVM needs to maintain a map between process ID and virtual machine ID. More specifically, FVM propagates virtual machine ID from a parent process to its child process by intercepting `NtCreateProcess` (or `NtCreateProcessEx` for Windows XP), which are the Windows equivalent of `fork`, so as to assign to the newly forked process the same virtual machine ID as its parent. There is one exception to this mechanism: When SCM forks a new process, the new process does not necessarily belong to the physical machine, SCM's operating environment. Instead, the virtual machine ID for the new process is determined by the program image argument of the `NtCreateProcess` call.

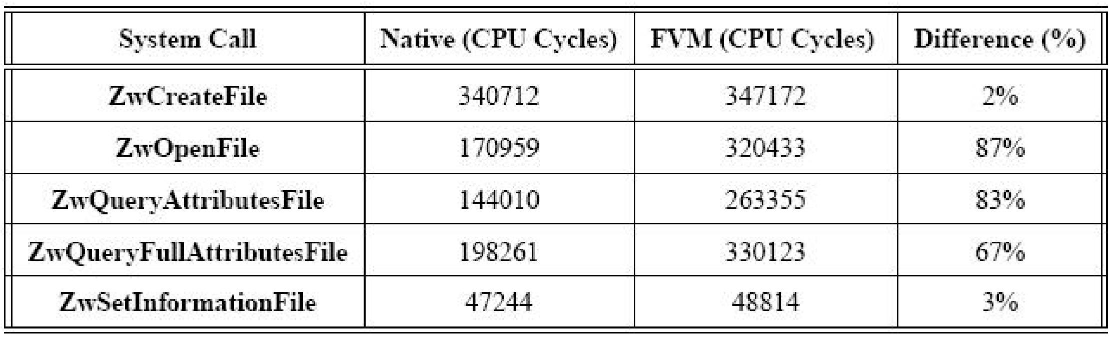
7.3.6 Implementation Considerations

The key building block of FVM implementation is system call interception. For each system call that needs to be instrumented, FVM modifies the corresponding entry in the System Services Dispatch Table (SSDT) to redirect the control through FVM's virtualization logic. FVM's system call interception module is loaded into the kernel as a kernel driver, and is designed to be extensible so that it can serve as a reusable framework for other projects that require similar system call interception functionality.

When a virtual machine is created, the system call interception driver allocates a data structure to maintain the virtual machine's states as described in Section 2.1. When a virtual machine is terminated, FVM discards and de-allocates its in-memory state. When a virtual machine is stopped, its state is saved to a file, from which the virtual machine can be restarted later on.

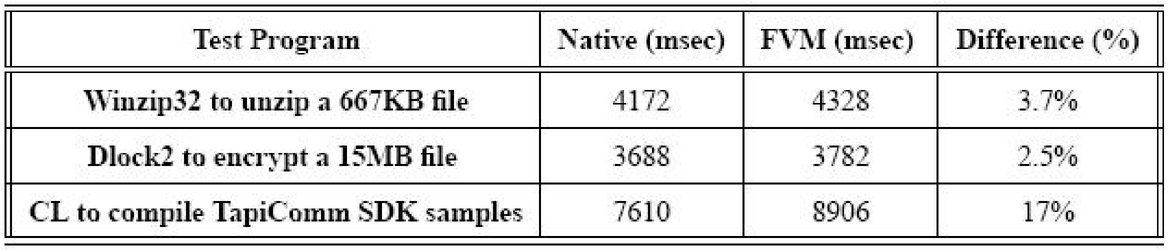
7.4 Performance Evaluation

The performance overhead of FVM comes from the additional interception overhead associated with every system call and the file/object/registry entry copying overhead for an ”open for write” system call. In some sense, the latter should be considered as a component of the cost of starting up a new virtual machine, only distributed over time. Obviously, if a large file needs to be copied to a virtual machine's private directory, the performance penalty is significant. For more heavy-weight virtual machine technologies such as VMware,



System Call	Native (CPU Cycles)	FVM (CPU Cycles)	Difference (%)
ZwCreateFile	340712	347172	2%
ZwOpenFile	170959	320433	87%
ZwQueryAttributesFile	144010	263355	83%
ZwQueryFullAttributesFile	198261	330123	67%
ZwSetInformationFile	47244	48814	3%

Figure 14: Interception overhead for file system related system calls.



Test Program	Native (msec)	FVM (msec)	Difference (%)
Winzip32 to unzip a 667KB file	4172	4328	3.7%
Dlock2 to encrypt a 15MB file	3688	3782	2.5%
CL to compile TapiComm SDK samples	7610	8906	17%

Figure 15: Batch program latencies experienced in FVM

this copying cost also exists and needs to be paid out front; moreover, the sharing of files among virtual machines is not possible.

We measured the system call interception overhead in terms of number of processor cycles spent in a system call for some windows-based applications executing in an FVM virtual machine and compared it with that of the natively executing system call. We ran the applications and the FVM prototype over Windows 2000 server on a Pentium-4 2.8GHz Dell Dimension 2400 desktop machine. For Microsoft Word that opens a 2.8MB .doc file, the average system call overheads for five frequently used file-related system calls are shown in Figure 14. Opening files and querying file attributes are seen to suffer more as these invoke the same system call on both the FVM private directory and the original directory, if the file only reside in the original directory. Fortunately the impact on the overall application execution is negligible, because most system calls do not require any interception.

We also tested three batch programs' runtime overhead due to system call interception, as shown in Figure 15. The runtime overhead for unzipping a 667KB file using WinZip32 under an FVM is about 3.7% of that of the native execution. This result also indicates that although some system call interception overhead is nontrivial, the proportion of these system calls out of all the system calls made by some applications may be small so the total runtime overhead is not significant.

Finally we tested the start-up latency for some Windows-based interactive applications when they run in the native mode and under an FVM, and measured the elapsed time from when a user clicks on the application to when it has finished its initialization and is waiting for user input. This measurement includes the

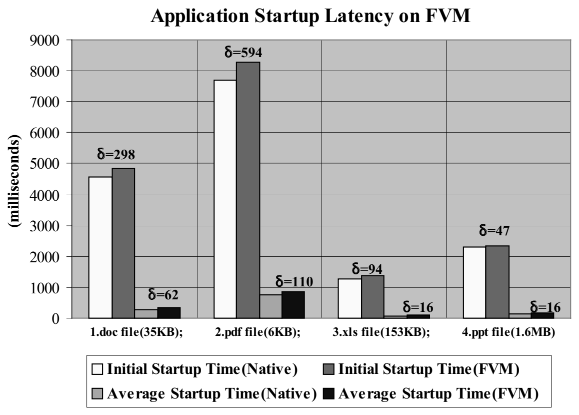


Figure 16: *The startup latency for invoking four Windows applications in a new FVM virtual machine*

initial startup time, which is the startup latency when the application runs for the first time after the physical machine reboots, and the average startup time, which is the average startup latency when the application runs for the second time onwards. The average startup time is much shorter than the initial startup time due to different levels of caching and memory paging. When running Adobe Acrobat Reader against a 6KB pdf file, the additional virtualization overhead `delta` between initial startup times in the native mode and under an FVM is approximately 594 milliseconds, and decreases to 110 milliseconds for the average startup time. This difference arises because most file copying only takes place in the first time when the application runs under an FVM. Tests using other applications show similar results. Figure 16 shows the startup latency of invoking four common Windows applications in a new FVM virtual machine.

The space overhead associated with an FVM state is small because it includes only the file/registry delete log. After we start some Windows-based interactive applications on an FVM for a while, the delete log is always zero, because usually application processes will not delete any existing files or registry keys.

7.5 Application

7.5.1 Secure Mobile Code Execution

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 a Java script embedded within an HTML page, a Visual-Basic script contained in a WORD document, an HTML Help file, an ActiveX Control, a Java applet, a transparent browser plug-in or DLL, a new document viewer installed on demand, an explicitly downloaded executable binary, etc. Because a mobile code typically runs in the context of the user that downloads it, it can issue any system calls that the user is allowed to make, including deleting files, modifying configurations or registry entries, sending emails, or installing back-door programs in the home directory.

FVM enables an intrusion-tolerant approach to the mobile code problem [35, 36]. Whenever a piece of mobile code, be it an email attachment or a downloaded file through web browser, FTP or P2P , is scheduled to run, it is automatically executed in a new FVM virtual machine without user intervention. This virtual machine provides the same operating environment as the host machine, but can effectively isolate any side effects of the mobile code should it turn out to be malicious. To further protect the host machine, the visibility of the virtual machine may need to be further limited. For example, the downloading user's home directory should be hidden to prevent viral propagation through address book. Identifying all possible mobile code execution mechanisms and confining their execution through FVM in a way transparent to users will be a major research focus of this project.

7.5.2 Automatic Clean Uninstall

Although the existing Windows Installer [37] infrastructure eases and automates many installation related tasks, it can hardly do anything when a spyware gets installed through another software that the user installs. Spyware could also be installed through security vulnerabilities in applications like web browser or by another spyware that is already in the system. Examples of common spyware include Bonzi Buddy, SaveNow, Search Assistant, Enhanced Mediaoads, and so on. Moreover, there are handwritten installation scripts that are impossible to automatically uninstall and are usually left to the vendor to provide the uninstallation scripts.

FVM guarantees a clean automatic uninstall of any application by confining all of the installation side-effects inside a virtual machine. When the user tries to install an application, it is automatically started within a separate FVM virtual machine that contains all the side-effects including any spyware installations. Since the side-effects are isolated irrespective of whether they are caused by the installer service or the handwritten installer scripts, FVM is able to uninstall absolutely any kind of installation programs. The modifications through the Windows Installer Service are caught similar to how it figures out the modifications through other services (discussed in Service virtualization).

8 Conclusion

Virtualization technologies provide several important features that make it a very powerful tool to be used across a wide range of applications. These include but are not limited to server consolidation, application sandboxing, access to varieties of hardware and OSes, debugging, mobile computing, packaging (for appliances), testing, easy system administration, and quality of service. All these features have made these technologies immensely popular in academia as well as industry. A closer look reveals that although most of them present a similar operating environment to the end-user, they greatly vary in their architecture design, implementation, and the level of abstraction at which they operate at. In this paper, we study a range of virtual machine technologies and propose a taxonomy for the levels of abstraction at which they function. We identify five different categories that account for most of the the implementation variations: instruction set architecture level, hardware abstraction layer (HAL) level, operating system level, library level, and application level virtual machines. We discuss the basic insight as well as the general approach behind this taxonomy so that it would not be difficult to extend it further in the future if such a need arises. We survey a few examples from each of the categories of virtual machines and study the general design, implementation details, challeging issues involved, and do a comparative study of these. We then propose our own idea of a virtual machine implementation, which we call as *Featherweight Virtual Machine*, that aims at providing an extremely lightweight virtual isolated infrastructure which can act as a playground machine for the purpose of test-driving potentially dangerous programs. We also demonstrate the effectiveness of such a technology by applying it to applications: secure mobile code execution and automatic clean uninstall.

Virtualization is an area that has been in existence since 1960s and attracts a heavy attention from the research community even today. So, we believe, such a survey could actually help people analyze new developments in the area and help put them in perspective. As a future work, we are trying to study a variety of new taxonomies like cross-platform and multi-layer technologies and fit them in the current taxonomy. We also are working on improving the degree of isolation among FVM machines by analyzing the protocols of the common services they share. We are trying to apply FVM to another important application: memory stick based mobile computing.

References

[1] Wine Project, "Wine user guide," http://www.winehq.com/site/docs/wine-user/index.

[2] Kevin Lawton, Bryce Denney, N. David Guarneri, Volker Ruppert, Christophe Bothamy, and Michael Calabrese, "Bochs x86 pc emulator users manual," http://bochs.sourceforge.net/, 2003.

[3] "Transmeta corp crusoe processor," http://www.erc.msstate.edu/ reese/EE8063/html/transmeta/transmeta.pdf.

[4] "Qemu cpu emulator," http://fabrice.bellard.free.fr/qemu/qemu-tech.html, 2004.

[5] Susanta Nanda, Wei Li, Lap chung Lam, and Tzi cker Chiueh, "Binary interpretation using runtime disassembly," http://www.ecsl.cs.sunysb.edu/bird/index.html, 2005.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, "Dynamo: a transparent dynamic optimization system," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 1–12, 2000.

[7] VMware, "Vmware workstation," http://www.vmware.com/products/desktop/ws_features.html.

[8] Jay Munro, "Virtual machines and vmware," *PC Magazine*, December 2001.

[9] 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 nineteenth ACM symposium on Operating systems principles*. 2003, pp. 164–177, ACM Press.

[10] Microsoft, "Microsoft virtual pc 2004," http://www.microsoft.com/windows/virtualpc/default.mspx.

[11] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," Technical Report 02-02-01, 2002.

[12] J. Navaro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent support for superpages," in *ACM Operating Systems Review, Winter 2002 Special Issue*, December 2002, pp. 89–104.

[13] R. Kessler and M. Hill, "Page placement algorithms for large real-indexed caches," in *ACM Transaction on Computer Systems*, November 1992, pp. 338–359.

[14] Plex86, "Plex86 x86 virtual machine," http://savannah.nongnu.org/projects/plex86.

[15] Jeff Dike, "A user-mode port of the linux kernel," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[16] Dan Aloni, “Cooperative linux,” in *Proceedings of the Linux Symposium*, July 2004, pp. 23–31.

[17] P. Kamp and R. Watson, “Jails: Confining the omnipotent root,” in *Proceedings of the 2nd International SANE Conference*, 2000.

[18] AspLinux, “Linux kernel virtualization project,” http://www.asplinux.ru.

[19] Ensim, “Virtual private servers,” http://www.ensim.com/products/privateservers/index.html.

[20] Amit Singh, ,” http://www.kernelthread.com.

[21] Bill Venners, “The lean, mean, virtual machine,” *Java World*, June 1996.

[22] Microsoft, “Common language infrastructure (cli) concepts and architecture,” Draft, October 2002.

[23] “Parrotcode: Parrot documentation,” http://www.parrotcode.org/docs/.

[24] Sun Microsystems, “Wabi 2.1 documentation,” http://docs.sun.com/app/docs/doc/802-3253.

[25] Steven Ginzburg, “Lxrun: linux emulator,” http://www.ugcs.caltech.edu/ steven/lxrun/, January 2001.

[26] Mainsoft, “Port business-critical applications from windows to unix and linux,” White-papaer.

[27] Hideki Eiraku and Yasushi Shinjo, “Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions,” in *In Proceeding of USENIX BSDCon*, September 2003.

[28] Z. Liang, V. Venkatakrishnan, and R. Sekar, “Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs,” in *Proceedings of Annual Computer Security Applications Conference*, December 2003.

[29] Softricity, “Application virtualization,” http://www.softricity.com/products/systemguard.asp.

[30] Jitit Corporation, “Thinstall overview,” http://thinstall.com/help/?introduction.htm.

[31] Microsoft Corporation, “Technical overview of terminal services,” http://download.microsoft.com/download/2/8/1/281f4d94-ee89-4b21-9f9e-9accef44a743/TerminalServerOverview.doc.

[32] C. Soules, G. Goodson, J. Strunk, and G. Ganger, “Metadata efficiency in a comprehensive versioning file system,” in *Proceedings of Conference on File and Storage Technologies*, April 2003.

[33] N. Zhu and T. Chiueh, “Design, implementation, and evaluation of repairable file service,” in *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, June 2003.

[34] I. Ivanov, “Api hooking revealed,” http://www.codeproject.com/system/hooksys.asp, 2002.

[35] T. Chiueh, L. Lam, Y. Yu, P. Cheng, and C. Chang, “Secure mobile code execution service,” in *in Proceedings of Virus Bulletin Conference (VB)*, August 2004.

[36] T. Chiueh, H. Sankaran, and A. Neogi, “Spout: A transparent distributed execution engine for java applets,” *IEEE Journal of Selected Areas in Communications*, vol. 20, no. 7, September 2002.

[37] Microsoft Corporation, “Windows installer service overview,” http://www.microsoft.com/windows2000/techinfo/howitworks/management/installer.asp.