Chapter 8
The role of virtual testing labs

Abstract Security certification involves expensive testing challenges that require innovative solutions. In this chapter we discuss how to address this problem by using a virtual testing environment early on and throughout the testing process. Recent technological advances in open source virtual environments in fact satisfy the demands of test-based software certification, since virtual testing environments can run the actual binary that ships in the final product. Also, a virtual test laboratory can simulate not only the system being tested but also the other systems it interacts with.

8.1 Introduction

The term *virtualization* can be used for any software technology that hides the physical characteristics of computing resources from the software executed on them, be it an application or an operating system. A *virtual execution environment* can run software programs written for different physical environments, giving to each program the illusion of being executed on the platform it was originally written for.

The potentiality of such an environment for carrying out testing and test-based certification come immediately to mind. Thanks to virtualization, it looks possible to set up multiple test environments on the same physical machine, saving both time and money. However, how easy is it to set up a virtual testing environment to support the security certification process? More importantly, can we trust test results obtained on the virtual platform to be equivalent to those obtained on a native one?

To answer these questions, we need to take a look at the basic principles on which the idea of virtualization is based. Let us consider the typical dual-state organization of a *computer processing unit* (CPU). A CPU can operate

either in privileged *kernel* or non-privileged (*user*) mode.[1] In kernel mode, all instructions belonging to the CPU instruction set are available to software, whereas in user mode, I/O and other privileged instructions are not available (i.e., they would generate an exception if attempted). User programs can execute the user mode hardware instructions or make *system calls* to the OS kernel in order to request privileged functions (e.g., I/O) performed on their behalf by kernel code.

It is clear that in this dual-state any software that requires direct access to I/O instructions cannot be run alongside the kernel. So how can we execute one operating system kernel on top of another? The answer is by executing on the first (host) kernel a simulated computer environment, called *virtual machine*, where the second (guest) kernel can be run just as if it was installed on a stand-alone hardware platform. To allow access to specific peripheral devices, the simulation must support the guest's interfaces to those devices (see Figure 8.1).

Although the beginning of virtualization dates back to the 1960s,[2] when it was employed to allow application environments to share the same underlying mainframe hardware, it was only in recent years that it reached out to the public market as a way to decouple physical computing facilities from the execution environment expected by applications [5]. Today, many virtual machines are simulated on a single physical machine and their number is limited only by the hosts hardware resources. Also, there is no requirement for a guest OS to be the same as the host one. Furthermore, since the operating systems are running on top of virtual, rather than physical, hardware, we can easily change the physical hardware without affecting the operating systems' drivers or function [17].

There are several approaches to platform virtualization: Hardware Emulation/Simulation, Native/Full Virtulization, Paravirtualization, and Operating System (OS) Level Virtualization (container/jail system).

- *Hardware Emulation/Simulation.* In this method, one or more VMs are created on a host system. Each virtual machine emulates some real or fictional hardware, which in turn requires real resources from the host machine. In principle, the emulator can run one or more arbitrary *guest operating system* without modifications, since the underlying hardware is completely simulated; however, kernel-mode CPU instructions executed by the guest OS will need to be trapped by a *virtual machine monitor* (VMM) to avoid interference with other guests. Specifically, the virtualization safe instructions are executed directly in the processor, while the unsafe ones (typically privileged instructions) get intercepted and

[1] Actually, some CPUs have as many as four or even six states. Most operating systems, however, would only use two, so we will not deal with multiple levels of privileges here.

[2] In the mid 1960s, the IBM Watson Research Center started the M44/44X Project, whose architecture was based on virtual machines. The main machine was an IBM 7044 (M44) and each virtual machine was an image of the main machine (44X).

trapped by the VMM.[3] The VMM can be run directly on the real hardware, without requiring a host operating system, or it can be hosted, that is, run as an application on top of a host operating system. Full emulation has a substantial computational overhead and can be very slow. Emulation's main advantage is the ability to simulate hardware which is not yet available.

- *Full Virtualization.* This approach creates a virtual execution environment for running unmodified operating system images, fully replicating the original guest operating system behavior and facilities on the host system. The paravirtualization approach is used by the most currently well-established virtualization platforms, such as *VMWare* [16].
- *Paravirtualization* . The full virtualization approach outlined above uses the virtual machine to mediate between the guest operating systems and the native hardware. Since (guest) VMs run in unprivileged mode, mode-sensitive instructions that require a privileged mode do not work properly, while other kernel-mode instructions need to be trapped by the VM, slowing down execution. The Paravirtualization approach tackles this problem using a simplified VMM called *hypervisor.* Paravirtualization relies on dynamic modification of the guest OS code to avoid unnecessary use of kernel-mode instructions. It enables running different OSs in a single host environment, but requires them to be patched to know they are running under the hypervisor rather than on real hardware. In other words, the host environment presents a software interface with dedicated APIs that can be used by a modified OS. As a consequence, the virtualization-unsafe privileged instructions can be identified and trapped by the hypervisor, and translated into virtualization-safe directly from the guest modified OS. Paravirtualization offers performance close to the one of an unvirtualized system, and, like full virtualization, can support multiple different OSs concurrently. The paravirtualization approach is used by some open source virtualization platforms, such as, Xen [18].
- *OS Level Virtualization* The notion of operating system-level virtualization was originally introduced with the Mach operating system [14]. OS Level Virtualization supports a single OS. Different copies of the same operating systems are executed as user-mode servers isolated from one another. Applications running in a given guest environment view it as a stand-alone system. When a guest program is executed on a server tries to make a system call, the guest OS in the server maps it out to the host system. Both the servers and the host must therefore run the same OS kernel, but different Linux distributions on the different servers are allowed.

All the above virtualization approaches have become widespread thanks to the variety of applications areas in which virtual environments can be

[3] In a variation of this approach, unsafe privileged instructions are executed directly via hardware, achieving a better performance level.

deployed. The scenarios in which virtual machines can be used are many, and testing toward multiple platform is clearly an important one [8], since virtualization can be used to combine on the same server different operating systems.

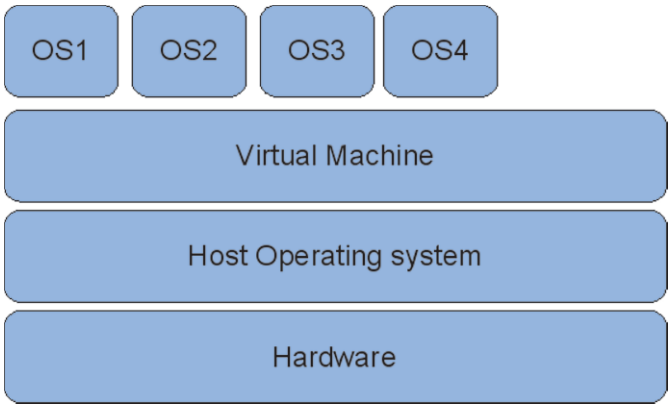

Fig. 8.1: General model of virtualization infrastructure.

In this chapter we provide a brief, informal overview of virtualization internals, aimed at understanding the role of virtual environments in software testing and certification.

8.2 An Overview of Virtualization Internals

To understand how a virtual platform can be used for security testing and certification, let us start by providing a very simplified yet hopefully accurate description of how virtualization actually works. For a more detailed discussion, the interested reader can consult [11].

As mentioned above, most CPU architectures have two levels of privilege: kernel and user-mode instructions. One might envision a VMM as a user mode program which receives in input the binaries of the guest software it is supposed to run. This way, the guest software's user mode instructions can be executed directly on the physical CPU (without involving the VMM), while the kernel mode instructions will cause a trap intercepted and simulated by the VMM in software. In principle, the VMM could completely reproduce the behavior of a real CPU (in this case, the guest software would feature a complete operating system, including a whole set of device drivers). In prac-

tice, the VMM usually maps some of the I/O functions to the host operating system [4].

Problems arise from the fact that many CPU instruction sets include *mode sensitive* instructions that belong to both user and kernel-mode instruction sets. The behavior of mode sensitive instructions depends on the processor current execution mode, and they cannot be trapped like kernel mode ones.[5] For this reason, the original Intel x86 hardware is not straightforwardly virtualizable. Many techniques have been proposed to address the virtualization-unfriendliness due to mode-sensitive instructions. A basic technique to handle double mode instructions is scanning code dynamically and inserting before each mode-sensitive instruction an illegal instruction, which causes a shift to kernel mode and triggers a trap. A related issue is the one of *system calls*. When a process belonging to a guest system running on a VMM enters kernel mode in order to invoke a system call, the shift to kernel mode is trapped by the VMM, which in turn should invoke the guest (and not the host) operating system. A solution is the VMM to use a host system call like `ptrace`() to identify system call invocation on the part of the guest software. When trapping the corresponding shift to kernel mode, the VMM will not execute the call on the host system; rathe,r it will notify the guest system kernel (e.g., by sending a signal), in order to trigger the appropriate action.

8.2.1 Virtualization Environments

Recent CPUs are capable of running all instructions in an unprivileged hardware subsystem, and virtualization software can take advantage of this feature (often called *hardware assisted* virtualization) to eliminate the need for execution-time code scanning. Figuring out efficient virtualization mechanisms, particularly when the underlying hardware is not virtualization-friendly, is still an active area of research; however, the above description should have clarified the basic trapping mechanism through which a virtual machine can operate as a user program under a host kernel, and still look like a "real" machine to its guest software. Software virtualization platforms can set up multiple virtual machines, each of which can be identical to the underlying computer. This section provides a list of some existing virtualization platforms relevant to our purposes.

- *User-Mode Linux*. User-Mode Linux, or simply UML, is a port of the Linux kernel to become a user mode program. In other words, UML is the Linux kernel ported to run on itself. UML runs as a set of Linux

[4] Alternatively, some I/O devices can be stubbed by means of NULL drivers, i.e. device drivers that do nothing.

[5] In the Intel 32 bit architecture, mode sensitive instructions like `STR` can be executed both in user and kernel-mode level, but retrieve different values.

user processes, which run normally until they trap to the kernel. UML originally ran in what is now referred to as the `tt` (trace thread) mode, where a special trace thread uses the `ptrace` call to check when UML threads try and execute a system call. Then the trace thread converts the original call to an effectless one (e.g., `getpid`()), and notifies the UML user-mode kernel to execute the original system call.

- *VMware.* VMware Workstation [16] was introduced in 1999, while the GSX Server and ESX Server products were announced in 2001. VMware Workstation (as well as the GSX Server) runs on top of a host operating system (such as Windows or Linux). It acts as both a VMM (talking directly to the hardware), and as an application that runs on top of the host operating system. VMWare Workstation's architecture includes three main components: a user-level application (VMApp), a device driver (VMDriver) for the host system, and a virtual machine monitor (VMM). As a program runs, its execution context can switch from native (that is, the host's) to virtual (that is, belonging to a virtual machine). The VMDriver is responsible for this switching; for instance, an I/O instruction attempted by a guest system is trapped by the VMDriver and forwarded to the VMApp, which executes in the host's context and performs the I/O using the "regular" system calls of the host operating system [13]. VMware includes numerous optimizations that reduce virtualization overhead. One of the key features for using VMWare for software testing is VMWares *non-persistent mode.* In non-persistent mode, any disk actions are forgotten when the machine is halted and the guest OS image returns to its original state. This is a relevant feature in an environment for software testing, because tests need to start in a known state. VMware ESX Server enables a physical computer to look like a pool of secure virtual servers, each with its own operating systems. Unlike VMware workstation, ESX Server does not need a host operating system, as it runs directly on host hardware. This introduces the problem of mode-sensitive instructions we mentioned earlier. When a guest program tries to execute a mode sensitive instruction it is difficult to call in the VMM, because these instructions have a user mode version and do not cause an exception when run in user mode. VMware ESX Server catches mode-sensitive instructions by rewriting portions of the guest kernel's code to insert traps at appropriate places.
- *z/VM.* z/VM [19], a multiple-access operating system that implements IBM virtualization technology, is the successor to IBM's historical VM/ESA operating system. z/VM can support multiple guest operating systems (there may be version, architecture, or other constraints), such as Linux, OS/390, TPF, VSE/ESA, z/OS, and z/VM itself. z/VM includes comprehensive system management API's for managing virtual images. The real machine's resources are managed by the z/VM Control Program (CP), which also provides the multiple virtual machines. A virtual machine can be defined by its architecture (ESA, XA, and XC, that refer to specific

IBM architectures), and its storage configuration (V=R, V=F, and V=V, refers to how the virtual machine's storage is related to the real storage on the host).

- *Xen.* Xen is a virtual environment developed by the University of Cambridge [6, 2, 18] and released under the GNU GPL license. Xen's VMM, called *hypervisor*, embraces the paravirtualization approach, in that it supports x86/32 and x86/64 hardware platforms, but requires the guest operating system kernel to be ported to the x86-xenon architecture [6]. However, when hardware support for virtualization is available, Xen can run unmodified guest kernels, coming closer to the full virtualization approach. We shall discuss Xen in more detail in section 8.3.2.

8.2.2 Comparing technologies

Let us briefly discuss how the different approaches to virtualization suit the needs of software testing and certification. Full emulation can provide an exact replica of hardware for testing, development, or running code written for a different CPU. This technique is of paramount importance for running proprietary operating systems which can not be modified. It is however computationally very expensive. OS Level Virtualization is a technique aimed at supporting production sites (e.g., Web server farms), rather than software testing or development, since the individual kernels that run as servers are not completely independent from each other. Paravirtualization addresses the performance problem of full virtualization, while preserving good isolation between virtual machines. However, the mediation by the hypervisor requires a level of patching or dynamic modification of the guest OS which is best suited to open source platforms like Linux. Paravirtualization is most useful for testing and distributing software, and for stress tests (Chapter 3) trying to crash the software under test without affecting the host computer. Individual users can be satisfied with emulation or full virtualization products, such as, VMware Player, and VMWare Server for Linux and Windows, and with free products for Linux, such as, Qemu [10] and Bochs [3].

Of course, emulation (full virtualization) has practically no alternative when deploying, on a hardware platform, software originally written for a different hardware architecture. Paravirtualization is an interesting alternative to full virtualization for automating software testing activities. Using paravirtualization, a single machine can be used to test applications in multiple configurations and on different OSs. Often, development is done on one platform or distribution, but has to be verified in other environments. Also, it is possible to quickly create all combinations, and assign them to testers. Another benefit is performing tests on multiple platforms in parallel: a failure in one VM does not stop testing in others.

8.3 Virtual Testing Labs

Since they were introduced, virtualization platforms have caputred the interest of software suppliers as a way to reduce testing costs. Testing is one of the most expensive activities that software suppliers must incorporate in their development process. Many of the testing methodologies we described in Chapter 3 are the result of years of collaborative work by academics and researchers. In a typical software development project, around 30 % of the project's effort is used for testing. Instead, in a mission critical project, software testing is known to take between 50 to 80 % of projects effort [4]. Being able to provide the right testing infrastructure in a short time and at a reasonable cost is a major issue in reducing the impact of testing on software projects. Historically, however, the only option for comprehensively testing a software system, required replicating the system execution environment in a test lab.

Today, many suppliers have adopted alternative approaches including full-system simulation early on and throughout the software development process. Paravirtualization provides an efficient and flexible environment for software testing. More tests can be executed in parallel without affecting the outcome of each other, since they are running in different environments. For example, if two operating systems are installed in two virtual environments, running a test on the first one would not affect the second.

In terms of test-based certification, virtualization offers a new prospective for security testing. Before releasing any software product, typically it has to pass all the functional and security tests designed for it. However, in certification related security testing (Chapter 3), testers need to run the tests on different configurations with different input parameters to check all possible sources of vulnerabilities. In such cases, virtualization offers an effective environment to run certification-related security tests. Testers can simulate different hacking scenarios on different virtual machines, or create an entire virtual network to simulate networks attacks such as flooding attacks. Let us now describe how a testing environment can be set up based on Xen open source virtualization platform. Then, we will briefly discuss how CC tests can be conducted in a Xen-based virtual execution environment.

8.3.1 The Open Virtual Testing Lab

We now present our Xen-based *open virtual lab* (OVL) and its usage for carrying out the testing required by CC certification. An OVL is composed by different OVL virtual machines, each one consisting of an image of the Linux operating system and application-level software. OVL administrators can interact with the OVL virtual machines via a user-friendly administration interface (OVL-AI) [1] presented in Section 8.3.5.

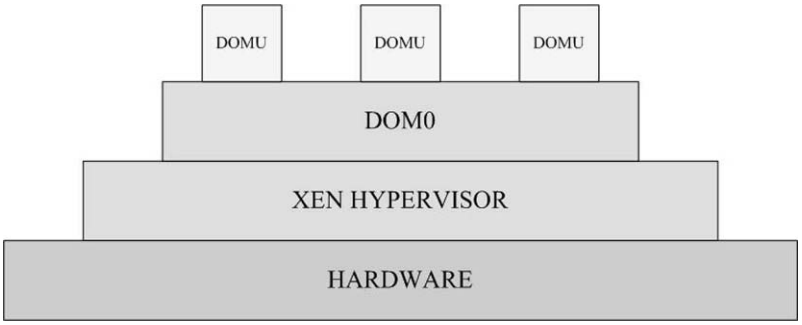

Fig. 8.2: Xen system layers.

8.3.2 Xen Overview

A Xen virtualization system is composed of multiple software layers (see Figure 8.2). Individual virtual execution environments are called *domains*. Xen's hypervisor [6] manages the scheduling operation related to the execution of each domain, while each guest operating system manages the VM application scheduling. During system initialization, a domain with special privileges, called *Domain 0*, is automatically created. Domain 0 can initialize other domains (*DomUs*) and manage their virtual devices. Most management and administration tasks are performed through this special domain.

Xen's current usage scenarios include kernel development, operating system and network configuration testing, server consolidation, and server resources allocation. Several hosting companies have recently adopted Xen to create *public virtual computing facilities*, that is, Web farms capable of flexibly increasing or decreasing their computing capacity. On a public virtual computing facility, customers can commission one, hundreds, or even thousands of server instances simultaneously, enabling Web applications to automatically scale up or down depending on computational needs.

8.3.3 OVL key aspects

The Open Virtual Lab provides each user with a complete Linux-based system image. Also, OVL allows for setting up *virtual internet networks*, by connecting multiple virtual machines, to perform network tests. This feature allows testers to set up their own client-server applications in a virtual network environment. OVL's full support for network programming and middleware is a distinctive feature with respect to commercial virtual laboratories, which focus more on network equipment configuration than on distributed application development.

OVL supports two adoption models: *OVL as a product*, that is, OVL distributed and adopted as a Xen-based open source environment; and *OVL as a service*, showing how OVL can be shared with testers from partner institutions. In both models, costs are mostly related to hosting the environment or purchasing the hardware for running it, since OVL is entirely open source software licensed under the GPL license.

In OVL, each virtual machine is represented by an image of its operating system and application-level software. When configuration changes on a set of virtual machines are needed, OVL administrators can operate via the OVL Administration Interface (OVL-AI). In particular, OVL's design is focused on supporting *scale-up* operations [1]. In a scale-up approach, the system is expanded by adding more devices to an existing node. This action consists in modifying the configuration of every single virtual machine adding, for example, more processors, storage and memory space, or network interfaces, depending on testers needs in a particular situation. Instead, in a scale-out approach, the system is expanded by adding more nodes. In this case, the number of available virtual machines can again be increased (or reduced) easily by OVL-AI. This operation will be beneficial, for example, when new testers join or leave the virtual testing environment.

8.3.4 Hardware and Software Requirements

Intuitively, OVL hardware requirements are essentially two: a storage unit large enough to give a complete software environment to all testers, and enough RAM memory to manage hundreds of virtual machines at the same time. Fortunately, both these requirements can be met remaining within the limits of a tight budget.

The implementation of OVL's virtual machines required some additional considerations.

- *Protection.* Under OVL, each virtual machine has to be an efficient, isolated duplicate of a real machine [9]. In other words, every virtual machine must work in a sealed environment, insulating its disks and memory address space and protecting its system integrity from VM failures.
- *Uniformity.* All virtual machines support a complete and up-to-date operating system to provide the testers with all the instruments needed to carry out administration tasks and test programs. While paravirtualized VMM can, in principle, support a diverse set of guest operating systems, some hardware constraints, in particular the 64-bit server architecture, restrict the range of acceptable guest kernels.

By default, OVLs virtual machines are implemented on the Gentoo *Gentoo* Linux distribution. Gentoo [15] has some distinctive characteristics that fit the requirements of a virtual testing environment. First, a major feature

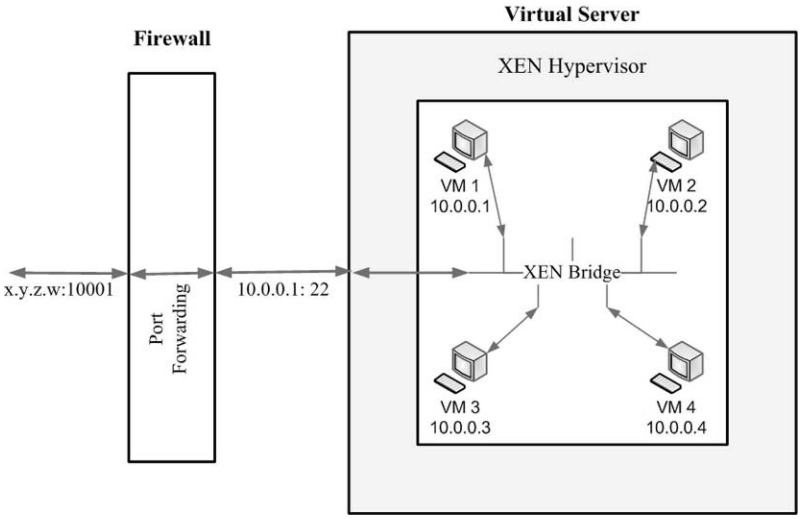

Fig. 8.3: Communications between virtual machines and the external net.

of Gentoo distribution is its high adaptability, because of a technology called *Portage*. Portage performs several key functions: *software distribution*, that permits developers to install and compile only the needed packages that can be added at any time without reinstalling the entire system; *package building and installation*, that allows building a custom version of the package optimized for the underlying hardware; and *automatic updating* of the entire system. Second, Gentoo supports 64 bit hardware architectures and implements the Xen environment in full. Finally, Gentoo is an open source system, distributed under GNU General Public License.

In the current OVL environment, each tester accesses his or her own virtual machine using a secure `ssh` client connected directly to the OVL firewall on a specific port number (computed as `tester_id + 10000`) (see Figure 8.3). Based on the source port, the OVL firewall forwards the connection to the corresponding virtual machine. Figure 8.3 shows how the tester whose `tester_id` is equal to 1 gains access to the firewall. Based on the tester's port number (`10001`), firewall rules forward the incoming connection to the local IP that identifies the tester's own virtual machine. Looking at the example in Figure 8.3, the incoming communication on port `10001` is forwarded to the local IP address `10.0.0.1` on port `22`, and then to the virtual machine `1`.

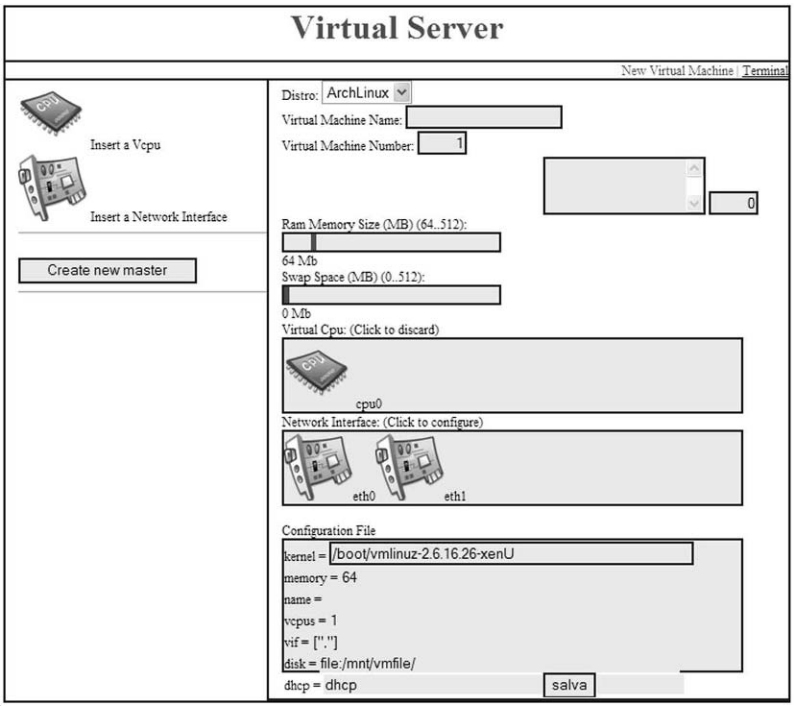

Fig. 8.4: The OVL Administration Interface (OVL-AI).

8.3.5 OVL Administration Interface

The Administration Interface (OVL-AI) module lies at the core of the OVL environment. OVL-AI enables simple management of the entire system via a straightforward Web interface (see Figure 8.4). OVL-AI provides a simplified procedure for the creation, configuration, and disposal of single virtual machines, or pools of virtual machines. Configuration is performed by choosing visually the simulated hardware cards to be inserted in each virtual machine. OVL-AI has been implemented following a multi-tiered approach. Namely, OVL-AI relies on *AJAX* on the client-side, on *PHP* on the server-side, and on *Bash*, for the interaction with the OVL server's operating system.

8.4 Using OVL to perform LTP tests

We now describe how the set of LPT tests used in CC certification process can be executed over a Xen-based virtualization platform like OVL.

Our experiments used a Xen virtual environment based on a Fedora Core 7 distribution, a general purpose Linux distribution developed by Fedora community and sponsored by Red Hat, which supports the Xen hypervisor in a native way. In addition, the Fedora distribution fully embraces the Open Source philosophy and wants "to be on the leading edge of free and open source technology, by adopting and helping develop new features and version upgrades" [7].

The main purposes of our experiments was to investigate and prove the reliability of a virtual environment as a base for the CC certification process. Today, in fact, the CC certification of a system or a product is strictly bounded to the TOE, that is, the precise HW configuration and the OS running over it (see Chapter 3). OS virtualization represents a key factor in limiting this drawbacks, if it provides the same security strength of an OS running on a physical machine. This would also make the testing and certification of an OS less costly, in terms of required hardware.

We then tested the OVL-based environment to prove that the LTP results under a virtualized system are the same of traditional testing, or at least have negligible variations.[6]

Machine Type	**Kernel Type**	**Total failures over 860 tests**
Fedora 7	2.6.20-2925.9.fc7**xen**	14
	2.6.21.7.fc7**xen**	14
	2.6.21-1.3194.fc7	9
	2.6.22.1-41.fc7	9
Fedora 8	2.6.21-2950.fc8**xen**	6
	2.6.21.7-2.fc8**xen**	6
	2.6.23.1-42.fc8	1

Table 8.1: LTP Test Results

Table 8.1 presents our results based on more than 860 tests. The discrepancies between physical and virtual results are 0.6% for Fedora 7, and 0.5% for Fedora 8, and are mostly caused by test growfiles having more than 13 repetitions. It is then probable that such discrepancies are not caused by security issues, but rather by tests generating a lot of I/O processes that cause failure due to the expiration timeout. Also, our experimental results show a minimal discrepancy between the LTP failures in the virtual and in the real machine (less then 1%) due to the kernel version, but no false negatives. In conclusion, our experiments prove that although a perfect matching between real and virtual environments is not possible, the Xen paravirtualization technique provides a reliable environment suitable for CC certification process.

[6] As discussed in Section 8.2.1, it is not possible to have exactly the same kernel version running on virtual and real systems, since the modified kernel version must be prepared to be integrated with Xen.

8.5 Conclusions

The availability of a testing infrastructure is a major factor in keeping software testing costs under control, especially as a part of test-based certification processes [12]. Linux SLES 8 CC certification tests showed practically no discrepancies when re-executed under a virtual Xen-based environment.

References

1. M. Anisetti, V. Bellandi, A. Colombo, M. Cremonini, E. Damiani, F. Frati, J.T. Hounsou, and D. Rebeccani. Learning computer networking on open paravirtual laboratories. *IEEE Transactions on Education*, 50(4):302–311, November 2007.
2. M. Anisetti, V. Bellandi, E. Damiani, F. Frati, U. Raimondi, and D. Rebeccani. The open source virtual lab: a case study. In *Proc. of the Workshop on Free and Open Source Learning Environments and Tools (FOSLET 2006)*, Como, Italy, June 2006.
3. *Bochs*. `http://bochs.sourceforge.net/`.
4. J. Collofello and K. Vehathiri. An environment for training computer science students on software testing. In *Proc. of the 35th Annual Conference Frontiers in Education (FIE 2005)*, Indianapolis, Indiana, USA, October.
5. D. Dobrilovic and Z. Stojanov. Using virtualization software in operating systems course. In *Proc. of the 4th IEEE International Conference on Information Technology: Research and Education (ITRE 2006)*, Tel Aviv, Israel, October 2006.
6. B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, Bolton Landing, NY, USA, October 2003.
7. Red Hat. Inc. Fedora objectives. `http://fedoraproject.org/wiki/Objectives`.
8. P.S. Magnusson. The virtual test lab. *Computer*, 5(95–97):38, May 2005.
9. G.J. Popek and R.P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.
10. *Qemu open source processor emulator*. `http://bellard.org/qemu/`.
11. M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technologies and future trends. *Computer*, 5(39–47):38, May 2005.
12. S. Seetharama and K. Murthy. Test optimization using software virtualization. *IEEE Software*, 5(66–69):23, September-October 2006.
13. J. Sugerman, G. Venkitachalam, and L. Beng-Hong. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proc. of the USENIX Annual Technical Conference 2002*, Monterey, CA, USA, June 2002.
14. The mach project home page. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html/`.
15. G.K. Thiruvathukal. Gentoo linux: The next generation of linux. *Computing in Science & Engineering*, 6(5):66–74, September-October 2004.
16. *Vmware*. `http://www.vmware.com/`.
17. C. Wolf and E.M. Halter. *Virtualization from the Desktop to the Enterprise*. Apress, 2005.
18. *Xen*. `http://www.xen.org/`.
19. *z/VM*. `http://www.vm.ibm.com/`.