A problem in building a new, secure operating system is that existing applications may not run on the new system. Operating systems define an application programmer interface (API) consisting of a set of system calls it supports. New operating systems often define new APIs, resulting in the need to port applications and/or write new applications. This has been a major problem in gaining acceptance for secure operating systems based on emerging kernels, such as Trusted Mach \cite{TrustedMach} and Flask \cite{Flask}, and the reason for so much focus on securing commercial systems.

An alternative that enables execution of multiple operating systems on one computer is called a virtual machine system. A virtual machine system enables multiple operating system instances and their applications to run concurrently on a single physical machine. Each operating system instance runs in a virtualized environment that emulates a physical platform, called a virtual machine (VM). While each operating system still manages the state of its applications and abstractions (e.g., files, sockets, processes, users, etc.), it does not actually manage use of the physical hardware.

In virtual machine systems, a new component is introduced, called a virtual machine monitor (VMM), that multiplexes physical system resources among the operating systems in the VMs. The virtual machine operating systems no longer control the use of system hardware, but instead receive hardware access only via an indirection through the VMM. Only the VMM runs in supervisor mode, so only it has access to system hardware directly.

VMM architectures are classified into two types, as shown in Figure 11.1, distinguished by whether the VMM runs directly on the hardware (Type 1) or on another host operating system (Type 2). Since a Type 2 VMM requires the services of a host operating system in order to run, its trust model must include this operating system (and perhaps it is insecure). As a result, the trusted computing base of a Type 1 VMM can be smaller than that of a Type 2 VMM. Examples of Type 1 VMMs include IBM's VM/370 \cite{VM370}, Xen \cite{Xen}, and VMWare's ESX Server \cite{ESX}. Examples of Type 2 VMMs include VMWare's GSX Server \cite{GSX}, Microsoft's Virtual PC \cite{VirtualPC}, and User-Mode Linux (UML) \cite{UML}. Of course, Type 1 VMMs must also provide system services, such as IP networking, so care must be taken to keep the Type 1 VMM's trusted computing base as small as possible.

There are variants Type 2 VMMs as well where all the VMs use the same kernel interface, so all VMs use the host operating system for handling all system calls. Such systems include Solaris Containers (Zones) \cite{SolarisContainers} (see Chapter 8) and FreeBSD Jails \cite{FreeBSDJails} (see Section 7.5.3). In both of these systems, the host operating system creates isolated computing environments akin to virtual machines, but all the system calls from the virtualized systems are forwarded to the host operating system. There are no guest operating systems in these systems.
Virtual machine systems were originally envisioned as a means to provide better resource utilization, by enabling multiple software systems to use a single system's hardware. Virtual machine system designs sometimes focus on supporting legacy code [39, 69], but other virtual machine designs have focused on security, or at least isolation, as a main goal [161, 259, 332]. In general, virtual machine system designers now expect to: (1) support legacy applications effectively, with little or no modification; (2) provide effective isolation and security enforcement; and (3) with a modest performance overhead that would be undetectable given modern hardware. For security, virtual machine systems have a couple of significant advantages over traditional systems.
First, virtual machine systems offer the potential of reducing the size of the trusted computing base. Since operating systems run in VMs, it is possible to remove the operating system from the trusted computing base. The VMM now becomes the trusted computing base. However, whether this actually reduces the size of the system's trusted computing base depends on how the VMM is designed. While a few have specially-designed, small code bases, many VMMs depend themselves on a complete, ordinary operating system (even Type I VMMs).

Second, virtual machine systems provide an additional, coarser-grained layer of control for securing a system. The hope is that coarser-grained enforcement translates into simpler mediation and simpler policies. With respect to the reference monitor guarantees, virtual machine systems should make it easier to ensure complete mediation and make it easier to verify that security goals are met. First, a VMM reference monitor only needs to mediate the distribution of resources among VMs and inter-VM communication. Since resources are typically partitioned among VMs (i.e., they are not shared), securing resource distribution is often simpler. Once a VM is started, only communication with other VMs needs to be mediated. Typically, only a small number of primitives enable inter-VM communication. Second, the VMM policy will likely be simpler than a secure operating system policy. Because the number of VMs is smaller than the number of processes running on an operating system and the number of possible inter-VM communications is smaller than the number of resources in a traditional system (e.g., files), it should be easier to verify that the VMM policy expresses the intended security goals.

Rushby identified the reasons above as motivation for moving from the security kernel approach (see Chapter 6) to a new approach, called a separation kernel [259, 260]. We first review Rushby's arguments for the separation kernel architecture. We then examine the design and implementation of a secure virtual machine system, the VAX VMM Security Kernel [161]. We then examine issues in building secure operating systems in the context of the current virtual machine systems.

While the additional layer of control offered by virtual machine systems presents these potential benefits, building secure virtual machine systems is not without challenges [107]. First, virtual machine systems may generate problems for administrators, as there will be many more VMs to administer and disinfect than physical machines. Second, the ability to save, restore, and migrate VMs may generate difficulties in ensuring that the VM software base is current and consistent with organizational requirements. Third, the identity of virtual machines will be harder to determine than the identity of physical machines. Virtual machines may even be able to migrate across administrative domains. Fourth, data leaks of VMs into various physical systems and the integrity impact of individual systems into VMs may be difficult to track [51, 52]. Virtual machine security solutions must account for these challenges in order to leverage its benefits.

11.1 SEPARATION KERNELS

In 1981, Rushby examined the difficulty of building security kernel systems, such as Scomp and GEMSOS discussed in Chapter 6. Rushby found that security kernel systems, despite their near-
minimal trusted computing base, had a significant, uncontrolled reliance on trusted services. As a result, he defined an alternative approach that he called a separation kernel [259, 260].

In a multilevel (MLS) system, if any process can write data from a higher sensitivity level to a lower sensitivity level, it violates the MLS policy (see the Bell-LaPadula policy [23] in Chapter 5). However, some services are entrusted with such operations, such as inline encryption systems [259], that encrypt secret data and send it via public networks. Also, other services may be trusted to process data at multiple sensitivity levels with leakage, such as file and print servers. In an MLS system, such processes are simply trusted, and the MLS policy is not enforced on them.

Rushby claimed that ensuring the correct behavior of trusted services of a security kernel system is too complex. In a general purpose system, we have a large number of trusted services, potentially complex interactions among trusted services, and a variety of interfaces accessible to untrusted processes. The SELinux system with its 30+ trusted programs is indicative of the number of trusted programs in a general-purpose system. The interactions among the resulting trusted processes are not clearly identified, but they are likely to be complex. Most dangerous of all is the number of ways that untrusted processes may invoke trusted programs. In minimal security kernel system, such as Scomp and GEMSOS, there were 30–40 gates defined to control such invocations. In a modern operating system, there are hundreds of system calls.

Rushby’s solution is to treat each trusted program as one would a single node in a distributed system. For example, a file server node would be a single-purpose system attached to other systems via a single communication channel. If the “file server adheres to and enforces the multilevel security policy, the security of the rest of the system follows” [259]. That is, enforcement of system security goals can be composed from isolated elements that “adhere to and enforce” [259] those security goals.

Rushby coined the name of such a system as a separation kernel to distinguish it from a security kernel. A separation kernel emphasizes independence and authorized communication. Each trusted service runs in an isolated and independent system, perhaps on the same physical platform or perhaps not, and the services can only be accessed by a small number of mediated communication channels. The separation kernel is capable of complete mediation of such communication channels, such that the services could be isolated completely from the remainder of the system.

Rushby noted the similarity between the separation kernel concept and virtual machine systems, as we described them above. The major distinction is that separation kernels do not require that the separation kernel provide a virtualized hardware API, as a virtual machine monitor does. The trusted services in a separation kernel may be customized to a separation kernel system and minimized (e.g., not run a guest operating system). With increasing popularity of paravirtualized hypervisors, such as Xen [19], which require some awareness of running on a virtual machine monitor, and custom VMs as proposed for Terra [105], the line between a separation kernel and a virtual machine system is becoming blurrier each year.

A particular family of systems that implement the separation kernel approach are called Multiple Independent Levels of Security (MILS) systems [131, 7, 193, 9]. A MILS system architecture
11.2. VAX VMM SECURITY KERNEL

is shown in Figure 11.2. A each service runs in an isolated regime supported by a MILS middleware layer. Coarse-grained communication channels, analogous to network communication are provided, and an MILS separation kernel mediates access. If a trusted service is required (i.e., is trusted to enforce MLS as described above), it placed in its own regime, and the MILS middleware forwards unsafe requests to such services automatically. For example, Figure 11.2 shows a service that encrypts network traffic before being forwarded to the network service. Such trusted services should be small, so they may be verifiable. The MILS separation kernel architecture has been applied to mission-critical deployments for some time, but it is just now starting to garner attention in the mainstream. A proposal for how to construction and evaluate MILS systems has been proposed [233] (i.e., a protection profile, see Chapter 12), as has a critique [347].

**11.2 VAX VMM SECURITY KERNEL**

The VAX VMM Security Kernel is a virtual machine system that aims to achieve the goals of a secure operating system [161]. The VAX VMM runs untrusted VMs in such a manner that it can control all inter-VM communications, even covert channels. The key features of the VAX VMM are its implementation of virtualization, which ensures that the untrusted VMs cannot circumvent the authority of the VMM, and its layered system architecture, which improves security via modularity and minimizing interdependencies.

The VAX VMM design was begun in 1981 from a discussion between Paul Karger and Steve Lipner. The project was motivated by the KVM/370 system, which was a retrofit of the security into the existing IBM VM/370 system [114]. The KVM/370 design was limited by the need to reuse the code from the existing VM/370 system. The KVM/370 design isolates VMs into separate security classes within the architecture of the VM/370 system by adding a layer between the VM/370
VMM and the untrusted VMs, called the Non-Kernel Control Program (NKCP). An NKCP was created for each MLS sensitivity level (e.g., secret and confidential), and they ensured that all VM communication across sensitivity levels was mediated. However, the addition of this additional layer means that access to VMM functions generated two context switches, one to NKCP and one to the VMM. This had a significant effect on system performance, reducing the performance by a factor of 2–10 [114]. In addition, the VAX VMM had more extensive support for security management and covert channel prevention not present in the KVM/370 system.

11.2.1 VAX VMM DESIGN
The architecture of the VAX VMM Security Kernel system is shown in Figure 11.3. The VAX VMM is a Type 1 VMM, in that it runs directly on the hardware. The VMM includes services for storage (on tape and disk) and for printing. In general, the VMM must multiplex all physical devices among VMs, but some key devices, in particular Ethernet networking devices, were not supported, which became problematic for the system’s deployment. Each VM could run at its own clearance, and the VMM includes a reference monitor that ensures that any use of physical resources is mediated according to a mandatory access control policy.

![Figure 11.3: The VAX VMM System Architecture: virtual machines are labeled according to the secrecy of the information that they can obtain through the VMM security kernel to the system’s physical devices.](image)
The VMM architecture results in coarser-grained physical resources than for a traditional operating system. The VAX VMM views access to individual devices and storage (i.e., disk and tape) volumes as objects. Thus, its reference monitor mediates access at this level. The VAX VMM also provides an abstraction called virtual disks, which partitions the physical disk into isolated chunks that may or may not correspond to the physical disk boundaries. The use of virtual disks enables two VMs at different clearances to share a single physical disk securely (e.g., preventing one VM from accessing the other’s data).

The VAX VMM enforces both secrecy and integrity requirements upon its VMs. Versions of the Bell-LaPadula secrecy model [23] and Biba integrity model [27] (see Chapter 5) are supported. In order to express permissions that do not fit into these information flow models, the VAX VMM provides the means for specifying additional privileges. For example, if a user is permitted to see some data at a higher secrecy level than she is allowed by her clearance, a user privilege can be specified to permit such access. The use of such privileges requires a trusted path (see Chapter 7), and such uses are audited by the VMM.

The VAX VMM itself is designed in a layered fashion, motivated by the Multics design work of Janson [151], Reed [254], and the Naval Postgraduate School [67]. The idea is that each layer adds well-defined VMM functionality, and no layer depends on functionality provided by a higher (i.e., less-trusted) layer. For example, the I/O layer provides device I/O (i.e., supplies the VMM system drivers), and the VM physical memory layer uses some of these drivers to manage physical memory. The VM virtual memory layer uses both to implement manage access to a prescribed amount of physical memory per VM and ensure proper mapping of device resources (e.g., storage via the I/O layer) to virtual memory.

In building the VAX VMM security kernel, four major challenges had to be addressed: (1) virtualizing the protection rings of the VAX processor; (2) identifying the sensitive instructions of the VAX processor; (3) emulating I/O operations generated by the untrusted user VMs; and (4) enabling the VAX VMM system to be self-virtualizable. The first two challenges ensure that the VMM’s reference monitor truly provides complete mediation. The second two challenges provide function in a secure manner.

First, the VAX VMM designers had to add a new virtual ring to the VAX processor. The VAX processor supports four rings: kernel, executive, supervisor, and user. A traditional VAX OS (VMS) ran in the kernel ring, but also some specific VMS system software ran in the executive and supervisor rings as well. Thus, all four physical rings were already used. The VAX VMM design runs the VMM in the kernel ring, and compresses the OS kernel and executive into a single ring. While some protections between kernel and executive code are added, in general, the OS kernel may not be protected from bugs in the executive since they run in the same ring.

Second, even though the OS kernel is run a higher ring than the VMM, the processor may still permit the higher ring to run security-sensitive operations, thus circumventing complete mediation. Processor instructions that may be run only in the privileged ring (i.e., the kernel ring in the VAX architecture) are said to be privileged. Processor instructions that reference or modify sensitive data
in the privileged ring are said to be sensitive, and are security-sensitive in building a VMM reference monitor. Popek and Goldberg [247] state that for an architecture to be virtualizable, all sensitive instructions must be privileged. However, for the VAX instruction set, this was not the case, so the designers had to extend the VAX architecture (i.e., change the VAX microcode actually) to indicate whether code was running in a VM, so that the VMM could emulate its execution securely. For example, the hardware register that stored the current ring number had to be emulated to prevent the OS from discovering that it is being virtualized (e.g., to prevent a false error from being raised).

Third, in addition to emulating sensitive, but unprivileged instructions, the VAX VMM must also emulate access to I/O devices. Since only the VMM runs in kernel, devices are no longer accessible to the VM’s guest operating systems (i.e., all physical devices are only accessible from the VMM). Virtualizing I/O access for the VAX processor was especially difficult because I/O was implemented by reading and writing registers that are mapped to physical memory. In the guest VMs, these addresses would no longer mapped to the real registers. To emulate this, the VAX VMM required a small change to kernel software to invoke a ring trap once the I/O memory is updated. The VMM then uses the VMs I/O memory setup to process the real I/O request.

The result of making all sensitive processor instructions privileged is that the VAX VMM system is self-virtualizable. A self-virtualizable VMM can run in one of its own VMMs, permitting recursive construction of VM systems. Self-virtualization is a nontrivial property of processors. Until the introduction of the Intel VT and AMD Pacifica processors, no x86 architectures were self-virtualizable, thus any VMM design for x86 (see Section 11.3) had to address similar problems as in the VAX VMM design. Ways in which the x86 is not self-virtualizable are described by Robin and Irvine [258].

11.2.2 VAX VMM EVALUATION

We evaluate the security of VAX VMM system using the reference monitor principles stated in Chapter 2. The VAX VMM design made several considerations to prevent covert communication channels, in addition to the overt channels controlled by the reference monitor interface. However, it is difficult to guarantee reference monitor tamperproofing and verifiability in real systems, as we will show. Nonetheless, the VAX VMM system has been carefully designed with security in mind, and aimed for an A1-assurance according to the Orange Book [304].

1. Complete Mediation: How does the reference monitor interface ensure that all security-sensitive operations are mediated without creating security problems, such as TOCTTOU?

The requirement for mediation in the VAX VMM is that all security-sensitive instructions in the VAX processor’s instruction set be privileged. As a result, all the instructions that enable updates of VMM state or enable communication via I/O are trapped to the VMM for mediation. Instruction-level mediation requires provides access to all the system resources in need of modification, so TOCTTOU is not a problem.
2. **Complete Mediation**: Does the reference monitor interface mediate security-sensitive operations on all system resources?

In the VAX VMM design virtualization enables mediation of VM operations, such as disk volume and device access. Since the VM OS cannot access any privileged instruction and the changes to the VAX microcode for the VAX VMM ensures that all sensitive instructions are privileged, the VMM has the opportunity to mediate all security-sensitive commands.

3. **Complete Mediation**: How do we verify that the reference monitor interface provides complete mediation?

Privileged instruction traps define the reference monitor interface which provides a reliable mediation if indeed all security-sensitive operations are privileged. Note that the VAX VMM depends on modified operating systems to ensure that I/O commands take the necessary trap. If an attacker can control the VM's operating system code, she can cause a sensitive I/O operation to circumvent the mediation.

4. **Tamperproof**: How does the system protect the reference monitor, including its protection system, from modification?

The VAX VMM reference monitor and protection system are contained within the VMM itself in ring 0. A small amount of user administration is possible from the VMs based on `SECURE` commands. These require a trusted path between the user and the VAX VMM. How the system distinguishes between trusted users and untrusted users is unclear.

Also, there are several interfaces by which the untrusted VM code can invoke trusted VMM code, such as the execution of sensitive instructions and I/O emulation (i.e., device access). While the design of the VAX VMM mediates all these entry points, the design must further ensure that all the data received from the untrusted VM is handled properly. Unlike Multics, which uses gatekeepers to ensure that malicious input data is filtered, the VAX VMM provides no explicit mechanism for such filtering. We imagine that such filtering is done in an ad hoc manner, but its description is not detailed.

5. **Tamperproof**: Does the system's protection system protect the trusted computing base programs?

The VAX VMM contains no trusted code outside of ring 0. Administrators use a trusted path to access code running inside the VAX VMM to perform administrative operations. Other trusted services, such as authentication are also performed in the `secure server` inside the VMM. This approach is unique to the systems that we have examined. On the positive side, it limits assurance to the VMM itself. Also, the required use of a trusted path to access administrative services significantly limits the ways that trusted code can access such services. On the negative side, any change in trusted services requires changes to the VMM.
6. **Verifiable**: What is basis for the correctness of the system’s trusted computing base?

In order to assure correctness of the VAX VMM software to the A1-assurance level of the Orange Book [304], the design \(^1\) was formally specified and analyzed for verify that it satisfied the security policy model. In addition, the system implementation was informally shown to be consistent with the design. Further, the development process of the VAX VMM was tightly controlled. All design decisions and code were reviewed, including any changes to either. The filtering of untrusted input, mentioned under tamperproofing above, was tested thoroughly for response to a variety of legal, illegal, and malformed requests. Even the CPUs were tested for correct implementation of VAX architecture specification.

7. **Verifiable**: Does the protection system enforce the system’s security goals?

The VAX VMM’s security goals are embodied in the Bell-LaPadula and Biba information flow models. If Bell-LaPadula was strictly followed, then no software could leak data to an unauthorized user. Similarly, if Biba was strictly followed, then no software would depend on untrusted code or data. In practice, Bell-LaPadula and Biba are both too restrictive, as the designers acknowledge by the addition of system privileges. However, the ability to verify that the system ensures a concrete security goal in the presence of a set of assigned privileges is a challenging task. The VAX VMM system provided no specialized support for understanding the impact of privilege assignments on the resultant information flows. Presumably, these would be manually verified by the administrators of individual deployments.

The design of the VAX VMM system also provided several countermeasures for controlling covert channels. In addition to an informal analysis, the designers also used a technique called the Shared-Resource Matrix [162, 163] to identify storage channels. Many storage channels were eliminated by preallocation of resources, excepting storage channels due to disk arm movement [275] which required a special technique [160]. Covert timing channels are even more difficult to address, and in the course of the VAX VMM project new means were developed to identify covert timing channels [341] and counter these channels by obfuscating timing [140].

### 11.2.3 VAX VMM RESULT

Despite the diligent efforts of the VAX VMM team, successful pilot deployments, and the A1-assurance preparations, Digital Equipment Corporation (DEC) canceled the project in March 1990. The exact reasons for the cancellation have not been revealed and remain a mystery. However, the reasons why a functional, high security system would be discarded just as it became ready for commercial deployment would be illuminating to those aiming to build high security software in the future.

\(^1\) Specifically, the part of the system design called the *formal top-level specification* was modeled and analyzed.
The VAX VMM kernel was stable in early 1988, self-hosting \(^2\) by mid-1988, and supported DEC’s VMS and Ultrix-32 operating systems by 1989. An external field test was performed in late 1989, and execution performance was found to be “acceptable.”

Undoubtedly, the field test and other business case information was used to determine not to go forward with the VAX VMM as a commercial product. Despite the secrecy, we wonder what aspects of the system caused it not to be sold, given acceptable performance. In the rest of this section, we examine some of the challenges faced in building the VAX VMM kernel system that we should consider when constructing future systems.

First, all the device drivers were run in the VMM, which ensured trusted access to hardware, particularly in these days well before an I/O MMUs (see the discussion on I/O MMUs and direct memory addressing in Chapter 6). However, as operating systems must support a large number of devices, new devices are introduced frequently (and often contain bugs \(^82\)), and the A1-assurance required detailed code reviews for any new code, this would present a challenge for system maintenance.

Second, the combination of Pascal, PL/1, and a significant amount of assembler (nearly 1/4 of the VMM code) was somewhat unusual for operating systems, circa 1990. Particularly the presence of a significant amount of assembly code, over 11,000 LOC, would present challenges in maintaining A1 assurance.

Third, the VAX multiprocessor hardware introduced high performance covert channels (see Section 5.4) on the shared multiprocessor bus. To remedy this problem, the most effective solution given hardware constraints was to use a fuzzy time \(^{140}\) approach whereby certain operations are delayed to prevent an attacker from accurately communicating using such a channel. Delays naturally slow the system’s performance, particularly for I/O operations which are the focus of the delays. Although the research paper reports a slowdown of only 5–6%, the performance analysis is not detailed, so some significant performance degradations may have been a concern for commercialization.

11.3 SECURITY IN OTHER VIRTUAL MACHINE SYSTEMS

We briefly examine the state of security in current VM systems. While the initial motivation for VMs was to better leverage physical resources, security has become a recent focus. Current commercial operating systems have failed to provide adequate, manageable security controls, but building a brand new operating system is no longer a practical option. As in the case of the VAX VMM, a new VM system enables the development of new security controls while still being able to execute existing software, and the ability to execute legacy code is a requirement of any system. However, we must construct VM systems in a manner that avoids the problems of operating systems, such as rootkits \(^{170}\).

\(^2\)By self-hosting, we mean that the VAX VMM could be used to build new versions of the VAX VMM.
IBM’s Processor Resource/Systems Manager [142] (PR/SM, pronounced “prism”) is a type 1 hypervisor that is capable of running a variety of operating systems, including various IBM OSes and Linux, on IBM mainframe hardware, now called the zSeries. PR/SM enables a security administrator to configure virtual machines such that complete isolation is ensured. That is, PR/SM prevents the sharing of the physical systems I/O resources, so no virtual machine can learn about any virtual machines use of system I/O. PR/SM enables such isolation using an Interpretive Execution Facility that only allows virtual machines to execute processor instructions in a controlled manner, which restricts covert channel communications. PR/SM has been evaluated to the Common Criteria EAL 5 in 2005 [14]. Since the focus of the PR/SM product is isolation its evaluation does not aim for MLS protection profiles, such as LSPP (see Chapter 12).

VMware Although VM systems have been around for many years (e.g., IBM z/VM [143]) and have had a reputation for security for quite some time, the introduction of VMware, a VM system for the ubiquitous x86 processor, brought VM systems to the masses [299]. VMware security is enabled by its dynamic translation of resource requests (CPU, memory, I/O) into virtualized commands that VMware VM monitors can mediate. VMware can both restrict the types of commands that can be performed via this translation and determine the interface for mediation.

When a VMware guest VM executes a storage or network request, the VMware VMM mediates the request and determines how this request will actually be implemented. For network requests, the VMware ESX server defines virtual ports which determine the network configuration for that VM (e.g., its MAC address and forwarding tables) independently from its physical platform [46]. A template specification is attached to a VM from which its connectivity can be defined regardless of the host upon which it runs. For storage, traditional I/O requests to a local disk can be converted into a variety of storage system requests [46]. Thus, simpler I/O requests can be authorized and converted into SAN requests, if desired.

The VMware VirtualCenter defines the policy over VMware VM interactions. Resources are grouped into resource pools which partition the CPU and memory resources. Resources pools may be arranged hierarchically and delegated to other subjects. The VMware VirtualCenter uses Windows security controls (see Chapter 7) and roles to define access. The partitioning of resources using pools enables isolation, but if one VM can delegate control of a pool resource to another VM, then a compromise may lead to an information flow that violates the system’s security goals. VMware’s system design includes a variety of hardening features to protect the VMM’s trusted computing base [47], but this does not ensure that a guest VM with control of resource pools cannot be compromised. VMware supports the introspection of guest VMs [106], which may enable the detection of a guest VM compromise from the protected VMM [321]. By running the intrusion detection software (e.g., virus scanners) outside the guest VM protects it from being compromised as well, it does not prevent intrusion or malicious behavior unless the compromise is detectable.
The first major effort that aimed to leverage VMware for security was the NetTop project [205, 136]. NetTop provides end-to-end secrecy protection using virtual machines for isolation, rather than physical machines. The NetTop system is built on a Type 2 VMware system that uses SELinux [229] (see Chapter 9) as its host operating system. Prior to NetTop, organizations with stringent secrecy requirements used isolated networks to prevent leakage of information. Because of the lack of security in commercial operating systems, these organizations required that distinct machines be connected to each network to prevent leakage. Using NetTop, individual VMs can be isolated on the same physical machine, so a single physical machine can be used to connect to multiple isolated networks.

VMware provides the isolation primitives for NetTop. In addition to storage and memory isolation, VMware supports virtualized VLANs, which limits the set of destination systems to which a VM may send network traffic in a LAN. For some time, network switches have supported the configuration of network partitions in a LAN, such that machines can only send to other members of their VLAN partition. VMware supports the assignment of a virtual machines to distinct VLANs, such that a VM can only send packets to other members of its VLAN partition (other VMs or physical machines). There are different ways to configure VMware VMs to use VLANs [46], but we describe one approach here. Using virtual ports, each VLAN is assigned to virtual port, and each guest VM is assigned to the virtual port as well. The virtual port tags the network frames, so that the VLAN switches can ensure that there is no leakage between VLANs.

Why VMware provides isolation primitives, SELinux is used to ensure that any inter-VM communication is authorized according to a mandatory access control policy. VMware by default uses a discretionary access control policy (based on Windows), so resources can be delegated by users and the VMs. Thus, a compromised VM could delegate rights that would cause an unauthorized leak of information. NetTop uses SELinux in two ways: (1) SELinux access control within a privileged VM defines a least privilege [265] policy whereby the permissions of services are contained, even if they are compromised and (2) SELinux defines the resource allotted to VMs by the system, and no delegation is authorized. For example, each virtual machine in NetTop is only authorized to access files with a corresponding label. As a result, a VM in one partition cannot access files in another partition. With the VMware isolation and SELinux policy, NetTop can isolate VMs on the same machine, under the assumption that the trusted computing base cannot be compromised. Even with the least privilege policy, it may be possible for particular trusted services to compromise NetTop should they be compromised.

Xen Xen is another x86 virtual machine system. Xen provides similar isolation guarantees as VMware (e.g., support for VLANs), but the management of inter-VM communication is different. Unlike the discretionary controls of VMware or the SELinux controls within a privileged VM, Xen provides mandatory access control (MAC) at the hypervisor (i.e., VMM) level. The exact implementation of a reference monitor in the Xen hypervisor is an ongoing project, but the aim is similar
to the Linux Security Modules (LSM) interface in Linux [342]. Xen isolation, without leveraging its MAC enforcement, has been applied to isolate web applications [66].

The Xen design aimed for virtualization performance originally, but security has also become a focus for Xen. Xen is a Type 1 VMM consisting of two major components: (1) a hypervisor that runs directly on the hardware and (2) a privileged VM that provides I/O and VM configuration support. The Xen hypervisor provides VM communication primitives, mainly aiming at increasing the performance of I/O emulation between the untrusted VMs and the privileged VM which does the actual I/O processing. Unlike the VAX VMM, Xen uses a VM (i.e., a traditional operating system running in a VM, Linux) to provide I/O emulation to the untrusted VMs.

Security for Xen consists of mediating communication between VMs and controlling the distribution of resources (i.e., access to physical devices and memory). To achieve this, two projects have added a reference monitor interface to the Xen hypervisor. Xen sHype is a reference monitor interface that mainly focuses on inter-VM communication control [263]. The Xen communication mechanisms (i.e., grant tables and event channels [19]) are mediated to ensure that only authorized VMs can communicate. However, the distribution and use of resources is controlled by the privileged VM in an ad hoc manner. Since Xen’s privileged VM partitions disk and memory resources and multiplexes network resources among its VMs, no overt communication using these resources is permitted by default. However, this puts a lot of trust in the ordinary OS (Linux) running in the privileged VM to partition resources correctly. Should these resources be shared among VMs in the future, then there sharing must be mediated by trusted software. The second project, the Xen Security Modules (XSM) reference monitor interface mediates both communication and resource distribution [57]. However, now that the reference monitor hooks are in place, it is necessary to determine the policies necessary to utilize these XSM hooks to enforce practical security goals.

11.4 SUMMARY

Virtual machine systems, and more generally separation kernels, provide a layer of abstraction between the operating system and the physical platform. This enables better utilization of hardware, the ability to run multiple operating systems and their applications on one device, and a point of indirection that may be beneficial for security control. Because the complexity of operating systems has prevented them from being effective and manageable arbiters of security, separation kernels and virtual machine monitors are seen as a layer where security guarantees can be practically enforced. While these systems have been around for a while and have historically supported security, secure VMMs (i.e., in the manner required in Chapter 2) are not readily available. The VAX VMM defined a formally rigorous design and implementation of a secure VMM system, but it was never released as a product. The VAX VMM design demonstrated that VMM security depends on control of all sensitive commands, including those performed by I/O resources. But, this created a conflict, as the number and variety of drivers tends to overwhelm the code management and testing required for formal assurance. MILS separation kernel systems are also being developed, but are specialized systems not yet leveraged by mainstream computing. VMware and Xen are available VMMs that
are being extended to support the enforcement of security, but they emulate I/O in an ordinary operating system. The trade-off between what function belongs in the VMM, and what function can be performed in ordinary operating systems with sufficient security guarantees is an ongoing source of debate.