CHAPTER 6

Security Kernels

While the Multics project was winding down in the mid-1970s, a number of vendors and researchers gained confidence that a secure operating system could be constructed and that there was a market for such an operating system, within the US government anyway. Many of the leaders of these operating system projects were former members of the Multics team, but they now led other research groups or development groups. Even Honeywell, the owner of the Multics system, was looking for other ways to leverage the knowledge that it gained through the Multics experience.

While the Multics security mechanisms far exceeded those of the commercial operating systems of the day, it had become a complex system and some of the decisions that went into its design needed to be revisited. Multics was designed to be a general-purpose operating system that enforced security goals, but it was becoming increasingly clear that balancing generality, security, and performance was a very difficult challenge, particularly given the performance of hardware in the mid-70s. As a result, two directions emerged, one that focused on generality and performance with limited security mechanisms (e.g., UNIX) and another that emphasized verifiable security with reasonable performance for limited application suites (i.e., the security kernel). In the former case, popular, but insecure, systems (see Chapter 4) were built and a variety of efforts have been subsequently made to retrofit a secure infrastructure for such systems (see Chapters 7 through 9). In this chapter we examine the latter approach.

In the late 1970s and early 1980s, there were several projects that aimed to build a secure operating system from scratch, addressing security limitations of the Multics system. These included the Secure Communications Processor (Scomp) [99] from Honeywell, the Gemini Secure Operating System (GSOS or GEMSOS) [290] from Gemini, the Secure Ada Target (SAT) [34, 125, 124] and subsequent LOCK systems [293, 273, 274, 292, 276] from Honeywell and Secure Computing, respectively, which are based on the Provably Secure Operating System (PSOS) design [92, 226], the Kernelized Secure Operating System (KSOS) [198] from Ford Aerospace and Communications, the Boeing Secure LAN [298], and several custom guard systems (mostly proprietary, unpublished systems). In this chapter, we examine two of these systems, Scomp and GEMSOS, to demonstrate the design and implementation decisions behind the development of security kernels. These two systems represent two different implementation platforms for building a security kernel: Scomp uses custom hardware designed for security enforcement, whereas GEMSOS was limited to existing, commercially-popular hardware (i.e., the Intel x86 platform). These systems show what can be done when even the hardware is optimized for security (Scomp) and the limitations imposed on the design when available hardware is used (GEMSOS). Recent advances in commercial hardware, such as I/O MMUs, may enable us to revisit some Scomp design decisions.

6.1 THE SECURITY KERNEL

The major technical insight that emerged at this time was that a secure operating system needed a small, verifiably correct foundation upon which the security of the system can be derived. This foundation was called a *security kernel* [108]. A security kernel is defined as the hardware and software necessary to realize the reference monitor abstraction [10]. A security kernel design includes hardware mechanisms leveraged by a minimal, software trusted computing base (TCB) to achieve the reference monitor concept guarantees of tamperproofing, complete mediation, and verifiability (see Definition 2.6).

The first security kernel was prototyped by MITRE in 1974. It directly managed the system's physical resources with less than 20 subroutines in less than 1000 source lines of code. In addition to identifying *what* is necessary to build a security kernel that implements a reference monitor, this experience and the Multics experience indicated *how* a security kernel should be built. While mediation and tamperproofing are fundamental to the design of a security kernel, in building a security kernel the focus became verification. Three core principles emerged [10]. First, a security kernel has to *implement a specific security policy*, as it can only be verified as being secure with respect to some specific security goals. A security goal policy (e.g., based on information flow, see Chapter 5) must be defined in a mandatory protection system (see Definition 2.4) to enable verification. Second, the design of the security kernel must define a verifiable *protection behavior of the system as a whole*. That is, the system mechanisms must be comprehensively assessed to verify that they implement the desired security goals. This must be in the context of the security kernel's specified security policy. Third, the implementation of the kernel must be shown to be *faithful to the security model's design*. While a mathematical formalism may describe the design of the security kernel and enable its formal verification, the implementation of the security kernel in source code must not invalidate the principles established in the design.

Thus, the design and implementation of security kernels focused on the design of hardware, a minimal kernel, and supporting trusted services that could be verified to implement a specific security policy, multilevel security. While Multics had been designed to implement security on a particular hardware platform, the design of security kernels included the design of hardware that would enable efficient mediation of all accesses. The design of security kernel operating systems leverages this hardware to provide a small number of mechanism necessary to enforce multilevel security. Finally, some trusted services are identified, such as file systems and process management, that are necessary to build a functional system.

The primary goal of most security kernel efforts became *verification* that the source level implementation satisfies the reference monitor concept. This motivated the exploration of formal and semi-formal methods for verifying that a design implemented the intended security goals and for verifying that a resultant source code implementation satisfied the verified design. As Turing showed that no general algorithm can show that any program satisfies a particular property (e.g., halts or is secure), such security verification must be customized to the individual systems and designs. The work in security kernel verification motivated the subsequent methodologies for system security

assurance (see Chapter 12. The optimistic hope that formal tools would be developed that could automatically support formal assurance has not been fulfilled, but nonetheless assurance is still the most practical means known to ensure that a system implements a security goal.

Verification that an implementation correctly enforces a system's security goals goes far beyond verifying the authorization mechanisms are implemented correctly. The system implementation must be verified to ensure that all system resource mechanisms (see Chapter 1) are not vulnerable to attack. As computing hardware is complex, assurance of correct use of hardware for implementing system resources is nontrivial. Consider the memory system. A hardware component called the Translation Lookaside Buffer (TLB) holds a cache of mappings from virtual memory pages to their physical memory counterparts. If an attacker can modify these mappings they may be able to compromise the secrecy and integrity of system and user application data. Depending on the system architecture, TLBs may be filled by hardware or software. For hardware-filled TLBs, the system implementation must ensure that the page table entries used to fill the TLB cannot be modified by attackers. For software-filled TLBs, the refill code and data used by the code must be isolated from any attacker behavior. Further, other attacks may be possible if an attacker can gain access to secret memory after it is released. For example, heap allocation mechanisms must be verified to ensure clearing of all secret memory (e.g., to prevent *object reuse*). Even across reboots, secret data may be leaked as BIOS systems are inconsistent about whether they clear memory on boot or not, and data remains in memory for sometime after shutdown. As a result of these and other possible attack vectors (e.g., covert channels, see Chapter 5), careful verification of system implementations is necessary to ensure reference monitor guarantees, but it is a complex task.

In this chapter, we examine two of systems whose designs aimed for the most comprehensively assured security, Honeywell's Scomp [99] and Gemini's GEMSOS [290]. Both these systems achieved the highest assurance rating ever achieved for an operating system, A1 as defined by the Orange Book [304] assurance methodology [1]. Scomp was used as the basis for the design of the assurance criteria. GEMSOS is still available today [5].

6.2 SECURE COMMUNICATIONS PROCESSOR

The Honeywell Secure Communications Processor (Scomp) system is a security kernel-based system [99] designed to implement the Multics's multilevel security (MLS) requirements [23], see Chapter 3. The original idea was to build a security kernel and an emulator to enable execution of an ordinary operating system (UNIX), as was done by KSOS [198] and the UCLA Secure Data UNIX system [248]. After the performance and security of such emulated systems was found to be insufficient, a decision was made to build a new application interface for Scomp that provides applications with the necessary security that runs with reasonable performance.

The performance of a emulated system run on a security kernel is impacted by two issues. First, the emulation may involve converting between incompatible representations of the two systems. For example, UNIX I/O copies data directly to the application's address space (e.g., on a file or

[1]The GEMSOS A1 evaluation was as part of the BLACKER system.

network read), but Scomp maintains data in individually managed segments to which access must be authorized. As a result, rather than getting a filled buffer as in UNIX, Scomp I/O provides a reference to a segment with the data. Second, the hardware features of a system may not supply efficient primitives for emulated function. For example, Multics hardware did not provide hardware support for ring crossings (i.e., protection domain transitions), so these must be implemented in software at a higher cost.

Further, the emulated system may include mechanisms that are not secure with respect the requirements of the security kernel. For example, UNIX supports the transfer of file descriptors on `fork` and `exec` operations. Thus, a parent process may be able to leak data to the child or provide the means for the child to leak its own data. This problem must be addressed in secure versions of commercial operating systems, see Chapter 7, but the Scomp designers felt these and similar issues warranted a new application interface. The conflict between functional interfaces and how to secure them is fundamental to the design of secure systems.

As a result of these performance and security concerns, the Scomp designers developed not only a security kernel, but also new hardware mechanisms and a new application interface for writing programs to the security kernel. Below, we discuss the overall architecture, major features, and impact on application development.

6.2.1 SCOMP ARCHITECTURE

The Scomp system architecture is shown in Figure 6.1. The *Scomp trusted computing* base consists of three components running in rings 0, 1, and 2. The Scomp Trusted Operating System (STOP) consists of a security kernel running in ring 0 and trusted software running in ring 1. The trusted functions of the Scomp Kernel Interface Package (SKIP) run in ring 2. Applications access protected resources managed by the Scomp trusted computing base using the SKIP library running in the application's address space.

The Scomp security kernel mediates access to all protected resources using an MLS policy. When an application process needs access to a protected resource (i.e., a memory or I/O segment), it must ask the security kernel for a *hardware descriptor* sufficient to access this resource. The security kernel authorizes whether the process can access the resource, and if authorized, builds a hardware descriptor for accessing this resource. The security kernel stores the hardware descriptor and returns a reference to the descriptor to the process for subsequent use. A Scomp hardware descriptor includes an object reference and the authorized access permissions for that process.

Isolation, and hence tamperproofing, is implemented by a ring protection mechanism. Similar to Multics, an access bracket mechanism controls whether code in one ring is permitted to request services from another ring. Unlike Multics, all rings and ring transitions are implemented in hardware.

Complete mediation is implemented in hardware. All requests for memory or device access are mediated by security protection hardware described below in Section 6.2.2.

For the first time in an operating system development process, verification that the Scomp security model and implementation enforce the MLS policy was a first-class task. Scomp's trusted

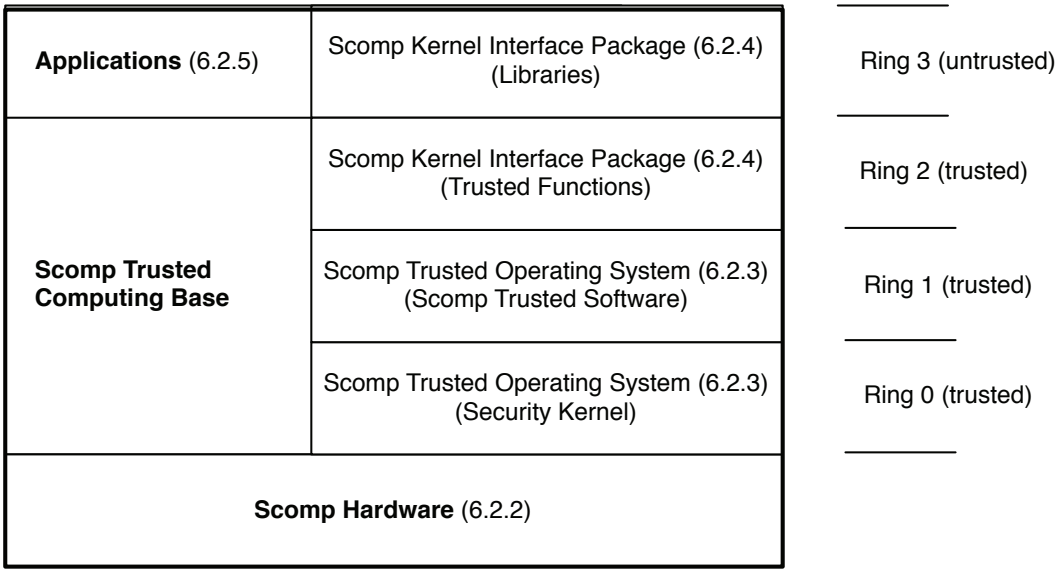

Figure 6.1: The Scomp system architecture consists of hardware security mechanisms, the Scomp Trusted Operating System (STOP), and the Scomp Kernel Interface Package (SKIP). The Scomp trusted computing base consists of code in rings 0 to 2, so the SKIP libraries are not trusted.

software is verified using two technologies. First, SRI's Hierarchical Development Methodology [48] is used to verify that a formal model of the security kernel's specification, called the *formal top-level specification* (FLTS), enforces the MLS policy. Second, trusted software outside the kernel is verified using a procedural specification applied using the Gypsy methodology [118].

Based on the hardware enforcement features (complete mediation), protected software in the trusted computing base (tamperproofing), and the formal verification of the security kernel and other trusted software (verification), Scomp defined a process for building secure systems to satisfy the reference monitor concept. This process became the basis for the A1 evaluation level (i.e., the most secure evaluation level) of the DoD's Trusted Computer System Evaluation Criteria [304].

6.2.2 SCOMP HARDWARE

The Scomp hardware design is based on the ideas of the Multics system with two key changes. First, the Multics protection ring mechanism is concentrated in four protection rings and is extended to enforce more limited access for applications on ring transitions. Second, a *security protection module* is defined to mediate all memory and I/O accesses in the Scomp system.

First, the Scomp hardware implements four protection rings. The security kernel runs in the most privileged protection ring, ring 0, and user software runs in the least privileged ring, ring 3. Trusted software outside of the security kernel may occupy either ring 1 or 2.

The Scomp hardware supports a call-return mechanism that enables procedures in a less-privileged ring to invoke a procedure in a higher-privileged ring. The Scomp call-return mechanism is similar to the ring bracket access mechanism in Multics. However, Scomp hardware also provides a mechanism to access caller-supplied arguments at the caller's privilege level, called *argument addressing mode*. This mechanism enables the kernel to prevent itself from accessing data that the caller could not access. For example, the kernel interface can define a memory-mapping operation that requires the caller to supply a memory page. Using argument addressing mode, the kernel could only use the memory reference if the caller has access to this memory reference. This prevents the *confused deputy problem* [129] in a manner analogous to capability systems, see Chapter 10.

Second, the Scomp hardware includes a component called the *security protection module* (SPM) that provides a tamperproof service to mediate all memory and I/O accesses, as required for a reference monitor. As shown in Figure 6.2, the SPM mediates the main system bus (called the Level 6/DPS 6 bus) that provides access to peripherals and memory (e.g., memory and PCI buses combined). Also, the virtual memory interface unit uses the SPM to convert virtual address to physical segment addresses.

Each process has a *descriptor base root* that references the memory and I/O descriptors available to a process. Any virtual memory access references a descriptor that is used to authorize the request and access the physical location. Memory descriptors contain a pointer to physical memory, access permissions, and memory management data. Note that each access to words in memory is mediated by the SPM, so an access check accompanies any reference to memory. Since each instruction may make a memory reference for its instruction and its operands, plus access to page table entries, multiple access control checks may occur on each instruction. Accesses that hit in the hardware cache do not incur a memory reference and its accompanying authorization. Contrast this with modern systems, which authorize memory access at the page level, with a simple check on the page table or TLB entry for read or write access.

Mediation of I/O is similar to that for memory. An I/O operation is a request to a virtual name for a physical device. The SPM uses this virtual name to retrieve an I/O descriptor for the device. Access to this I/O descriptor is authorized similarly to memory descriptors. This mechanism supports two types of *direct memory access* (DMA) [2]. First, for premapped DMA, the SPM authorizes access to the device and the memory prior to the first request only. Since the I/O descriptors have been authorized and are used for each I/O request, subsequent requests no longer require authorization (i.e., the authorized descriptors are cached). Second, for mapped I/O, the I/O addresses for DMA are sent to the device. Thus, each access of the device to physical memory is authorized by the SPM (e.g., based on the process that owns the device).

Because the kernel builds the I/O descriptor and the hardware authorizes the I/O operation, it is possible to run I/O commands (i.e., drivers) in unprivileged processes (i.e., outside the kernel). This has performance and security advantages. First, once the access is mediated, the user process may interact with the devices directly, thus removing the need for kernel processing and context

[2] Direct memory access (DMA) enables devices to write into the system's physical memory without involving the system's CPU.

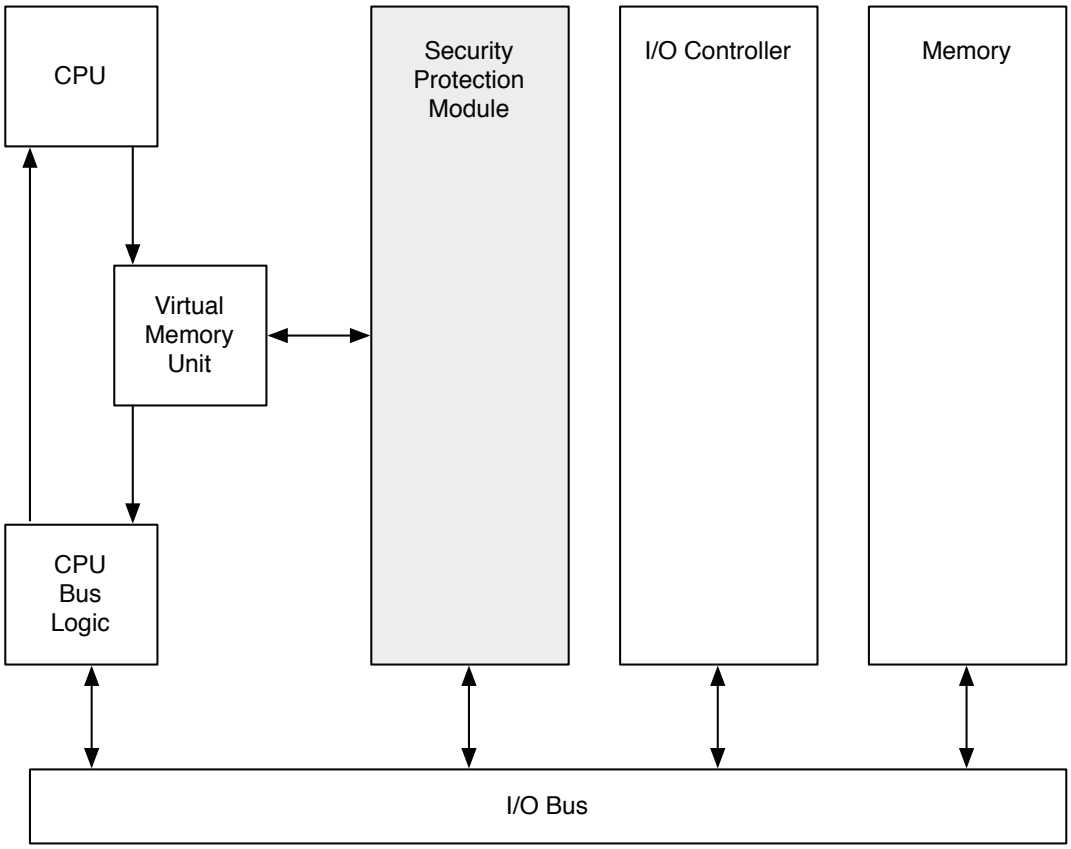

Figure 6.2: The Scomp *security protection module* (SPM) mediates all accesses to I/O controllers and memory by mediating the I/O bus. The SPM also translates virtual addresses to physical segment addresses for authorization.

switches. Second, drivers have been shown to be the source of many kernel errors [82], so the ability to remove them from the trusted computing base would improve software correctness. Further, in this architecture, new I/O devices can be added without modifying the kernel or the SPM, so the system's extensibility does not suffer from this approach.

Modern hardware, such as the x86 architecture, adopted the four-ring architecture of Scomp, but not the I/O mediation provided by the SPM. As a result, such systems are vulnerable to DMA devices. Buggy or malicious drivers can configure a DMA device to write at any physical memory location, so kernel memory may be overwritten. Since DMA bypasses the CPU, the kernel cannot prevent such writes in software. Recently, both Intel and AMD have released processors with I/O memory protection, called an I/O MMU [141, 8]. An I/O MMU also mediates an I/O bus (the

PCI bus), but the devices communicate using virtual addresses rather than I/O descriptors. The I/O MMU translates these *virtual I/O addresses* to physical addresses, authorizing access to the resultant physical address. The I/O MMU approach provides the advantages envisioned by the SPM architecture. For example, *passthrough I/O* built on I/O MMUs enables I/O to be conveyed directly from the device to untrusted processes.

The use of virtual I/O addresses rather than the I/O descriptors as in Scomp reflects the change from segmented protection of early systems to page protection of modern systems. While segmentation is still supported, modern operating systems use address spaces based on sets of pages managed by *page tables* rather than by managing memory segments.

6.2.3 SCOMP TRUSTED OPERATING PROGRAM

Technically, the Scomp Trusted Operating Program (STOP) consists of three components: (1) a *security kernel*; (2) a set of *trusted software*; and (3) a Scomp kernel interface package for user applications. We describe the first two in this subsection, and discuss user applications and the interface package in the following subsection.

Security Kernel The Scomp security kernel provides fundamental system processing in ring 0. The security kernel provides memory management, process scheduling, interrupt management, auditing, and reference monitoring functions. Consistent with the idea of a security kernel, the function of the Scomp security kernel is minimized to reduce the amount of trusted code. As a result, the Scomp security kernel is only 10K source lines of code, mostly written in Pascal.

Kernel objects consist of processes, segments, and devices. Each are identified by a globally-unique, immutable 64-bit identifier. The kernel maintains access control information and status data for each object. The Scomp access control model is essentially the same as the Multics approach, see Chapter 3, consisting of multilevel security (i.e., Bell-LaPadula sensitivity levels and category sets [23]), ring bracket policies, and discretionary policies. The ring bracket representation is modified to describe access based on the owner of the object, groups, or others. Thus, an owner may be allowed to access an object from a different set of rings than an arbitrary user.

The security kernel defines 38 *gates* for processes running outside the kernel to invoke kernel services. Gates are analogous to system calls in modern operating systems, providing function to create objects, map segments, and pin physical memory to virtual addresses. Scomp gates are also analogous to Multics gates in that they provide argument validation to protect the integrity of security kernel.

Trusted Software Scomp trusted software runs services that do not require ring 0 privilege, but provide functions that must be trusted to enforce control on user applications properly. There are two types of trusted software. The first type of trusted software is trusted not to violate system secrecy or integrity goals. For example, secure loader is trusted to load a process for any subject that ensures correct enforcement of that subject's information flows. The second type of trusted

software is trusted to maintain the security policy correctly. For example, services that modify user authentication data must be trusted.

Scomp has 23 processes that implement trusted functions, consisting of 11K source lines of code, written in the C language. There are three general types of trusted software. First, *trusted user services* provide an interface to Scomp for the user. User services include login, discretionary access control management, mandatory level selection, and process management. Second, *trusted operation services* provide functions that enable the system administrators to manage the system. Scomp require system management by operators who started the system, maintain mandatory policies (multilevel security and ring bracket policies), and collect and evaluate audit logs. Trusted operation services include a secure startup service and various operator commands (e.g.,setting time). Third, Scomp *trusted maintenance services* enable the system administrator to modify system data, such as install new versions of programs.

Scomp trusted software are invoked via a trusted communications path with the user. This is the familiar "secure attention sequence" used in modern systems. The purpose of this communication path is to prevent malicious software (e.g., Trojan horses) from masquerading as the user (e.g., trying to guess passwords and login). Further, the user also learns that she is communicating directly with the trusted software when the trusted communications path is invoked. Only the kernel can receive the interrupt invoked, so the user can be certain that the resulting response originated from trusted software.

6.2.4 SCOMP KERNEL INTERFACE PACKAGE

The Scomp kernel interface package (SKIP) provides a uniform interface for user applications to access trusted functions. SKIP code is divided into two parts. First, SKIP functions implement trusted operations on user-level objects: files via a hierarchical file system, processes via process management, and concurrent I/O via an event mechanism. SKIP functions are allowed to manipulate system state, so they are also trusted not to violate MLS requirements, like trusted software. Second, a SKIP library provides a high-level interface for accessing such functions. SKIP libraries run in the protection domain of user applications, so they are not trusted with system state. SKIP functions run in ring 2, and the libraries run with user applications in ring 3.

SKIP functions in ring 2 are also invoked via gates, similarly to the kernel and trusted software. For example, calls to modify the file system state, such as renaming a file, are invoked from a user application using the SKIP library which invokes a SKIP gate before being processed in ring 2. Thus, file system operations are protected from user applications.

SKIP provides a library for user applications to access files, modify file contents, and manage the file hierarchy. The actual file system operations, and its state, are maintained by the SKIP functions in ring 2, in the manner described above. All file operations are authorized based on the requestor's sensitivity level and ring number. This results in a file system hierarchy where the sensitivity level is nondecreasing from the root.

The SKIP library also enables applications to perform various kinds of I/O. In effect, the device drivers are provided in the SKIP library. As described above, the SPM provides a protection mechanism that mediates application access to devices and device access to memory segments. To enable concurrency control across multiple applications, the SKIP function provides an event mechanism in ring 2. The event mechanism processes interrupts, maintains a queue of requests, and provides event notification. Handlers may be defined by user applications, so they are run in the library.

6.2.5 SCOMP APPLICATIONS

The Scomp hardware and software is general purpose, but it defines a completely new application interface, so new application software needs to be constructed. Actually, the original idea was to run a UNIX emulator on Scomp, so UNIX applications could be run, but emulation was found to be too slow and too insecure as described above. There was also a plan to map UNIX system calls to the SKIP interface. The results of this effort do not appear to have been documented.

For example, the Scomp system was the basis for a *mail guard* [71]. Multilevel security ensures that secrets are protected from leakage, but in a military environment orders, often based on the analysis of secret information, must be conveyed to less cleared subordinates. In order to ensure that orders are delivered without leaking secrets, assured mail guards are run on the Scomp environment. The mail guards evaluate the content of the orders using specialized filters before forwarding them. Scomp is an ideal platform as it is assured to enforce the multilevel security requirements and execute the mail guard without allowing malicious software to take control of its execution.

Other general purpose applications used in an multilevel secure environment may also benefit from Scomp, so Honeywell developed other applications as well. For example, Honeywell implemented the Secure Office Management System for Scomp [100]. The software consists of a word processor, email, spreadsheet, database, and printing support. Such software provides a variety of features with each function, and is multilevel-aware to help the user navigate issues with information secrecy management and release. The challenge is that other vendors developed office software for ordinary operating systems with greater features, and these became the de facto standard for office processing. As a result, the government employees adopted this software and the insecure environments in which they run.

6.2.6 SCOMP EVALUATION

1. **Complete Mediation**: How does the reference monitor interface ensure that all security-sensitive operations are mediated correctly?

   Scomp performs all mediation in hardware, so complete mediation is always performed on the correct system resource.

2. **Complete Mediation**: Does the reference monitor interface mediate security-sensitive operations on all system resources?

All system resources are segments, memory and I/O, and all instructions access segments, so the hardware-based mediation of Scomp mediates all security-sensitive accesses.

The Scomp file system in ring 2 controls access to files, so higher-level, file policies may be written. However, initial access to file data depends on access to I/O. The file system must be trusted to prevent unauthorized access to one process' file data by another process.

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

   Hardware verification justifies complete mediation in Scomp.

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

   Scomp uses a protection rings to protect the security kernel from unauthorized modification. The security kernel runs in ring 0, and only 38 gates permit access to the kernel from other protection rings.

   Scomp uses a more complex version of the Multics discretionary ring bracket integrity model to express access to ring 0. Since it must be possible to update the kernel (e.g., by updating the file system), there are some subjects (and their processes) that could modify the kernel, including the protection system and reference monitor.

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

   Scomp also uses protection rings and bracket model to protect the integrity of the rest of its trusted computing base. The Scomp trusted computing base runs in rings 0, 1, and 2. No untrusted code is supposed to run in these rings. It is possible for untrusted processes in ring 3 to invoke code in the trusted computing base. The number of interfaces and associated gates that were implement to protect the trusted computing base is unclear.

6. **Verifiable**: What is basis for the correctness of the system's trusted computing base?

   The Scomp system design and its implementation's correspondence to that design were verified with formal analysis tools.

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

   Secrecy goals were enforced using a mandatory MLS policy for secrecy and discretionary bracket policies for integrity. Unlike prior systems, and most subsequent systems, the design of the system policies was also verified for correctness.

As a result, Scomp has mostly convincing answers to each of the questions above. While there are a few danger spots, such as the complexity of the interface to the trusted computing base, the

possibility of modifications to the discretionary bracket policy, and the inherent incompleteness of system verification, Scomp and other security kernels are about as close to secure operating systems as possible. The challenges for security kernel systems are their performance, practical utility, and maintenance complexity.

6.3 GEMINI SECURE OPERATING SYSTEM

Another security kernel system that emerged in the 1980s was Gemini Corporation's Standard Operating Systems (GEMSOS) [3] [290]. The Gemini company was founded in 1981 with the aim of developing a family of high-assurance systems for multilevel secure environments. Gemini aimed to build its security kernel (GEMSOS) from scratch, but unlike Honeywell's Scomp system, implement it to run on the available commercial hardware, the Intel x86 architecture. The GEMSOS kernel and some systems based on the kernel were assured at TCSEC A1 level [109, 306], as Scomp was. Importantly, GEMSOS is still an active product, supported by the Aesec Corporation [5].

As the GEMSOS architecture has significant similarities to Scomp, we provide a higher-level description of its design, highlighting the similarities and key differences.

Architecture The GEMSOS system architecture is shown in Figure 6.3. Like Scomp, GEMSOS consists of a security kernel and trusted software, but GEMSOS provides 8 protection rings, as does Multics. Since the x86 hardware only supports 4 rings, the security kernel runs in ring 0 and the trusted software in ring 1. The overall approach to security is derived from Multics as well, so secrecy protection is based on multilevel security labels and integrity is implemented by ring brackets.

GEMSOS does not define a kernel interface package, as Scomp does, but it does provide a library to make invoking the kernel gates easier, like the SKIP library. GEMSOS user-level processes (i.e., nonkernel code) access the security kernel using a Kernel Gate Library (KGL). This code runs outside the kernel, but because it may be used in user-level software that is part of trusted computing base, the KGL also is trusted code. This differs from the SKIP library which is not used in trusted software.

Security Kernel The GEMSOS security kernel design consists of a layered set of kernel functionality, shown in Figure 6.4. This design was influenced by a number of ideas, including the layering approach of Reed [253] and the information hiding approach of Parnas [241]. The result is that each layer is dependent only on the layers below them, and the state of each layer is only accessible via well-defined interfaces.

The GEMSOS security kernel is constructed from a base of generic functions to provide typical kernel services including memory, I/O, and process management, in addition to reference monitoring. The lowest four layers provide generic functions for accessing the hardware, switching execution contexts, and interrupt handling among others. The Kernel Device Layer then provides

[3] GEMSOS may alternatively be called GSOS by some.

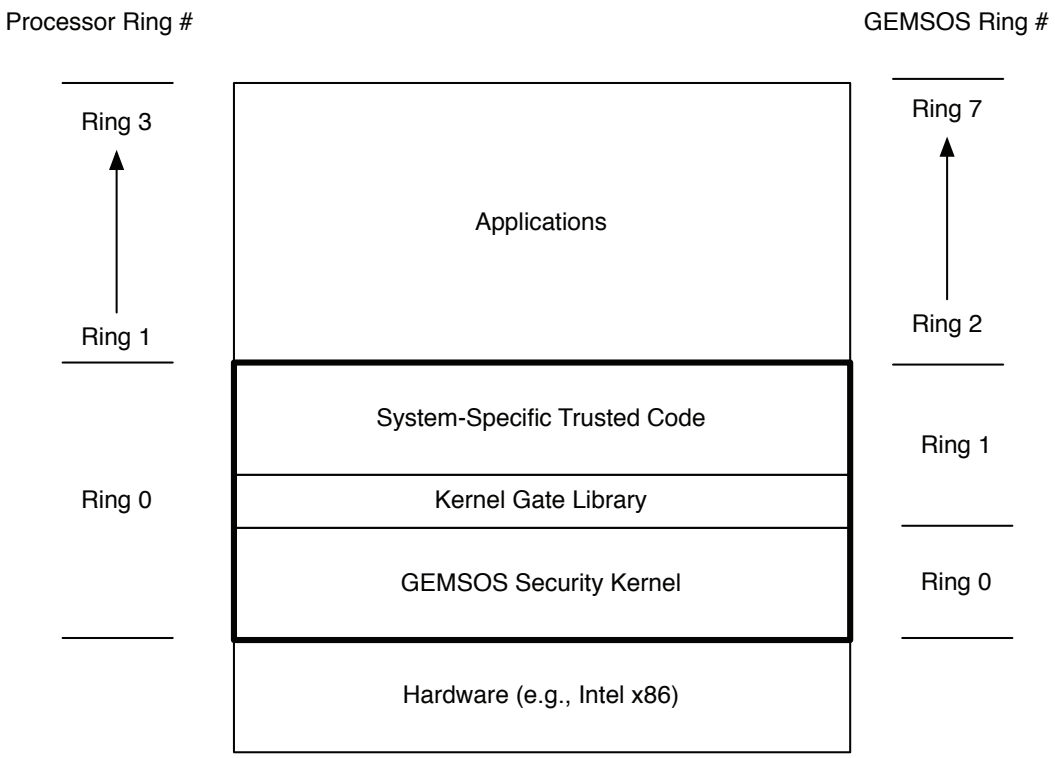

Figure 6.3: GEMSOS consists of a security kernel, gate library, and a layer of trusted software that is dependent on the deployed system. GEMSOS uses a software-based ring mechanism to simulate 8 protection rings.

other kernel layers with access to kernel-internal drivers. The Non-discretionary Access Control Layer implements the system reference monitor which enforces policies written in the Multics multilevel security model, see Chapter 3. The Secondary Storage Manager Layer provides the physical file system for GEMSOS user processes. Next comes the Internal Device Manager which provides the interface to device drivers. The Memory Manager Layer builds memory segments for kernel and user processes. The Upper Traffic Controller Layer provides support for multiprocessing using the concept of virtual processors. The top four layers, the Segment Manager Layer, the Upper Device Manager Layer, the Process Manager, and the Gate Layer all manage per-process resources: memory, I/O concurrency, processes, and system invocation, respectively.

The GEMSOS kernel architecture provides many of the services of ordinary kernels. But, the use of commercial hardware presented challenges to the designers. Because the x86 processor lacks the memory and device mediation of Scomp's Security Protection Module (SPM), device

Applications	
Gate Layer	↑
Process Manager (PM)	Process
Upper Device Manager (UDM)	Local
Segment Manager (SM)	↓
Upper Traffic Controller (UTC)	↑
Memory Manager (MM)	|
Inner Device Manager (IDM)	|
Secondary Storage Manager (SSM)	|
Non-Discretionary Security Manager (NDSM)	Kernel
Kernel Device Layer (KDL)	Global
Inner Traffic Controller (ITC)	|
Core Manager (CM)	|
Intersegment Linkage Layer (SG)	|
System Library (SL)	↓
Hardware	

Figure 6.4: GEMSOS Security Kernel Layers

drivers must be run in the GEMSOS kernel (e.g., in the Kernel Device Layer and Internal Device Manager). However, A1-level assurance requires verification of the correctness of all kernel (i.e., trusted computing base) code. Thus, as new devices and their drivers are introduced, this presents a management problem for the kernel. The availability of I/O MMUs [141, 8] would also enable the possibility of drivers outside the kernel.

The other major design similarity between GEMSOS and ordinary operating systems that differs from the Scomp is the presence of the file system in the kernel. In Scomp, the file system is implemented as part of the SKIP functional layer in ring 2. Recall that Scomp also included ring 2 software in the trusted computing base of the system. Later, researchers explored the design and implications of an untrusted file system on GEMSOS [146]. The GARNETS file system ran in a virtual machine outside the GEMSOS kernel, which results in an architecture similar to the Scomp approach. However, in the GARNET approach. the level of trust in the GARNET file system could be tangibly less than that of the kernel (i.e., it is not in the system TCB). The GARNET design required several workarounds to achieve the necessary functionality when this trust was removed, and may still require some trusted programs, albeit less trusted code than an entire file system.

GEMSOS defines 29 gates to access the security kernel, which is similar to the 38 gates provided by Scomp. The function offered by the gates are similar, although Scomp additionally provides function via the SKIP gates.

Trusted Software GEMSOS also provides a set of trusted software running outside the security kernel. The functions of the GEMSOS trusted software are similar to those provided by Scomp trusted software. For example, there are trusted software services for system administration.

Applications GEMSOS differs from Scomp in the number and variety of applications in which it was deployed. GEMSOS was commonly applied as a platform for securely connecting networked high security systems and isolating them from low security systems in the same network [242]. Also, GEMSOS was applied as a guard for an office software suite.

The most significant applications of GEMSOS were for network control. First, GEMSOS was the basis for the multilevel secure system called BLACKER [330]. BLACKER provided an A1-assured component for key distribution and secure communication to protect high secrecy data in transit between high secrecy networks. BLACKER consists of a set of encryption devices that enable the isolation of a high secrecy network from the rest of the Internet. Originally, different processors were used to handle the ciphertext (the *black* side) and the plaintext (the *red* side). Of course, it is important that ciphertext not leak the contents of plaintext, but if the ciphertext is created on a system with a Trojan horse, this cannot be guaranteed.

Second, GEMSOS was later applied to the general notion of a Trusted Network Processor (TNP) [306]. A TNP hosts one or more applications that require multilevel security enforcement. The applications themselves run on virtual processors in a multiprocessor system (potentially), but they are ignorant of multilevel security. The labeling of virtual processors and data and the enforcement of multilevel security are performed by the TNP component that mediates all communication on the multiprocessor bus.

A POSIX interface was developed for the GEMSOS system as well, although historically GEMSOS was applied to dedicated or embedded applications.

6.4 SUMMARY

The secure operating systems that followed the Multics system focused on the key limitations of Multics: performance and verifiability. The idea of a *security kernel*, a small kernel with minimal code in its trusted computing base, addressed both of these problems. First, the design of security kernel was customized to address performance bottlenecks, even by adding security function in hardware, such as Scomp's Security Protection Module. Second, the small size of the security kernels also motivated the development of system assurance methodologies to verify that these systems correctly implemented a secure operating system (see Definition 2.5). Multics was too large and complex to be verified, but both Scomp and GEMSOS were of manageable complexity such that the verification tools of the day, plus some manual effort, were sufficient to justify the correctness of these implementations. Such efforts led to the development of the assurance methodologies that we use today, see Chapter 12.

Security kernels are general purpose systems. The design of Scomp provides similar system function to UNIX systems of the day, and GEMSOS is a full kernel. Nonetheless, security kernels became niche systems. The performance, flexibility, and applications in UNIX systems and, later, Windows systems limited the market of security kernels to specialized, high security applications, such as guards. Further, the need to balance assurance and function became difficult for these security kernels. Maintaining the assurance of the kernel given the vast number of drivers that are developed, including some which are quite buggy, is very difficult. Scomp did not depend on drivers in the kernel, but it did depend on hardware features that were not available on common processors.

The need identified by security kernels has continued to exist. Scomp was succeeded by the XTS-300 and XTS-400 systems, now distributed by BAE [22]. GEMSOS is still available today from Aesec [5]. There have been other security kernel systems, including Boeing's secure LAN [298], Secure Computing Corp.'s LOCK system [293], and KSOS [198]. Also, the separation kernel systems (see Chapter 11) aim for a minimal, assured trusted computing base for deploying applications, and this architecture is also used frequently. As a result, it appears that the development of a minimal platform necessary to deploy the desired software is still the preferred option for security practitioners, although as we will see in next chapters, secure system alternatives that already support the desired applications becoming more popular and more secure.