Chapter 10

Isolation: Mapping and Multiplexing

In the physical world, walls are often erected when isolation is sought. A prison has walls to keep people in; a fortress has walls to keep people out. Some walls have windows and doors (perhaps incorporating fine mesh screens or thick steel bars), so that certain activity on one side cannot influence and/or be influenced by activity on the other side. But even solid walls do not hide all activity—apartment dwellers with loud neighbors know this.

For enforcing security in a computing system, an *isolation mechanism* restricts the environment from

- improperly influencing the system's operation and
- improperly being influenced by reading the system's state or by monitoring usage of system resources.

One important case is where the system is an operating system and its environment comprises clients it is executing; another important case is where the system is a client and its environment comprises other clients. System-provided abstractions to enforce isolation include processes, virtual machines, and containers. All prevent the environment from violating assumptions on which the security of a system might depend, so a system designed for a benign environment can be to be deployed in a more-hostile environment.

Implementations of processes, virtual machines, and containers have much in common: (i) the interposition of mappings in order to control what objects are visible to a system or to its environment and (ii) time multiplexing. Those common building blocks are thus a natural place to start our discussion. Detailed implementations for processes, virtual machines, and containers are then explored in detail.

10.1 Building Blocks for Isolation

10.1.1 Address-Translation Mappings

An *address-translation mapping* relocates some set of memory addresses and causes interrupts[1] for accesses to all others. Formally, it has a domain that is the set N of addresses that principals use and a range $M \cup \{\perp\}$, where M is a set of memory locations and $\perp$ (satisfying $\perp \notin M$) indicates that an address is not being mapped to a memory location. When a principal P executes an assignment statement `x := x + y` while using an address-translation mapping $NMmap_P$ that maps `x` and `y` then the CPU would store at memory address $NMmap_P(\texttt{x})$ the sum of the values at memory addresses $NMmap_P(\texttt{x})$ and $NMmap_P(\texttt{y})$.

Per-principal address-translation mappings $NMmap_P$ with domain N and range $M_P \cup \{\perp\}$ suffice to isolate memory accessed by each principal P if, for every pair P and Q of principals, $M_P \cap M_Q = \varnothing$ holds. This is because $M_P \cap M_Q = \varnothing$ implies that memory location $NMmap_P(n) \in M_P$ a principal P references using name n will necessarily be different from memory location $NMmap_Q(m) \in M_Q$ that another principal Q can access using any name m.

Implementation Considerations. Typical hardware realizations for address-translation mappings use a processor register `MmapReg` (say) to specify a mapping to use. Per-principal memory isolation is achieved by changing the value in `MmapReg` whenever the processor switches from executing instructions for one principal to executing instructions for another.

> **Address-Translation Context Switch Protocol.** Register `MmapReg` specifies mapping $NMmap_P$ whenever principal P is executing. Between executing an instruction for a principal P and an instruction for a different principal Q, register `MmapReg` is updated to $NMmap_Q$. □

Note, changing the value in `MmapReg` typically will cause a processor to flush pipelines and purge caches, so executing Address-Translation Context Switch Protocol could have performance implications.

Any mapping μ from finite sets N to M can be represented by a finite table where the entry for $n \in N$ contains $\mu(n)$. If N is the set of all addresses for a virtual memory, then storing this table would consume, if not exceed, the capacity of a processor's main memory, leaving no room for programs or data. However, smaller tables can be used to describe mappings if every entry in the table maps many elements from N rather than just mapping a single element.

Such a representation is employed in instruction set architectures that interpret a *virtual address* as comprising two parts: a segment name and an offset into that segment.[2] The elements of N are virtual addresses, and each segment

[1] Some reserve the term *interrupt* for describing asynchronous transfers of control and use terms *fault* or *trap* for synchronous transfers of control. We will use the single term *interrupt* for synchronous and for asynchronous transfers of control, since processor hardware follows the same protocol in either case.

[2] Figure 7.8 (page 153) depicts a typical segmented virtual memory.

is a block of consecutive memory words.[3] A *segment descriptor* gives a triple $\langle b, len, m \rangle$, with $0 \leq len$, and defines a mapping between the two subsets

$$\{b, b+1, \ldots, b+len-1\} \subseteq N$$
$$\{m, m+1, \ldots, m+len-1\} \subseteq M$$

where every virtual memory address $n \in N$ satisfying $b \leq n < b+len$ is mapped to memory location $m + (n-b)$. Therefore, a set *Segs* of segment descriptors, where each descriptor corresponds to a disjoint block of N, can be interpreted as defining a mapping *NMmap* from names $n \in N$ to memory addresses, as follows.

$$NMmap(n)\text{:} \begin{cases} m_i + (n - b_i) & \text{if } \langle b_i, len_i, m_i \rangle \in Segs \text{ and } b_i \leq n < b_i + len_i \\ \perp & \text{otherwise} \end{cases}$$

How *Segs* is represented depends on the instruction set architecture, as do the details of how `MmapReg` specifies mapping $NMmap(\cdot)$. `MmapReg` is called the *segment table register* on some computers; it contains the memory address for a table of segment descriptors (with that table itself a segment). Other computers eliminate a level of indirection by providing a small number (typically 2 or 4) of registers `MmapReg`$[i]$, each storing a segment descriptor. Here, *Segs* is the (small) set of segment descriptors contained in those registers. A third scheme, found in early computers, supported name mappings where *Segs* was defined by using two registers: *base register* `Base` and *limit register* `Lim`. This pair of registers defined a set *Segs* that contained the single segment descriptor $\langle 0, L, B \rangle$ for L the value contained in `Lim` and B the value contained in `Base`. An address $n \in N$ satisfying $0 \leq n < L$ was relocated to $n + B$; if $L \leq n$ then an interrupt was generated.

Incorporating Access Control. The set of operations that a principal is authorized to use for accessing some region of memory can be described by associating per-principal address-translation mappings with sets of operations. A simple implementation would extend each segment descriptor with a set *Ops* of permitted operations. Thus, a segment descriptor $\langle b, len, m, Ops \rangle$ would specify an address-translation mapping $NMmap(\cdot)$ for accessing a region of memory by using operations in $op \in Ops$ but no other operations.

$$NMmap(n)\text{:} \begin{cases} m_i + (n - b_i) & \text{if } \langle b_i, len_i, m_i, Ops_i \rangle \in Segs \\ & \quad \text{and } b_i \leq n < b_i + len_i \text{ and } op \in Ops_i \\ \perp & \text{otherwise} \end{cases}$$

[3] By further dividing each segment into fixed-size pages, a contiguous region of main memory is not needed for holding the contents of a segment, nor is it necessary for all pages comprising a segment to be resident. A virtual address now comprises three parts: a segment name, a page name, and an offset into the page. An additional mapping, implemented by hardware, translates each page name to the main memory address of the page frame that holds this page. The existence of paging should be transparent, and thus it is ignored in this chapter.

10.1.2 Time Multiplexing

If a bank of registers and/or region of storage is being *time multiplexed* among a set of principals then periods of exclusive access rotate among those principals. When each period of exclusive access by a principal P ends then the values in the register bank and storage region are saved in isolated storage $memSaved[P]$ (say), to be restored at the start of the next period of exclusive access by P:

> **Time-Multiplexing Context Switch Protocol.** At the end of each period of exclusive access by P to *mem*, a register bank and/or a storage region:
>
> 1. Suspend execution of P.
> 2. $memSaved[P] := mem$;
> 3. $Q :=$ `sched()`;
> 4. $mem := memSaved[Q]$;
> 5. Resume execution of Q. □

Note the considerable flexibility in the scheduling policy implemented by $Q :=$ `sched()` to select the principal Q that is next resumed. It might be *round-robin*, where each principal receives access for a fixed period and in a rotation. Or, to satisfy performance requirements, a scheduler might make selections according to past resource consumption and/or system state.

10.2 Processes

The environment for a program executing on a computer includes: (i) a predefined instruction set implemented by hardware, and (ii) a state comprising processor registers and memory. So multiple programs being executed together on the same computer can affect each other's environments if they share memory and/or registers. A *process* is an executing program whose memory and registers are unaffected by the actions of other processes. Such isolation is attractive for enforcing security in a computer where programs are executing on behalf of different principals.

10.2.1 Memory and Register Isolation

Isolation for a process's main memory can be enforced by using per-process address translation mappings. Isolation for the processor's registers is achieved by time multiplexing. Notice that provided `MmapReg` is a processor register, steps 2 and 4 of Time-Multiplexing Context Switch Protocol (page 304) implement Address-Translation Context Switch Protocol (page 302). Also, because the program counter is a processor register, step 4 of Time-Multiplexing Context Switch Protocol causes that register to be loaded and, therefore, each process resumes execution where it left off when it was suspended.

10.2.2 Implementing Time Multiplexing

Time-Multiplexing Context Switch Protocol has a straightforward implementation on any processor that supports timer interrupts. We start by sketching typical processor support for interrupts, followed by a discussion of typical processor support for timer interrupts.

Interrupts. On a processor that supports interrupts, an *interrupt handler* for each class I of interrupts is specified by storing its address in `IntHndlr`$[I]$`.new`, and a Boolean flag `IntHndlr`$[I]$`.enbl` indicates whether class I interrupts are *enabled* or *disabled*.[4]

> **Interrupt Processing.** Once an interrupt of class I has been *raised*, it remains *pending* until the value of `IntHndlr`$[I]$`.enbl` signifies that interrupts of this class are enabled. While enabled, interrupts in a class are *delivered* in the order raised; the instruction set architecture defines an ordering (say) I_1, I_2, ..., I_n for interrupt delivery across different classes. And a processor delivers an interrupt of class I as follows.
>
> – The current processor state is pushed onto a stack `IntOldStates` stored in main memory.[5]
>
> – Values are loaded into the processor registers from `IntHndlr`$[I]$`.new`, causing the code for an interrupt handler to start executing because of the new values in the program counter, general-purpose registers, etc. □

Timer Interrupts. A processor that supports timer interrupts usually will have a `Timer` processor register. T seconds after `Timer` is loaded with an integer value T, a *timer interrupt* is raised.

Time-Multiplexing Protocol Implementation. `Timer` is loaded with a value that bounds the interval of exclusive access. For step 2 of Time-Multiplexing Context Switch Protocol, interrupt handler `TimerHndlr` for timer interrupts pops the top entry on `IntOldStates` to *ProcState*$[P]$, where P is the process that was just suspended. `TimerHndlr` also performs step 3 through step 5. Specifically, `TimerHndlr` (or system software it invokes) selects some process Q to next execute, loads `Timer` with some integer value, and restores the processor registers from *ProcState*$[Q]$. Execution of Q thus resumes, with the appropriate address translation mapping and with values for the program counter and other processor registers that were saved when execution of Q was last suspended.

[4] To simplify the exposition, we assume that `IntHndlr`$[\cdot]$ is implemented by a processor register, even though this information actually might be stored by a table located at a pre-specified and fixed memory location or at the address contained in some a pre-defined processor register. When `IntHndlr`$[\cdot]$ is stored in memory, that region would be excluded from the range of all address-translation mappings.

[5] Depending on the instruction-set architecture, `IntOldStates` might be stored at some pre-specified address in main memory or stored at a location designated by the contents of a fixed processor register.

Protection by using Processor Modes. A process that updates `MmapReg`, `Timer`, `IntHndlr`[·], or *ProcState* could compromise isolation being enforced by address translation (§10.1.1) and time multiplexing (§10.1.2). Therefore, instruction set architectures distinguish between:

- *User-mode instructions.* Instructions in $Inst_U$ access only a subset of the processor registers, and they access main memory using an address-translation mapping.
- *System-mode instructions.* $Inst_S$ adds to $Inst_U$ instructions to read/write all of the processor registers and main memory, as well as instructions concerned with performing input/output operations.[6]

Such instruction set architectures then facilitate isolation by (i) providing a `mode` register to control whether instructions from $Inst_U$ or $Inst_S$ are available for execution, (ii) signaling a *privilege interrupt* when an instruction ι is executed unless `mode` = $Inst_S$ or $\iota \in Inst_U$ hold, and (iii) excluding from $Inst_U$ any instruction that can cause changes to `MmapReg`, `Timer`, `IntOldStates`, or `IntHndlr`[·]. A process executing with `mode` = $Inst_U$ executes instructions from $Inst_U$ and, thus, is not able to compromise the integrity of address translation or time multiplexing by updating `MmapReg`, `Timer`, `IntOldStates`, or `IntHndlr`[·] or compromise isolation by initiating an input/output operation that stores into *ProcState* or the memory of another process.

But limiting execution to instructions from $Inst_U$ is too restrictive for interrupt handlers and certain other system software. `TimerHndlr`, for example, updates `Timer` and `MmapReg` prior to resuming a process, so `TimerHndlr` must execute with `mode` = $Inst_S$. To have assurance that isolation still will be maintained, we must have some basis for trust that any software executing with `mode` = $Inst_S$ will not misbehave. This trust could derive from an analysis of that code, from the presence of mechanisms to restrict execution, or a combination. Obviously, assurance is easier to establish if only certain software can execute with `mode` = $Inst_S$—a small fragment of system software, for example.

Extending the User-mode Instruction Set. Attempting to execute a system-mode instruction when `mode` = $Inst_U$ holds will cause a privilege interrupt. So system software is written to provide safe versions of any functionality needed by user-mode software. By first checking arguments it is passed, that system software prevents system-mode instructions it will execute from violating the integrity of address translation and time multiplexing.

Processors typically provide a user-mode *supervisor call* instruction `svc` for allowing user-mode execution to invoke system software. Execution of `svc` by a process P causes an interrupt of class `svcInt`. That interrupt handler executes with `mode` = $Inst_S$; it parses the operands passed with the `svc`, and then it invokes system code to perform the requested service. Once the requested service

[6] Input/output devices typically do not use address-translation mappings when accessing memory and, therefore, instructions to control input/output operations are not user-mode instructions.

has been completed: a process Q is selected to next execute, `Timer` is loaded with some integer value, and the processor state is loaded from *ProcState*[Q], thereby causing execution of Q to be resumed.

10.2.3 Deciding on Kernel Functionality

An `svc` sometimes can be used to undermine process isolation. To illustrate, here are functions that an operating system kernel might make available though `svc`'s.

- *Interprocess communication and synchronization primitives.* Such primitives facilitate cooperation among processes and, therefore, they allow one process to influence the execution of another. But processes that can influence each other are no longer completely isolated from each other.

- *Dynamic allocation primitives.* Resource usage by one process becomes visible to other processes through any delays that arise from allocation requests when there is contention for resources. Isolation is thus compromised.

How much might a given `svc` compromise process isolation? To answer would require analyzing any code that gets executed after the `svc` is invoked as well as analyzing any code that reads the updated state—a potentially large body of code. So a formal analysis is unlikely to be feasible, and an informal analysis might overlook things. Furthermore, any weakening of process isolation that `svc`'s cause could be offset if user-mode software becomes simpler because of the added kernel functionality. We should be more inclined to trust user-mode software that is simpler, a benefit that might outweigh the weaker isolation.

Design and implementation flaws in a kernel are another cause of compromise for process isolation. For example, a bug in an interrupt handler might allow system code invoked by one process to contaminate *ProcState* being stored for some other process. With processes likely to serve as a system's principals, a kernel will be part of the system's trusted computing base. That argues for making design trade-offs in favor of assurance and, therefore, argues that we should prefer small and simple kernels over kernels that are rich in functionality.

The trade-offs associated with deciding what functionality to put in a kernel have led to various design philosophies. Two important views are discussed below. They differ in (i) assumptions about the feasibility of imposing a single security policy on an entire user community and (ii) the relative importance of assurance over functionality. Both embrace the position that the kernel's design, if not its code, should be amenable to a formal analysis.

Separation Kernels. A *separation kernel* implements processes and interprocess communication, but provides no additional functionality. So the environment it creates is indistinguishable from a distributed system where each

process executes on a separate processor and uses message passing for communication with other processes. Because a separation kernel offers only limited functionality, it can be small and simple.

A justification for trust is one motivation for the small size and simplicity. But the limited functionality that a separation kernel provides also reflects the widely held belief that policy should be separable from mechanism. Mechanisms a kernel might provide for accessing resources would have to enforce some class of security policies, yet no single class is well suited for all applications. A separation kernel sidesteps the issue—it doesn't implement operations for resource access, so it avoids the need to choose access control policies.

Security Kernel. For computing systems intended to store and process objects, the trusted computing base comprises the software that implements process isolation plus the software that controls access to objects according to some policy. A *security kernel* provides exactly this functionality (and no more). It implements processes and it mediates all accesses those processes make to objects.

When a security kernel is used, assurance for the entire trusted computing base follows from the assurance argument for the security kernel. That is attractive, because a security kernel can be small which facilitates establishing its assurance. In addition, when a security kernel is in use, a single implementation of isolation is protecting both the reference monitor and the implementation of process isolation. So less total mechanism is involved, and there is less to trust.

Use of a security kernel, however, does mean that a single security policy must suffice for mediating accesses by all processes to all objects. This is a mixed blessing. When different applications require radically different policies—discretionary access control for some and mandatory access control for others. for example— then the security kernel only can enforce a weak policy, which each application must extend.

10.2.4 *Hardware Rings of Protection

In a hierarchically-structured system, each *layer* maintains state and provides operations for use by higher layers but not by lower layers. The lowest layer, implemented as hardware, provides machine language instructions; these operate on registers and main memory. Other layers are implemented in software. They hide, redefine, and/or augment operations exported by lower layers. A defining characteristic for all layers in a hierarchical systems is:

> **Trust in Hierarchical Systems.** In a hierarchical system, higher layers may trust lower layers, but lower layers do not trust higher layers. □

Thus, correct operation of a layer is allowed to depend on correct operation of lower layers. Higher layers, however, may well attempt to subvert lower layers. One way to rule out such attacks is additional mechanism. That is what we next discuss.

Enforcing Isolation from Higher Layers. Because lower layers are trusted by higher layers, execution in a given layer is authorized to read and write all state being maintained by higher layers but not authorized to read or write state maintained by lower layers. That is, for any layers L and L',

$$L' \prec L \;\Rightarrow\; (\mathit{read}(L') \supseteq \mathit{read}(L) \;\wedge\; \mathit{write}(L') \supseteq \mathit{write}(L)) \tag{10.1}$$

where: (i) relation $L' \prec L$ on layers holds[7] when layer L' is below (hence, may be trusted by) layer L, (ii) set $\mathit{read}(L)$ enumerates parts of the state that code in layer L is authorized to read, and (iii) set $\mathit{write}(L)$ enumerates parts of the state that code in layer L is authorized to write.

CPU hardware for address translation often will directly enforce (10.1) by implementing a metaphor of nested or concentric *rings* and providing

- a register `curRing` to associate a ring with current execution, and
- a means to specify access restrictions for execution in each given ring.

Each layer L is associated with a non-negative integer value $\mathit{ring}(L)$ that corresponds to the ordering of layers:

$$L' \prec L \;\Rightarrow\; 0 \le \mathit{ring}(L') < \mathit{ring}(L). \tag{10.2}$$

Access restrictions on the code being executed are imposed according to the value of `curRing` and two fields we add to segment descriptors. Specifically, a segment descriptor $\langle b_i, \mathit{len}_i, m_i \rangle$ for mapping names n satisfying $b_i \le n < b_i + \mathit{len}_i$ (as discussed on page 303) is extended with fields

- rb_i an integer specifying a *read bracket* comprising the set of layers L satisfying $0 \le \mathit{ring}(L) \le rb_i$, and
- wb_i an integer specifying a *write bracket* comprising the set of layers L satisfying $0 \le \mathit{ring}(L) \le wb_i$.

A read access to a name in segment $\langle b_i, \mathit{len}_i, m_i, rb_i, wb_i \rangle$ is authorized only if $0 \le$ `curRing` $\le rb_i$ holds, because then the currently executing layer is in the read bracket for the segment being accessed; an access violation interrupt occurs otherwise. If rb_i is negative then, reading from the segment is never allowed. Writes are analogous, but restricted by the write bracket.[8] Typically, wb_i and rb_i are defined in such way that $wb_i \le rb_i$ holds, so that code can read what it has written.

[7] In a hierarchically structured system, relation $\prec$ is total, irreflexive, asymmetric, and transitive. So it is impossible to have both $L' \prec L$ and $L \prec L'$ hold. And for every pair of layers L and L', either $L' \prec L$ or $L \prec L'$ holds.

[8] So `curRing` can be seen as a generalization of `mode`, with smaller values for `curRing` authorizing access to additional state rather than to additional instructions. The distinction between state and instructions can be ignored here, because the effect of executing any instruction is to perform reads and writes to main memory and processor state—restricting access to state is thus equivalent to disallowing execution of certain instruction instances.

Notice that (10.1) is satisfied by a segment $\langle b_i, len_i, m_i, rb_i, wb_i \rangle$ no matter what values are used to define the read and write brackets. Here is a proof. Consider a name n satisfying $b_i \le n < b_i + len_i$, and suppose that $L' \prec L$ holds. We prove $read(L') \supseteq read(L)$ holds, as required by (10.1), by showing that if $n \in read(L)$ holds then so does $n \in read(L')$. If a read to some name n succeeds while layer L is executing (so `curRing` $= ring(L)$ and $n \in read(L)$ hold) then $0 \le ring(L) \le rb_i$ must be satisfied. We have $0 \le ring(L') < ring(L)$ from $L' \prec L$ and (10.2), so $0 \le ring(L') \le rb_i$ follows from $0 \le ring(L) \le rb_i$. Thus, reading n while executing in layer L' does not cause an access violation: $n \in read(L')$ holds, as we needed to show. The argument to prove conjunct $write(L') \supseteq write(L)$ in the consequent of (10.1) is analogous.

Operation Invocation. In hierarchically-structured systems, a `call` instruction executed by code at layer L is allowed to proceed only if (i) the destination is the *gate* for an operation *op* exported by some lower layer L' and (ii) *op* is not being hidden or redefined by any interposed layer L'' where $L' \prec L'' \prec L$. Enforcement of these restrictions can be controlled by incorporating information into descriptors. The descriptor for each segment i now would also include:

- $nGates_i$, the number of gates that segment i contains. Entry points are enumerated in the first $nGates_i$ words of segment i: word 1 contains the address in segment i of the entry point for operation 1, word 2 contains the address in segment i of the entry point for operation 2, and so on through word $nGates_i$.

- xb_i, a non-negative integer that the processor loads into `curRing` whenever an instruction from segment i is being executed. So in order to specify that segment i stores code for a layer L then xb_i is set equal to $ring(L)$.

- cb_i, the largest value of `curRing` from which a **call** is permitted to a gate in segment i. By setting $cb_i = xb_i + 1$, operations defined by gates in segment i are hidden or redefined by the layer immediately above; and if $cb_i > xb_i + 1$ then higher layers can themselves directly invoke operations defined by gates in segment i.[9]

Execution of "`call` *op*" then proceeds as follows, where *op* is presumed to be given as a segment name i and offset p in words. To start, the hardware checks that $p \le nGates_i$ and $xb_i <$ `curRing` $\le cb_i$ hold, thereby establishing that (i) *op* identifies a gate in segment i and (ii) *op* is visible to the layer executing the `call`. If these checks succeed then execution of the `call` pushes onto a stack the return address and the value of `curRing`, loads xb_i into `curRing`, and loads into the program counter the value found in word p of segment i (thereby branching to the entry point of *op*). When execution of the operation completes, that code

[9] This scheme does not offer the flexibility to specify that only certain specific operations that a layer exports should be hidden or redefined in higher layers. Such flexibility could be attractive, but it is not part of the support existing modern CPUs provide for rings.

is responsible for restoring the program counter and `curRing` by popping the stack, thereby returning control to the caller.

Care is required when referencing arguments from within the body of *op*—otherwise, confused deputy attacks (see page 135) are possible. Of concern is a caller that (i) is executing in some layer not in the read (write) bracket for the segment named in an argument *arg* and (ii) passes *arg* to an operation in some lower layer L that is in the read bracket. So the caller is not itself authorized to read (write) *arg* but, by exploiting a confused deputy in layer L, effects access to *arg* nevertheless. The obvious defense is for each operation to check whether its arguments are accessible to their callers. This check can be performed in software by consulting the corresponding segment descriptor for each of the arguments. Some processors offer a separate addressing mode to facilitate such checks. This addressing mode allows accesses to be made under a temporarily increased value for `curRing`, such as the value in `curRing` when the `call` executed.

Layers versus Processes. Layers and processes are both concerned with system structure. They are orthogonal constructs: a process might be layered, or a layer might itself be implemented by a set of processes. Both constructs facilitate decomposing a larger system into smaller units that each can be understood and analyzed separately. So both constructs help in assurance arguments. The two types structures do differ in what isolation they enforce, each allowing different assumptions to be made about the environment. However, each construct helps with the Principle of Least Privilege by enforcing isolation that restricts what parts of the overall system each individual component can access. So, in both cases, structure is being leveraged for defense.

10.3 Virtual Machines

An instruction set architecture that is implemented by software is known as a *virtual machine*. Execution of a virtual machine instruction can be fast if it involves executing only a single instruction on the underlying processor. Opportunities for such *direct execution* are more frequent if a virtual machine's instruction set resembles the instruction set for the underlying processor. A second benefit of having such a resemblance is that existing programs written for the underlying processor do not have to be modified or even recompiled for execution by the virtual machine.

A *virtual machine manager* (VMM), also known as a *virtual machine monitor* or a *hypervisor*, is a software layer that executes on some underlying processor to provide one or more virtual machines. A *type I* VMM runs on bare hardware; a *type II* VMM runs above a software layer (typically, an operating system); and a real or virtual machine V is considered *self-virtualizing* if V can run a VMM that implements virtual machines having the same instruction set as V.

A VMM enforces isolation for the memory, registers, and input/output devices associated with each virtual machine, as well as isolating resources used internally by the VMM. Input/output devices are the sole means for one virtual machine to communicate with another virtual machine or to interact with the environment. In comparison, an operating system typically will provide primitives (e.g., `svc`'s) so that processes can share resources, communicate, and synchronize. So the activities of one process can influence the activities of another. The isolation between processes thus is somewhat weaker than the isolation between virtual machines, which have no ways to influence each other. In addition, because a VMM offers fewer services than an operating system, the VMM can be much smaller, which implies—all else equal—that a VMM is less likely to have vulnerabilities than an operating system.

This strong isolation of virtual machines and higher assurance of VMMs has proved useful in various settings.

- Cloud providers employ VMMs to give each customer an illusion of sole tenancy on some computers. The customer often can select an operating system and even an entire software stack to be loaded and run on each of those computers.[10]

- In an enterprise datacenter, VMMs enable a single computer to run multiple virtual machines, each hosting a server. If (as usually is the case) the servers are not busy most of the time, then this *server consolidation* avoids the overheads of running each server on a separate lightly-loaded dedicated machine. Moreover, with server consolidation, each server can be run on a (virtual) processor configured to best suit that server. Were the servers instead executed as processes under a single operating system then they all would be scheduled according to the same policy, likely resulting in worse performance.

- On the desktop, running a VMM can compensate for weak operating system security, because the isolation of virtual machines limits what attacks are possible when an operating system has become compromised. For example, if one virtual machine hosts applications to support personal banking and another is used for web-browsing then content that is downloaded during browsing would be prevented by the VMM from subverting the applications that access your a bank account.

Monitoring and debugging also are facilitated. A typical VMM provides a *virtual console* for each virtual machine it implements. The virtual console allows an executing virtual machine to be paused by a human operator, who then can inspect or change that virtual machine's memory and (virtual) processor registers. Application software, an operating system, or even a VMM itself now

[10] This kind of cloud computing is know as *infrastructure as a service* (IaaS). With *platform as a service* (PaaS), the customer is offered a computer that runs some pre-configured software stack. And *software as a service* (SaaS) provides customers with specific applications and/or databases that run in a datacenter.

can be debugged simply by running that software in some virtual machine and using the associated virtual console.

10.3.1 A VMM Implementation

Isolation for virtual machines can be enforced by using the same building blocks we employed above to enforce per-process isolation:

- address translation (§10.1.1) ensures that no virtual machine can retrieve or alter main memory allocated to another virtual machine or to the VMM, and
- time multiplexing (§10.1.2) ensures that no virtual machine can retrieve or alter the (virtual) processor registers of other virtual machines.

Address Translation for Virtual Machines. To ensure that the main memory for different virtual machines occupies non-overlapping memory regions of the underlying processor's main memory, the VMM establishes an address-translation mapping $VMap_V$ for each virtual machine V. $VMap_V$ is a mapping from set M of the addresses in the virtual machine V's memory to set $M_V \cup \{\perp\}$, where M_V are addresses for a region of the underlying processor's memory, $\perp$ signifies that an address is not mapped, and the following disjointness condition holds:

$$V \neq V' \Rightarrow (M_V \cap M_{V'} = \varnothing).$$

To isolate itself, memory used by the VMM is excluded from M_V for every virtual machine V.

An operating system or other software executing in a virtual machine V must be able to install its own mapping $NMmap_V$ (say) by loading V's (virtual) `MmapReg` register. Thereafter, a reference by V to address n should access memory location $VMap_V(NMmap_V(n))$. To achieve that effect, it suffices if, during execution of V, the underlying processor's `MmapReg` register specifies *strict composition*[11] $NMmap_V \circ VMap_V$ of address-translation mappings $NMmap_V$ and $VMap_V$:

$$(NMmap_V \circ VMap_V)(n)\text{:} \begin{cases} VMap_V(NMmap_V(n)) & \text{if } NMmap_V(n) \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

The expense to compute $NMmap_V \circ VMap_V$ from $NMmap_V$ and $VMap_V$ as part of every context switch would be prohibitive. But it is feasible for VMM to maintain a local data structure $NMVmap_V$ that equals $NMmap_V \circ VMap_V$ and for VMM to update $NMVmap_V$ incrementally each time virtual machine V updates $NMmap_V$. It suffices that the VMM maintain a shadow copy $SNMmap_V$ of the representation for $NMmap_V$ being stored in V's memory.

[11] A function F is defined to be *strict* iff $F(\perp) = \perp$ holds. And *composition* $F' \circ F''$ of two functions $F'\text{:}\, D' \rightarrow R'$ and $F''\text{:}\, D'' \rightarrow R''$, where $R' \subseteq D''$ holds, is defined to be function $F\text{:}\, D' \rightarrow R''$ such that $F(x) = F''(F'(x))$.

To keep $SNMmap_V$ current, the VMM must learn about updates to V's representation for $NMmap_V$. That is easily arranged on a processor having a *translation cache* that stores mapping information for recent accesses. VMM can infer the address of V's representation for $NMmap_V$, because VMM receives control whenever V attempts to load `MmapReg`—the address of V's representation for $NMmap_V$ will be an argument $Segs_V$ (say) to the load instruction. So translation cache entries can be marked in a way that causes the VMM's interrupt handler to get control whenever the VMM must update $SNMmap_V$.

Time Multiplexing for Virtual Machines. As discussed for implementing processes, limiting execution to user-mode instructions suffices to protect the integrity of the address-translation and time-multiplexing implementations. Therefore, virtual machines are executed with `mode` = $Inst_U$ on the underlying processor. Time multiplexing also requires that, for each virtual machine V, VMM maintains a data structure $VM[V].procState$ containing fields to store virtual machine V's registers (e.g., program counter, general-purpose registers, `MmapReg` register, `mode` register, and `IntHndlr`$[\cdot]$) as well as other fields used to derive the values to be loaded into the underlying processor's registers for running V.

System-mode Instructions. The VMM intercepts and then emulates system-mode instructions that a virtual machine V executes.

- *Intercept.* Virtual machines are executed with `mode` = $Inst_U$, causing instructions ι from $Inst_S$ to raise a privilege interrupt.[12]

- *Emulate.* A VMM-installed handler for privilege interrupts emulates ι by updating information being stored by VMM in $VM[V].procState$ and/or by executing system-mode instructions (with `mode` = $Inst_S$) on the underlying processor.

Some system-mode instruction have straightforward emulations. For example, a virtual machine V's update to its `MmapReg` register is emulated by VMM updating the value for that register stored in $VM[V].procState$. Other emulations are more complicated. For example, when the `Timer` register of virtual machine V is loaded with a value T, then emulation of the subsequent timer interrupt is facilitated if a field $VM[V].NextTimerInt$ is being maintained to store the earliest possible time[13] `TimeNow` + T for that timer interrupt.

[12] An assumption is being made here about the instruction set architecture. On some processors, however, a privilege interrupt is not raised by an attempt to execute an instruction $\iota \notin Inst_U$ when `mode` = $Inst_U$ holds. Methods to handle that case are discussed in §10.3.2.

[13] `TimeNow` is assumed to be a register that always contains the current time. Most hardware processors have such a register. If such a register is not available then it can be emulated by a variable *TimeOfDay* maintained by the VMM. The VMM records in another variable *LastIT* the value it last loaded into the interval timer. And whenever the VMMs handler for timer interrupts is invoked, *LastIT* is added to *TimeOfDay*.

Interrupts. Emulation also is used to implement a virtual machine's interrupts. For the interrupt processing sketched earlier (page 305), the VMM would maintain queues $VM[V].IntPending[I]$ that contain, in order of occurence, an element for each class I interrupt that has been raised at virtual machine V but not yet delivered. In addition, VMM emulates stack `IntOldStates` on a virtual machine by maintaining a separate stack $VM[V].IntOldStates$ of processor states for each virtual machine V. The VMM emulation for interrupt delivery is thus described by the following code, where I_1 through I_n are the classes of interrupts.

for $Int := I_1,\ I_2,\ \ldots,\ I_n$ **do**
  if $VM[V].IntHndlr[Int].enbl\ \wedge\ \neg$**empty**$(VM[V].IntPending[Int])$
    then push $VM[V].procState$ **onto** $VM[V].IntOldStates$
      $VM[V].procState\ :=\ VM[V].IntHndlr[Int].new$

An interrupt should be raised at a virtual machine V immediately for some instructions (e.g., `svc` or a system-mode instruction while the virtual machine is in user-mode). For other *delayed interrupt* instructions (e.g., loading the interval timer or initiating an input/output operation), an interrupt at V should be raised only after some external activity finishes. A VMM can emulate either kind of instructions, but only if the VMM receives control when the instruction is executed by V. So it suffices that execution by V of both kinds of instructions raise an interrupt on the underlying processor. The following conditions ensure those interrupts will be raised.

- Execution of a user-mode instruction that should immediately raise an interrupt at V will immediately raise an interrupt at the underlying processor.
- All delayed interrupt instructions are system-mode.

The VMM emulation for a delayed interrupt instruction initiates some requested activity which, when completed, raises an interrupt at V. Moreover, for improved performance, a VMM might combine requests from multiple virtual machines into a single request that the underlying processor makes. We see this in systems where a VMM uses a single (real) disk to implement separate virtual disks for each virtual machine. Input/output requests to virtual disks would be made by executing delayed interrupt instructions. But if the VMM combines these input/output operations into a single input/output operation to some real disk then a single interrupt will be raised to signify the completion. So, upon delivery of this single interrupt, the VMM would have to raise input/output interrupts at multiple virtual machines.

Resuming a Virtual Machine. Execution of a virtual machine V is resumed by loading the underlying processor's state using the information managed by the VMM. Specifics are given in the table that follows, where `Q` is the maximum

time-slice a single virtual machine may execute uninterrupted. Notice, the underlying processor's `mode` register does not reflect the virtual processor's mode while a virtual machine V is executing—the value of the virtual processor's `mode` register appears in $VM[V].procState$.

register	source for value to load
program counter	program counter in $VM[V].procState$
general purpose	register value in $VM[V].procState$
`mode`	$Inst_U$ no matter what is stored in $VM[V].procState$
`MmapReg`	$(VM[V].VMap_V) \circ (VM[V].procState.MmapReg)$
`Timer`	$\min(\texttt{TimeNow} + \texttt{Q}, \min_W(NextTimerInt_W))$

10.3.2 Binary Rewriting

On some processors, executing a system-mode instruction when `mode` = $Inst_U$ holds does not cause a privilege interrupt.[14] These *non-virtualizable* instructions are not intercepted but they must be emulated. That emulation can be invoked if, in any code that a virtual machine executes, we have replaced each non-virtualizable instruction with code that invokes the VMM. That program rewriting requires:

(i) A means to identify each non-virtualizable instruction and replace that instruction with other code.

(ii) A mechanism to invoke the VMM from within that replacement code.

Two approaches to (i) are prevalent in practice: binary translation (described next) and paravirtualization (described after, in §10.3.3). For (ii), many processors include a special *hypervisor call* instruction that, when executed, raises an interrupt associated with a distinct class; the corresponding interrupt handler is configured to invoke the VMM. Absent a hypervisor call instruction, the supervisor call instruction (`svc`) discussed earlier can be used, provided the VMM can distinguish `svc` executions intended to invoke operating system services from `svc` executions intended to invoke a VMM instruction emulation.[15]

Binary Translation Implementation. The process by which an *input executable* of a program in some *input machine language* is converted into an *output executable* for an equivalent program in some *output machine language* is known as *binary translation*.[16] We might want to migrate software that runs on one machine onto different hardware, or we might want existing hardware to execute

[14] The Intel X86 instruction set architecture is a noteworthy example. It has a few instructions (e.g., `IRET` and `POPF`) whose execution does not cause a privilege interrupt but has different effects depending on the value of `mode`.

[15] To distinguish an `svc` execution intended to invoke the VMM, a special operand value, never used by the operating system to specify a service, would suffice.

[16] A machine language program is commonly called a *binary*. So a program to convert between machine languages is doing translation from one machine's binary to another's, hence the name "binary translation".

programs written for hardware that does not yet exist. By taking a liberal view of what constitutes equivalent programs, binary translation also can be used to add instrumentation to machine language programs so that run-time behavior will be measured.

In *static binary translation*, a translator produces the entire output executable B' from input executable B before execution of B' starts. Static binary translation is impossible if run-time information determines which parts of B constitute instructions and which are representing data values, since a translator would be unable able to ascertain what fragments of the input executable should be converted. Uncertainty about instruction boundaries arises when instruction formats are variable-length, instruction alignment has few restrictions, computed branch destinations are supported, and/or instructions are mixed with data.

Dynamic binary translation converts instructions in the input executable only when those instructions are reached during execution. By alternating between execution and translation of instruction blocks, the translator can know and use the processor state produced by execution of the last block for converting the next block. That processor state not only provides the translator with the starting location for the first instruction in the block to be converted but also provides the translator with values needed for calculating the destination of a computed branch if that is the first instruction in this block.

> **Translation and Execution as Coroutines.** Given an input executable B, an offset d indicating the location in B for the next instruction to execute, and values to load into processor registers before execution commences:
>
> (1) Construct B' by translating instructions in B, starting at offset d and continuing until reaching a branch instruction ι whose destination is being computed.
>
> (2) Translate branch instruction ι into an instruction that transfers control to the translator. Use the offset for ι in B as the offset value d passed to the translator; the processor register values passed are whatever values those registers contain when the translation of ι is reached.
>
> (3) Execute B'. □

When execution of B' in step (3) reaches the translation of ι, control transfers to the translator (thereby returning to step (1)), which resumes converting B, starting with instruction ι.

VMM use of Dynamic Binary Translation. Dynamic binary translation enables a machine language containing non-virtualizable instructions to be implemented on a processor that has some form of hypervisor call.

Implementing Virtual Machines by using Binary Translation.

- Implement a dynamic binary translator that replaces system-mode instructions with hypervisor calls. By definition, non-virtualizable instructions are system-mode instructions, so the translator will replace all non-virtualizable instructions.
- Implement an interrupt handler for hypervisor calls. This handler should contain code for emulating each system-mode instruction.
- Modify the VMM code for Resuming a Virtual Machine (page 315) so that it transfers control to the dynamic binary translator, providing as arguments the values in the registers of the virtual machine. The value in the program counter serves as offset d for Translation and Execution as Coroutines, above. □

Avoiding Translation. Dynamic binary translation increases the size of the trusted computing base (by adding the binary translator) and increases run-time overhead (since performing the translation takes time and likely involves making a context switch). The larger trusted computing base seems unavoidable. But we can reduce the run-time overhead by limiting how much of the code gets translated during execution and by not translating the same block of instructions anew every time that block is to be executed. We now consider implementation of these optimizations.

If the following condition holds for the instruction set then the input executable is equivalent to the output executable, so dynamic binary translation is not necessary in order to obtain a binary to execute in user mode.

> **Binary Translation Elimination Condition.** Execution of any non-virtualizable instruction while in user-mode advances the program counter but does not make any other changes to the state (memory or registers) of the virtual machine. □

This condition holds for many of the commercially-available processors that have non-virtualizable instructions. In addition, the preponderance of code running on computers is user-mode; only operating system code executes in system-mode. So when a VMM is implemented using dynamic binary translation on a processor where Binary Translation Elimination Condition holds, then only the operating system code in a virtual machine must incur the run-time overhead of dynamic binary translation.

We now show that when Binary Translation Elimination Condition holds, executing the input executable in user mode is equivalent to executing the output executable. The interesting cases are system-mode instructions, given that Implementing Virtual Machines by using Binary Translation does not replace user-mode instructions.

> *Case 1: A system-mode instruction ι that is non-virtualizable.* According to Binary Translation Elimination Condition, execution of ι on a processor in user-mode will advance the program counter but change no other aspect

of the processor's state. That behavior is equivalent to what would be observed if ι were replaced by a hypervisor call and the hypervisor call interrupt handler emulated the user-mode execution of ι. So execution of ι in the input executable already exhibits equivalent behavior to execution of the output executable.

Case 2: Other system-mode instructions. Such an instruction ι will cause a privilege interrupt when executed, because virtual machines are executed by an underlying processor in user-mode. So, an interrupt handler installed by the VMM receives control and executes a routine to emulate ι. This behavior is equivalent to what would be observed if ι were replaced by a hypervisor call, because the hypervisor call interrupt handler in Implementing Virtual Machines by using Binary Translation (above) emulates execution of ι.

A second means for reducing run-time overhead from binary translation is to introduce a VMM-maintained *translation cache*, which stores output executables for previously executed (and, therefore, previously translated) blocks of instructions, and it also stores the values of any registers that affected the translation.

Use of a Translation Cache. For a block of instructions that starts at offset d, a binary translator need not produce an output executable for execution, provided

(i) the required output executable O was previously produced and is stored in the translation cache, and

(ii) output executable O stored in the translation cache is what the binary translator would produce if invoked now. □

Provided (i) and (ii) are cheap to check, Use of a Translation Cache lowers overhead—executing a block from the translation cache does not require translating that block again. When Binary Translation Elimination Condition holds too, the translation cache would store only those parts of the virtual machine's operating system that execute in system-mode; the full performance benefit of a translation cache thus is achieved by incurring only modest storage costs.

To check condition (ii) in Use of a Translation Cache, we can leverage address translation hardware to intercept those writes that could cause cache entries to become stale.

Translation Cache Invalidation.

- *When an output executable O is inserted into the translation cache.* Disable writes for a region of memory that includes all fragments of the input executable that the translator read when producing O.
- *When a write is attempted to a region of memory where writes have been disabled by the translation cache.* Delete the corresponding output executable from the translation cache; then allow the write to proceed. □

Condition (ii) is satisfied if the output executable is in the translation cache and if current register values equal cached values for registers that affected the translation.

A performance problem arises with this scheme, because address-translation hardware typically works at the granularity of memory pages but far less than a page is used to produce an output executable for a single block of instructions. So writing to a page could cause many output executables to be deleted from the translation cache. Some of those deletions would be unwarranted if only a small part of the page is being updated or if state (but not instructions) is what changed. However, the unwarranted deletions can be avoided if the implementation of condition (ii) saves in each cache entry the translator's input and uses that value for later comparison with the contents of memory. This checking of the binary translator's input can be incorporated into the output binary.

10.3.3 Paravirtualization

Transfers of control between a virtual machine and the VMM disrupt instruction pipelining and require main-memory caches to be purged. So performance suffers when a VMM implements system-mode instructions by emulating them in software. Moreover, having the VMM replicate the hardware interface leads to further performance problems.

- The operating system in a virtual machine duplicates work performed by the VMM. For example, input/output from an application running in a virtual machine involves executing a driver in the VMM as well as executing a driver in the operating system.

- Work done in the VMM can negate work done in the operating system. Re-ordering of transfer requests that a VMM's disk driver does to enhance disk performance might undermine request re-ordering done by the operating system's driver to enhance disk performance.

Such performance problems suggest favoring an instruction set that does not often involve software-emulation by the VMM.

Virtual machines implemented using *paravirtualization* support the same user-mode instructions as the underlying processor, a subset of its system-mode instructions, and a hypervisor call. The subset of supported system-mode instructions typically excludes system-mode instructions that are expensive to emulate in software and also excludes all non-virtualizable instructions.[17] VMM-serviced hypervisor calls replace the system-mode instructions that are not in the subset of supported system-mode instructions.

Software built exclusively from user-mode instructions does not have to be changed to run in a virtual machine implemented by paravirtualization. So paravirtualization is transparent to application software. But operating system

[17] Recall, non-virtualizable instructions are, by definition, system mode.

routines invoke system-mode instructions; that code must be changed for execution under paravirtualization. In practice, those changes are typically localized to a handful of routines.

Leverage from Hypervisor Calls. Paravirtualization offers the flexibility to define virtual machines having hypervisor calls that do not replicate the functionality of system-mode instructions. Abstractions well suited to virtualization now can be offered by a VMM. For instance, an abstract input/output device could well be easier to emulate in software than a real device is. And paravirtualization would enable a VMM to offer that simpler input/output device, resulting in a VMM that is smaller than one that incorporates emulations for real input/ouptut devices; operating system drivers in virtual machines now can be simpler, too. An abstract input/output device's interface also can be designed to discourage operating system driver functionality that is duplicated or negated by a VMM's software emulation of the device.

In addition, if virtual machines employ hypervisor calls to interact with VMM-implemented resources then functionality can be relocated from a VMM into separate, designated virtual machines.

> **Privileged Virtual Machines.** A designated virtual machine V can implement a given service for the VMM (and thus for other virtual machines) provided the VMM offers the following.
>
> – The VMM identifies a specific subset of its hypervisor calls as providing a *control interface* for the service.
>
> – The VMM identifies the designated virtual machine V as being *privileged* for the service. V might be, for example, the first virtual machine that the VMM boots or a virtual machine that boots some specific operating system.
>
> – The VMM ensures that hypervisor calls in the control interface for a service can be invoked only by a virtual machine that is privileged for that service. □

Virtual machines would still use ordinary hypervisor calls for requesting services from the VMM or for retrieving corresponding responses. But instead of the VMM incorporating all of the code to perform that service, the VMM would forward the request to a privileged virtual machine; hypervisor calls in the corresponding control interface are what allows that virtual machine to communicate with the VMM and with client virtual machines. Ordinary virtual machines cannot interfere, because ordinary virtual machines cannot invoke hypervisor calls from a control interface and, therefore, they cannot receive or reply to service requests from clients.

This architecture expands the trusted computing base to include the operating system and other code that runs in a privileged virtual machine. All else equal, establishing assurance for this larger code base would be more costly. The architecture does offer some benefits, though. First, by moving functionality from the VMM into virtual machines, the VMM involves less code, providing

a basis for increased assurance in the VMM. Second, code executing in a virtual machine that has an operating system (with all of its functionality) can be simpler than code that, being within the VMM, cannot use operating system services. Finally, the architecture allows an existing operating system with existing I/O drivers to provide virtual machines with access to input/output devices. We run this existing operating system in a privileged virtual machine, and doing so avoids the need to write or rewrite input/output drivers for execution in the VMM. Software emulation to create virtualized versions of input/output devices is also now straightforward—virtualized devices can be implemented as servers, benefiting from existing input/output drivers and other functionality that an operating system offers.

10.4 Containers

Isolation is undermined whenever resources are being shared. For example, one process can interfere with others by abusing files, network ports, or locks. Even a shared processor could be problematic if one process is able to initiate activities that deprive others of processor cycles, causing missed deadlines. A *container* is an environment in which specified system resources are accessible only to a given set of processes. Processes outside a container cannot use the container's resources to influence processes within the container and *vice versa*. Thus, for processes within a container, isolation is enforced for resources that the operating system is providing in addition to isolation being enforced for memory and registers.

10.4.1 Implementation of Containers

A run-time environment for supporting containers typically is located[18] in or above the system layer that is implementing the resources and processes to be isolated by a container. The run-time environment would provide the full set of system operations, but it would intercept invocations that should be blocked or where arguments must be checked or modified before executing the container support software's implementation of these same operations. Overhead is typically reduced by locating container support software in the kernel, although then each container must include software libraries for functionality needed beyond what the kenel provides.[19]

Isolation by Namespace Mappings. A *namespace mapping* translates a name used within a container to the name used by the operating system for accessing that resource. Per-container isolation cannot be violated by accessing resources with names from a given namespace if

[18] See Figure 10.1 on page 325.

[19] A container that includes its own libraries can be run on any system that exports a given (standardized) kernel interface—additional software need not be installed. Moreover, after the container has been installed, processes running on that host but outside the container can continue using other libraries for the same functionality.

(i) the only way for a process to affect or be influenced by a resource requires naming that resource in a system operation that is being intercepted,

(ii) the same namespace mapping is used by all processes within a given container, and

(iii) namespace mappings used in different containers have disjoint ranges.

Namespace mappings are also useful in connection with system operations where the set of affected objects is implicit rather than being explicitly given as arguments. This is because the ranges of the namespace mappings identify those system objects that are accessible within a given container, so an implicit reference now can be adjusted to refer to the appropriate subset.

Hierarchical namespaces warrant special attention, both because they are common in computing systems and because they admit namespace mappings with simple implementations. In a hierarchical namespace, set $\Pi(t)$ of names is the set of paths starting at the root t of some tree. Per-container isolation follows if each container uses only resources with names from a disjoint subtree since, by definition, $\Pi(t) \cap \Pi(t') = \varnothing$ holds when t and t' are roots of disjoint subtrees. Moreover, any individual partition of a namespace (if not too small) can itself be partitioned. So a software layer can allocate some of its resources to the next higher layer and even partition those resources among functions that higher layer implements. As an example, a container C that is associated with some tree t_C of a hierarchical namespace could host a set of (sub-)containers by partitioning t_C into disjoint subtrees, one per (sub-)container.

Namespace Mappings for Containers. Each type of resource that an operating system supports will have a namespace for identifying resource instances of that type. Different resource types (e.g., files, network ports, locks, processes) typically have different namespaces.[20] A process uses names from these namespaces as arguments when making system calls; a single namespace for each resource type is typically shared by all processes in the system.

An obvious route to getting per-container isolation for system resources is to interpose namespace mappings that satisfy isolation conditions (ii) and (iii) above. For each resource type, the same namespace mapping would be associated with all processes within a given container but mappings associated with different containers would have disjoint ranges. So per-container isolation is facilitated by having a way to interpose per-process state that specifies a namespace mapping to use for each namespace the operating system defines. We would want the effects of such mappings, once in place, to be irreversible and to be inherited when a new process is spawned. That suggests a namespace mapping installed by a child should compose with the namespace mapping associated with its parent rather than replacing the parent's mapping.

[20] Some operating systems, however, include all resources in the namespace for the file system. For example, each process would be associated with an entry in a `proc` subdirectory, each lock by an entry in a `locks` subdirectory, etc.

A file system `chroot` (change root) operation will illustrate. Execution of `chroot`(π) redefines the root of the current file system to be the directory at path π. The current file system is now a sub-tree of what it was, so only a subset of files are still accessible to the process or to any processes it spawns (assuming a process inherits the file system root from its creator). The new restriction on access to files means that executing `chroot` can further restrict what files are accessible but cannot reverse the effect of a previous `chroot` and restore access to files. Filesystem isolation for containers then follows by associating with each container C a disjoint subtree π_C and having the process that creates C invoke `chroot`(π_C) prior to spawning processes to populate C.

Performance Isolation. For most system resources, a scheduler determines which process next gets access and for how long. The goal is to provide guarantees for the time, space, and/or bandwidth each process consumes of some resource that is being managed. Nothing about this architecture requires that consumption be attributed to processes—attribution could be to containers. When attribution is to containers, then performance guarantees would be for aggregated activity by processes within a container rather than for activity by an individual process.

Per-container performance isolation is just a set of stringent performance guarantees. Therefore, performance isolation can be realized through a choice of scheduling policies. Limits and entitlements can ensure resource availability; fair-share schedulers can implement guarantees for time-multiplexed resources. Enforce these policies, and it is no longer possible for an attacker to perpetrate a denial of service attack on processes within one container by compromising another container.

Moreover, modest system support suffices for enforcing per-container performance isolation:

- The system would associate with each process a label identifying the container (if any) that hosts this process.
- Use of a resource by a process would be attributed to the container that contains the process. Capacity allocated to a container would be available for use by any process in that container.
- The system would associate with each container a label that the system's scheduler uses for assigning capacity of each resource to processes within the container.

10.4.2 Comparing Containers with Virtual Machines

Containers are virtual machines. They virtualize the interface an operating system kernel provides to processes, thereby creating an environment where the processes in one container are isolated from the processes in other containers—even though all might share underlying processors, their memories, and an operating system kernel. Isolation also is the goal when virtualizing a processor's

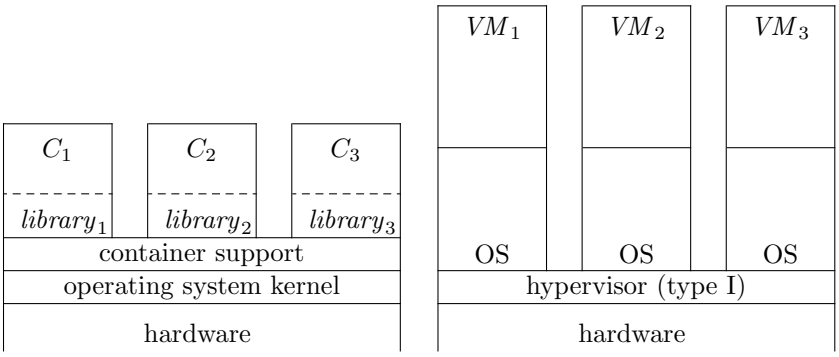

Figure 10.1: Architectures for Isolation

instruction set architecture. It is natural to wonder about the strengths and weakness that come with each type of virtual machine. When should we use one or the other?

Figure 10.1 compares the structure of a system that supports containers with a system that supports virtual machines. In that figure, the height of each layer was chosen to reflect common beliefs about that software:

- Virtual machines incorporate an entire operating system, so virtual machines are large. Containers only include application code and needed software libraries, so containers are much smaller.
- A type I hypervisor that does not use binary translation can be comparable in size to an operating system kernel, making that hypervisor small relative to the size of an operating system.
- Container support (i.e., namespace mappings, performance isolation, and management of other attributes) is comparable in size to an operating system kernel.

These common beliefs do not always hold, though. Large hypervisors are becoming more common in order to incorporate performance optimizations, employ binary translation, and/or include a full operating system to avoid duplicating input/output functionality or for type II hypervisors. Container support software is also starting to become bloated with features to extend the functionality of an underlying kernel.

Performance Comparison. Being part of a container should not slow execution of a process, since that process's instructions are directly executed, as are the instructions for any system software it invokes. In comparison, a virtual machine does not directly execute system-mode instructions, resulting in increased execution times for processes. Even processes that do not include system-mode

instructions are slowed when executed by a virtual machine if those processes invoke operating system services, because that operating system code runs slower when it execute system-mode instructions.

Because process creation involves system-mode instructions, the overhead of intercept and emulate or binary translation for virtual machines also means that starting a new process within a container is likely to be considerably faster than starting a new process within a virtual machine. The impact is not significant if new processes are started infrequently. But a common design for a network service is to start a new process for each request that is received. That design performs considerably better in a container than in a virtual machine. For deployment in a virtual machine, the better design is to have a fixed pool of server processes. Each server process loops, with each iteration removing a request from a shared queue, parsing the request, and delivering the indicated service.

To start a new virtual machine requires booting its operating system, which typically is time-consuming. Therefore, starting a new virtual machine cannot be frequent and cannot be on the critical path for generating interactive responses. One wouldn't start a new virtual machine to service each request, for example, even though that architecture brings strong isolation. In contrast, starting a new container involves only a small delay—time is required only to set up some tables (e.g., for namespace mappings) and to add entries or change other tables in the kernel (e.g., to record parameters for performance isolation and other things).

Isolation Comparison. The environment of a process includes abstractions that are provided by the operating system kernel. Some of these abstractions (e.g., files, locks, network ports, other processes) are accessed by making system calls; others (e.g., the processor) are accessed implicitly. Either way, by accessing these abstractions, one process might influence the environment of another process. By comparison, a virtual machine has only one way to influence its environment—input/output operations. So we conclude that processes per se provide weaker isolation guarantees than virtual machines.

Containers change the picture. The goal is to have the environment for a process executing in a container be affected only by other processes within that container, so processes executing within containers benefit from a stronger isolation guarantee than processes that are not within containers. That isolation guarantee, however, is weaker than what virtual machines provide. First, even if a container were to mediate access to all operating system resources, containers do not isolate one process in the container from other processes within that container. Second, container support software usually does not mediate access to all system resources, so avenues will typically remain for influence from outside a container.

Another basis for comparing isolation guarantees is to consider assurance. All else equal, the complexity of a software component is correlated with its size. Since more complex artifacts are thought to be easier to attack, component size

can be used to predict assurance about isolation guarantees. We first consider such an analysis for virtual machines.

- A type I hypervisor that does not use binary translation is considerably smaller than an operating system. So its size would predict isolation guarantees that a type I hypervisor enforces for virtual machines should be harder to compromise than the guarantees that an operating system enforces for processes executing within the virtual machine.

- A process might elevate its privilege or corrupt state by exploiting a bug in an operation that an operating system implements. This justifies predicting lower assurance for the isolation an operating system provides within a virtual machine than for the isolation that a hypervisor provides between virtual machines.

By running an operating system in a virtual machine, then, we would predict that an attacker executing in one virtual machine is more likely able to influence other activity on that virtual machine than able to influence activity in another virtual machine.

An assurance-based analysis for isolation with containers is subtle. The size of an operating system kernel with added container support software is comparable to the size of a type I hypervisor that does not use binary translation. But system operations accessible within a container are an avenue for perpetrating attacks that might not only influence other execution in the container but execution outside the container, too (though not extending beyond a virtual machine). That suggests there would be a weaker isolation guarantee for containers than for virtual machines. However, a type II hypervisor includes a full operating system (and thus considerably more code than required for running containers), justifying a lower degree of assurance in the isolation being provided for virtual machines; use of binary translation has the same effect.

Notes and Reading

Processes. Early computers executed one job at a time; enforcing isolation was not a concern. As technology improved, processors got faster but input/output devices didn't. A desire to maximize processor utilization then led to the advent of multiprogramming [48] and interrupts [5], whereby a processor executed instructions in parallel with (still, relatively slow) input/output operations. By having multiple jobs co-resident in main memory, if one job had to wait for an input/output operation to complete then the processor could execute another job. Isolation now was needed, however, to prevent a bug in one job from corrupting memory occupied by another [9]. So processors included a base and a limit register, and system software used these to associate a disjoint memory region with each job.

Batch processing is not conducive to program debugging; timesharing [32, 57]

is. Compatible Time Sharing System (CTSS) [11, 60] led the way[21] in the implementation of this form of multiprogramming, which gives each user a terminal and the illusion of exclusive access to a computer. In CTSS, timer interrupts enabled system software to time multiplex the processor's registers and main memory. A user had exclusive access to this *session state* for a *time slice*, in rotation with other users. Between time slices, the user's session state would be stored on an external magnetic drum. Input/output operations for swapping session states to/from the drum could run in parallel with the processor delivering time slices, because session states for multiple users occupied main memory concurrently. Base and bounds registers prevented execution by one user from corrupting memory being used by another. The base register also facilitated relocation, resulting in better memory utilization since holes now could be avoided when a session state was swapped into main memory.

The success of CTSS and promise of timesharing justified ARPA funding MIT's project[22] MAC (Multiple Access Computing) to explore the design and construction of Multics (Multiplexed information and computing system), a timesharing system that could serve as a public utility [10]. Because users of a public utility might not necessarily trust each other, system security was now a first-class concern. The designers of Multics proposed using a segmented (and paged) virtual memory [14] to enforce isolation for main memory. General Electric's GE-645 processor developed to host Multics was a modified version of the GE-635, which was similar to the IBM 7094 that had been running CTSS. Segment descriptors on the GE-645 were derived from the Burroughs B5000 [8], but with support added for hardware rings of protection [20, 56, 54]. Multics not only became a commercial product,[23] but the mechanisms and principles it contributed to the field of computer security are still having an impact [51]. The Multicians web site [37] gives a history of Project MAC and Multics, along with links to publications.

Operating systems grew larger and more complex, as patches, features, and performance optimizations were added. The larger system invariably would have more bugs and, therefore, be easier to attack. So Roger Schell proposed [65, 52, 22] an operating system architecture where security would depend only on the operating system kernel. This "security kernel" would only implement processes and a reference monitor. Because a security kernel would be small, assurance could be established by using formal methods. And because all accesses by processes were mediated in the security kernel, bugs elsewhere in the operating could not be exploited to compromise system security.

A number of implementation efforts were undertaken to validate the architecture Schell had been advocating. Most enforced security models based on

[21] CTSS first became operational in 1961 on a modified IBM 709. By 1964, a version was in production use at MIT, running on an IBM 7094 that included an interval timer and two 32K banks of memory. The label "compatible" was included in the name because the system also handled batch jobs using IBM's FMS (FORTRAN Monitor System), a widely used operating system for the IBM 709 and IBM 7094.

[22] Bell Telephone Laboratories was a collaborator from 1965 to 1969.

[23] The number of commercial installation peaked at about 80 by the early 1980's, with the last site shut down on October 30, 2000.

DoD's system of security classifications and user clearances—DoD was funding the efforts—and many at least started to construct some sort of formal arguments to establish correctness of their designs: a prototype built at MITRE for the DEC PDP-11/45 [53, 36], the Ford Aerospace KSOS (Kernelized Secure Operating System) secure UNIX system [33], Honeywell's STOP operating system for Scomp (secure communications processor) [17], a redesign of Multics [63, 55], and UCLA Data Secure Unix [61].

By locating access mediation within the kernel, a system was limited to enforcing a single security policy. That restriction turned out to be problematic for real deployments. First, the multilevel security policy being enforced by existing security kernels required having certain so-called "trusted processes" that would be exceptions to the policy. Second, different security policies made sense for different applications. These problems led Rushby [50] to invoke "less is more" and suggest separation kernels as an alternative to security kernels.

Virtual Machines. Virtual machines were developed at IBM—albeit, surreptitiously [58, 59]. Atlas [27], a computer built at the University of Manchester, had introduced the idea of demand paging, which created a larger virtual memory by time-multiplexing pages of physical memory. Demand paging freed programmers from the headaches of implementing storage management. It also facilitated timesharing, because the processor would automatically load and relocate session state, incrementally and as needed. To explore virtual memory, IBM researchers built the IBM M44 [38] by modifying an IBM 7044 so it supported dynamic address relocation; an associated operating system MOS (Modular Operating System) [39] implemented timesharing by providing M44x "virtual machines" (the first use of this term) that each was connected to a terminal and resembled an IBM 7044.

IBM interest and energy was focused elsewhere, though. The company was creating System/360, a family of processors that each implemented the same instruction set architecture but delivered different levels of performance. Previously, IBM had marketed one line of processors to commercial users and an incompatible one to scientific users. The System/360 family would serve all of IBM's markets. Support for timesharing, however, was not seen by IBM as important, and System/360 processors did not support virtual memory. So when IBM bid on providing the hardware for Multics, no System/360 processor was suitable. IBM had to propose building hardware for address translation to augment a System/360 processor. Project MAC's management rejected that, fearing non-standard hardware would discourage other sites from running Multics.

IBM had established the Cambridge Scientific Center[24] to foster relations with MIT and academia. Loss of the Multics bid and IBM's indifference to timesharing undermined that mission. However, Norm Rasmusssen, founding director of the Cambridge Scientific Center, did understand the importance of time-sharing, so he launched an effort there to build a time-sharing system. The

[24]The center was housed in the same building (575 Technology Square) as Project MAC.

system would run on a System/360 model 40 that had been augmented with custom hardware [28] for address translation. Operational in January 1967, the system comprised CP-40 [1], which implemented virtual System/360's, and the Cambridge Monitor System (CMS), a new single-user System/360 operating system that supported timesharing for one user.[25] This software was subsequently rewritten to run on a System/360 Model 67[26] and became available to IBM customers as CP-67/CMS [35]. A port to System 370 processors produced VM/370 [13]; a port of VM/370 to IBM's Z computers later became available as z/VM.

Once virtual machines were shown to be viable for timesharing, security researchers investigated trade-offs with using a hypervisor to enforce isolation [31] and to serve as a security kernel [46].[27] To evaluate having a hypervisor be a security kernel, UCLA built the UCLA-VM system [43] for a DEC PDP-11/45, and System Development Corporation built the KVM/370 [18] retrofit to IBM's VM/370. Further evidence that hypervisors should be trusted came from an IBM penetration study [2] of VM/370, where only a few dozen vulnerabilities were discovered, and most were connected to the idiosyncratic System 360 input/output architecture. However, the most compelling case for using a hypervisor to enforce security is the DEC VAX Security Kernel [24, 29], which met all DoD requirements for the highest levels of assurance, demonstrated tolerable levels of performance (i.e., factor of 2 degradation), and could run DEC's VMS and ULTRIX-32 operating systems on commercial hardware (albeit with microcode modifications to enable virtualization of the VAX architecture).

Not all instruction set architectures can be virtualized by emulating a small subset of the instructions and running the rest directly on the underlying processor. Goldberg's Ph.D. dissertation [19] discusses what makes instructions problematic for such so-called "trap and emulate" implementations on third generation processors—processors having two-modes of operation and base/bounds registers to relocate addresses. Terms type I, type II, self-virtualizing, and recursive virtual machines were also introduced in that dissertation. Subsequently, Goldberg and Popek [41] formalized conditions and proved that satisfying them suffices for implementing virtual machines by using trap and emulate on third-generation processors.

Outside of IBM's offerings, third-generation processors typically have not satisfied the Goldberg-Popek conditions. A VMM for one of those computers cannot just employ trap and emulate; it must use other approaches. One such approach is to change the instruction set architecture. Popek and Kline [42]

[25] Most time sharing systems are multi-user, which requires them to incorporate mechanisms for sharing resources. As a single-user system, CMS avoided that complexity and, thus, it represented a novel point in the design space.

[26] Announced August 1965, System/360 Model 67 brought address translation to the System/360 family. It and TSS (an ambitious multi-user timesharing system) were IBM's response to losing the Multics bid and realizing that timesharing would become more than a niche market.

[27] For enforcing multilevel security, a fixed security label is associated with each user, with virtual machine, and with virtual input/output device. A reference monitor incorporated into the hypervisor then enforces the usual access restrictions according to these labels.

report on commissioning such changes to a DEC PDP 11/45. Producers of processors do have both the incentive and the means to introduce modifications if those changes could bring significant increases to sales. That motivated DEC to undertake the microcode modifications to the VAX for supporting the DEC VAX Security Kernel. It also led Intel to develop the VT-x extensions (available November 2005) and AMD to develop the AMD-V extensions (available May 2006) to help with virtualization of the x86 architecture[28] because servers deployed in a cloud often run a VMM and the market for those servers is large.

There are software alternatives to trap and emulate for implementing a VMM when an instruction set architecture does not satisfy the Goldberg-Popek conditions. VMWare's virtualization [7, 15] for the x86 was the first to employ dynamic binary translation [16] for unvirtualizable instructions. The VMWare developers leveraged their experience [49] with using the Embra [64] binary translator.

The term paravirtualization was coined[29] for describing Denali [62], a VMM to support an x86 variant having a simplified virtual memory and interrupt architecture. That variant was devised intending to facilitate hosting large numbers of virtual machines running unmodified x86 applications as network services in a cloud. Xen [3], developed around the same time, also used paravirtualization. The goal for Xen was hosting commodity x86 operating systems Linux and Windows (albeit with some modifications to the code) that ran commodity applications (with no modifications to that code). Xen subsequently added support for full x86 virtualizations by leveraging VT-x and AMD-V.

Examples of paravirtualization, however, predate Denali and Xen. Disco [6] had implemented virtual machines for MIPS but changing the interrupt flag from being stored in a processor register to being stored in a special memory location. Long before that, however, IBM's CP-67 and VM/370 had repurposed the System/360's `diagnose` instruction to provide direct communication between a virtual machine's operating system and the VMM, thereby enabling the operating system to avoid duplicating activities that the VMM would perform. And the DEC VAX Security Kernel had used paravirtualization to avoid having to virtualize device I/O.

Containers. Containers bring together various security mechanisms that were developed to facilitate using Unix as a host for web servers and other network applications.[30] What follows is an abbreviated history; a detailed account is given by Randal [45]; that paper also gives the history of virtual machines and compares containers with virtual machines.

[28] Robin and Irvine [47] enumerate ways x86 does not satisfy the Popek-Goldberg conditions [41].

[29] Steve Gribble, the faculty member who directed the Denali effort, credits graduate student Andrew Whitaker with coining the term "paravirtualization" [21].

[30] Unlike the other isolation abstractions discussed in this chapter, containers are noteworthy for also serving as a widely used software distribution vehicle. Prominent examples of commercial technologies for development and deployment of containers include Docker (derived from Linux containers) and Kubernetes (developed at Google).

Bell Labs researchers had added the `chroot` command to Unix in 1979 [26]. A decade later, the Bell Labs Plan 9 [40] operating system featured a single hierarchical namespace that not only included directories and files but contained names for other system resources, too. However, stronger isolation and support for delegating administration would be needed for a system to host independent network servers and applications.

These requirements motivated the development of *jails* [23] in FreeBSD Unix version 4.0 (released in 2000). Each jail provided a disjoint set of processes with exclusive access to a sub-tree of the file system and to an IP address. Processes in a jail could use both unprivileged system operations and privileged ("root" in Unix terminology) system operations—but these operations could access only those system resources allocated to the jail. So processes in a jail could interact with each other, could use and administer resources allocated to the jail, but had no means to interact with or even ascertain the existence of other processes or other system resources.

The *zones* [44] construct was introduced in 2004 to support server consolidation under Solaris 10 (another Unix successor). FreeBSD jails lacked two capabilities needed for hosting such workloads. First, performance isolation is important when servers share a computing system and, therefore, zones (unlike jails) supported per-zone entitlements, limits, and partitions of certain system resources into resource pools, as well as fair-share CPU scheduling [25] for allocation of CPU capacity across (and within) different zones. Second, each zone (unlike a jail) had its own namespace mappings for certain system resources (e.g., semaphores and message queues used for communications and synchronization, and IP addresses). Given these mappings, configuration changes were not needed to avoid resource-name conflicts when a server running on its own computer was moved to a computer that was hosting multiple servers (each in a separate zone).

Developed by the LXC (LinuX Containers) project [30] and available starting in 2008, the underlying mechanisms for Linux containers [34] are a generalization of jails and zones. Biederman [4] had done an analysis and identified the namespaces for all system resources exported by the Linux kernel; a Linux container would have a separate copy of each namespace. Menage and Seth, working at Google, generalized Linux `cpusets`, and obtained *cgroups* as a mechanism for defining parameters to control each subsystem and for associating these parameters with a group of processes and their progeny [12]; Linux containers incorporated cgroups to support having per-container associated attributes. Some evolution was still to come—notably integration of authorization to restrict system operations—but the biggest changes would concern support for assembling the contents of containers, as exemplified by the Docker ecosystem.

Bibliography

[1] R.J. Adair, R.U. Bayles, L.W. Comeau, and R.J. Creasy. A virtual machine system for the 360/40. Technical Report 320-2007, Cambridge Scientific

Center, May 1966.

[2] C. R. Attanasio, P. W. Markstein, and R. J. Phillips. Penetrating an operating system: A study of VM/370 integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[3] 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*, SOSP '03, pages 164–177, New York, NY, USA, Ocotber 2003. Association for Computing Machinery.

[4] Eric W. Biederman. Multiple instances of the global Linux namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112, July 2006.

[5] F. P. Brooks. A program-controlled program interruption system. In *Papers and Discussions Presented at the December 9-13, 1957, Eastern Joint Computer Conference: Computers with Deadlines to Meet*, IRE-ACM-AIEE '57 (Eastern), pages 128–132. Association for Computing Machinery, 1957.

[6] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 143–156, New York, NY, USA, 1997. Association for Computing Machinery.

[7] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Transactions on Computer Systems*, 30(4), November 2012.

[8] Burroughs Corporation. *The Descriptor—A Definition of the B5000 Information Processing System*, 1961. Michigan.

[9] E. F. Codd, E. S. Lowry, E. McDonough, and C. A. Scalzi. Multiprogramming STRETCH: Feasibility considerations. *Communications of the ACM*, 2(11):13–17, November 1959.

[10] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *Proceedings of the 1965 Fall Joint Computer Conference, Part I*, AFIPS Conference Proceedings, pages 185–196, New York, NY, USA, November 1965. Association for Computing Machinery.

[11] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An experimental time-sharing system. In *Proceedings of the 1962 Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), pages 335–344, New York, NY, USA, May 1962. Association for Computing Machinery.

[12] Jonathan Corbet. Notes from a container, October 2007. https://lwn.net/Articles/256389/.

[13] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[14] Jack B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM*, 12(4):589–602, October 1965.

[15] Scott W. Devine, Edouard Bugnion, and Mendal Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. U.S. Patent 6,397,242 B1. Filed October 26, 1998, issued May 28, 2002.

[16] Kemal Ebcioğlu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 26–37, New York, NY, USA, 1997. Association for Computing Machinery.

[17] L. J. Fraim. Scomp: A solution to the multilevel security problem. *IEEE Computer*, 16(7):26–34, 1983.

[18] B. D. Gold, R. R. Linde, R. J. Peeler, M. Schaefer, J. F. Scheid, and P. D. Ward. A security retrofit of VM/370. In *Proceedings of the National Computer Conference*, NCC, pages 335–344. IEEE, 1979.

[19] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, February 1973.

[20] Robert M. Graham. Protection in an information processing utility. *Communication of the ACM*, 11(5):365–369, May 1968.

[21] Steve Gribble. Personal communication.

[22] Stanley R. Ames Jr., Morrie Gasser, and Roger R. Schell. Security kernel design and implementation: An introduction. *IEEE Computer*, 16(7):14–22, 1983.

[23] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, May 2000.

[24] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 2–19. IEEE Computer Society, May 1990.

[25] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[26] B. W. Kernighan and M. D. McIlroy. *UNIX™ Time-sharing system: UNIX Programmer's Manual*. Bell Telephone Laboratories, January 1979. Murray Hill, New Jersey.

[27] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, 1962.

[28] A. B. Lindquist, R. R. Seeber, and L. W. Comeau. A time-sharing system using an associative memory. *Proceedings of the IEEE*, 54(12):1774–1779, 1966.

[29] Steve Lipner, Trent Jaeger, and Mary Ellen Zurko. Lessons from VAX/SVS for high-assurance VM systems. *IEEE Security and Privacy*, 10(6):26–35, 2012.

[30] LXC. Infrastructure for container projects. Sponsored by Canonical Ltd. https://linuxcontainers.org.

[31] Stuart E. Madnick and John J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 210–224, New York, NY, USA, 1973. Association for Computing Machinery.

[32] John McCarthy. A time-sharing operator program for our projected IBM 709. Memoradum to Philip M. Morse, Cambridge, MA, 1959.

[33] E. J. McCauley and P. J. Drongowski. KSOS—The design of a secure operating system. In *Proceedings of the National Computer Conference*, NCC, pages 345–353. IEEE, 1979.

[34] Paul B. Menage. Adding generic process containers to the Linux kernel. In *Proceedings of the Linux Symposium*, volume 2, pages 45–57, July 2007.

[35] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, September 1970.

[36] Jonathan K. Millen. Security kernel validation in practice. *Communications of the ACM*, 19(5):243–250, May 1976.

[37] Multicians. https://multicians.org/index.html.

[38] R. A. Nelson. Mapping devices and the M44 data processing systsem. Technical Report RC 1303, IBM Research Division, October 1964.

[39] R. W. O'Neill. Experience using a time-shared multi-programming system with dynamic address relocation hardware. In *Proceedings of the April 18-20 1967, Spring Joint Computer Conference*, AFIPS 1967 (Spring), pages 611–621, New York, NY, USA, 1967. Association for Computing Machinery.

[40] Rob Pike, David L. Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(2):221–254, 1995.

[41] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[42] Gerald J. Popek and Charles S. Kline. The PDP-11 virtual machine architecture: A case study. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 97–105, New York, NY, USA, 1975. Association for Computing Machinery.

[43] Gerald J. Popek and Charles S. Kline. A verifiable protection system. In *Proceedings of the International Conference on Reliable Software*, pages 294–304, New York, NY, USA, 1975. Association for Computing Machinery.

[44] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, pages 241–254, USA, 2004. USENIX Association.

[45] Allison Randal. The ideal versus the real: Revisiting the history of virtual machines and containers. *ACM Computing Surveys*, 53(1), February 2020. Article 5.

[46] R. Rhode. Secure multilevel virtual computer systems. Technical Report ESD–TR 74 370, MITRE Corporation, Bedford, MA, February 1975.

[47] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, SSYM'00, USA, 2000. USENIX Association.

[48] Nathaniel Rochester. The computer and its peripheral equipment. In *Papers and Discussions Presented at the the November 7-9, 1955, Eastern Joint AIEE-IRE Computer Conference: Computers in Business and Industrial Systems*, AIEE-IRE '55 (Eastern), pages 64–69, New York, NY, USA, 1955. Association for Computing Machinery.

[49] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel Distributed Technology: Systems Applications*, 3(4):34–43, 1995.

[50] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. Association for Computing Machinery.

[51] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[52] Roger R. Schell, Peter J. Downey, and Gerald J. Popek. Preliminary design of secure military systems. Technical Report MCI–73–1, Air Force Systems Command, Directorate of Information Systems Technology, January 1973.

[53] W. L. Schiller. The design and specification of a security kernel for the PDP-11/45. Technical Report MTR-2934, MITRE Corporation, Bedford, MA, March 1975.

[54] M. D. Schroeder. *Cooperation of mutually suspicious subsystems in a computer utility*. PhD thesis, M.I.T. Department of Electrical Engineering, September 1972. Also available as M.I.T. Project MAC Technical Report TR-104.

[55] Michael D. Schroeder. Engineering a security kernel for Multics. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 25–32, New York, NY, USA, 1975. Association for Computing Machinery.

[56] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, March 1972.

[57] C. Strachey. Time-sharing in large fast computers. In *Proceedings of International Conference on Information Processing*, pages 336–341. UNESCO, June 1959.

[58] Tom van Vleck. The IBM 360/67 and CP/CMS. https://multicians.org/index.html.

[59] Melinda Varian. VM and the VM community: Past, present and future. In *SHARE 89 Technical Conference Proceedings, SHARING Worlds Of Knowledge*. SHARE Association, August 1997. Session 9059-9061.

[60] David Walden and Tom Van Vleck, editors. *The Compatible Time Sharing System (1961 – 1973) Fiftieth Aniversary Commemorative Overview*. IEEE Computer Society, June 2011. https://multicians.org/thvv/compatible-time-sharing-system.pdf.

[61] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, February 1980.

[62] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, 2002.

[63] J. Whitmore, A. Bensoussan, P. Green, D. Hunt, A. Kobziar, and J. Stern. Design for Multics security enhancements. Technical Report ESD–TR–74–176, Honeywell Information Systems, Cambridge, MA, December 1973.

[64] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, page 68–79, New York, NY, USA, 1996. Association for Computing Machinery.

[65] Jeffrey R. Yost. An interview with Roger R. Schell, Ph.D. Charles Babbage Institute, Center for the History of Information Teachnology, University of Minnesota, Minneapolis, May 2012.

