A Retrospective on the VAX VMM Security Kernel

Paul A. Karger, *Member, IEEE*, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn

Abstract—**This paper describes the development of a virtual-machine monitor (VMM) security kernel for the VAX architecture. The paper particularly focuses on how the system's hardware, microcode, and software are aimed at meeting A1-level security requirements while maintaining the standard interfaces and applications of the VMS and ULTRIX-32 operating systems. The VAX Security Kernel supports multiple concurrent virtual machines on a single VAX system, providing isolation and controlled sharing of sensitive data. Rigorous engineering standards were applied during development to comply with the assurance requirements for verification and configuration management. The VAX Security Kernel has been developed with a heavy emphasis on performance and system management tools. The kernel performs sufficiently well that much of its development was carried out in virtual machines running on the kernel itself, rather than in a conventional time-sharing system.**

Index Terms — **Computer security, virtual machines, covert channels, mandatory security, discretionary security, layered design, security kernels, protection rings.**

I. Introduction

THE VAX Security Kernel project was a research effort to determine what is required to build a production-quality security kernel, capable of receiving an A1 rating from the National Computer Security Center (NCSC). A production-quality security kernel is very different from the many research-quality security kernels that have been built in the past, and this effort has been primarily aimed at identifying the differences and their cost in development effort and kernel complexity. While the VAX Security Kernel was a technical success, underwent a highly successful external field test, and had many interested potential customers, the Digital Equipment Corporation chose not to bring it to market.

This paper describes how the VAX Security Kernel met its five major goals:

- Meet all A1 security requirements (described below)
- Run on commercial hardware without special modifications other than microcode changes for virtualization
- Provide software compatibility for applications written for both the VMS and ULTRIX-32 operating systems
- Provide an acceptable level of performance
- Meet the requirements of a commercial software product.

Manuscript received November 1, 1990; revised July 15, 1991. Recommended by T. F. Lunt and D. Cooper.

P. A. Karger is with the Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142.

M. E. Zurko, D. W. Bonin, and C. E. Kahn are with the Digital Equipment Corporation, 295 Foster Street, Littleton, MA 01460.

A. H. Mason is with the Digital Equipment Corporation, Spit Brook Road, Nashua, NH 03063.

IEEE Log Number 9103531.

Attempting to meet all of these goals, represented a very ambitious undertaking, because no system had ever simultaneously met the security, performance, and software compatibility goals. Indeed, very few systems have ever met just the security goal, as most so-called secure systems have proven to be easily penetrable [1].

II. Background

This section presents a brief summary of the basics of computer security as background for the remainder of the paper. Most secure systems are based on an abstract notion of *subjects* and *objects*. The secure system must mediate access requests from the subjects, typically users or processes, to the objects that contain information, typically files, or memory. This section outlines the concepts of discretionary and mandatory controls and describes how the NCSC evaluates allegedly secure systems.

A. Discretionary and Mandatory Security Controls

Discretionary access controls are the commonly available security controls found in most operating systems. They are called discretionary, because the access rights to an object may be determined at the discretion of the owner or controller of the object. Both access-control-list and capability systems are examples of discretionary access controls. The presence of Trojan horses in applications software can cause great difficulties with discretionary controls, because a Trojan horse could surreptitiously change the access rights on an object or could make a copy of protected information and give that copy to some unauthorized user. All forms of discretionary controls are vulnerable to this type of Trojan-horse attack. A Trojan horse in an access-control-list system could surreptitiously change the ACL of an object. A Trojan horse in a capability system could make a copy of a capability for a protected object, and then store that capability in some other object to which a penetrator would have read access. In both cases, the information is disclosed to an unauthorized recipient.

Lampson [2] has defined the *confinement problem* as determining whether there exists a series of operations in a security system that will ultimately leak some information to some unauthorized individual. Harrison *et al.* [3] have shown that there is no solution to the confinement problem for fully general, discretionary access controls, such as either a general access-control-list or capability system. Their argument is based on modeling the state transitions of the access control lists as the state transitions of a Turing machine. They show that solving the confinement problem is equivalent to solving the Turing-machine halting problem.

The paths over which a Trojan horse leaks information are called *covert channels.* Covert channels can be divided into two major categories: *storage channels* and *timing channels.* Information can be leaked through a storage channel by changing the values of any of the state variables of the system. Thus contents of files, names of files, and amount of disk space used are all examples of potential storage channels. A Trojan horse can leak information through a storage channel in a purely asynchronous fashion. There are no timing dependencies.

By contrast, information can be leaked through a timing channel by modifying the length of time that system functions take to complete. For example, a Trojan horse could encode information into deliberate modifications of the system page-fault rate. Timing channels all use synchronous communication and require some form of external clocking.

Mandatory access controls have been developed to deal with the Trojan horse problems of discretionary access controls. The distinguishing feature of mandatory access controls is that the system manager or security officer may constrain the owner of an object in determining who may have access rights to that object.

Lipner [4] and Denning [5] have shown that for *lattice security models,* unlike for fully general access matrices, it is possible to solve the confinement problem. All mandatory controls, to date, have been based on lattice security models.

A lattice security model consists of a set of *access classes* that form a partial ordering. Any two access classes may be less than, greater than, equal to, or not ordered with respect to one another. Two access classes that are not ordered are called *disjoint.* Furthermore, there exists a lowest access class, called *system low,* such that system low is less than or equal to all other access classes, and there exists a highest access class, called *system high,* such that all other access classes are less than or equal to system high.

A very simple lattice might consist of two access classes: LOW and HIGH. LOW is less than HIGH. LOW is system low, and HIGH is system high. A slightly more complex example might be a list of secrecy levels, such as UNCLASSIFIED, CONFIDENTIAL, SECRET, and TOP SECRET. Each level in the list represents data of increasing secrecy.

There is no requirement for a strict hierarchical relationship between access classes. The U.S. military services use a set of access classes that have two parts: a secrecy level and a set of categories. Categories represent compartments of information for which an individual must be specially cleared. To gain access to information in a category, an individual must be cleared, not only for the secrecy level of the information, but also for the specific category. For example, if there were a category NUCLEAR, and some information classified SECRET-NUCLEAR, then an individual with a TOP SECRET clearance would not be allowed to see that information, unless the individual were specifically authorized for the NUCLEAR category.

Information can belong to more than one category, and category comparison is done using subsets. Thus in the military lattice model, for access class A to be less than or equal to access class B, the secrecy level of A must be less than or equal to the secrecy level of B, and the category set of A must be an improper subset of the category set of B. Since two category sets may be disjoint, the complete set of access classes has only a partial ordering. There is a lowest access class, {UNCLASSIFIED-no categories}, and a highest access class, {TOP SECRET-all categories}. The access classes made up of levels and category sets form a lattice.

B. Overview of NCSC Criteria

The NCSC has developed computer security evaluation criteria [6] to aid DoD agencies in the procurement of secure computer systems. The criteria divide computer security systems into four major divisions, with classes within those divisions. Computer vendors submit their operating systems to the NCSC for design assistance and ultimately formal evaluation against the criteria. A number of commercially available systems have been successfully evaluated against the criteria. At least one commercial system has been evaluated in each of the four major divisions.

- **Division D: Minimal Protection**. This division contains only one class. It is reserved for those systems that have been evaluated, but that fail to meet the requirements for a higher evaluation class.
- **Division C: Discretionary Protection**. Classes in this division provide for discretionary (need-to-know) protection:
  Class (C1): Discretionary Security Protection: Class (C1) systems provide a minimal set of security features to separate users and their data. Most conventional time-sharing systems fall into this class.
  Class (C2): Controlled Access Protection: Class (C2) systems require a finer grained control system than class (C1) systems. For example, simple owner/group/world protection schemes would be unacceptable at class (C2). Class (C2) systems must also provide improved audit trails and login control procedures.
- **Division B: Mandatory Protection**. Classes in this division provide an implementation of the mandatory lattice security model:
  Class (B1): Labeled Security Protection: Class (B1) systems must label all storage objects and enforce the lattice security model on those objects. However, covert channels are not addressed in this class.
  Class (B2): Structured Protection: Class (B2) systems must label all system resources (as opposed to only storage objects), and must show that covert channels have either been eliminated or bandwidth limited. Also, a trusted communications path between the user and the system must provide two-way authentication.
  Class (B3): Security Domains: Class (B3) systems are required to isolate the security functions from the rest of the operating system, typically into some form of security kernel. At this class, access control lists are explicitly required. An informal descriptive top-level specification (DTLS) of the design is required.
- **Division A: Verified Protection**. Division A systems are characterized by the use of formal mathematical meth-

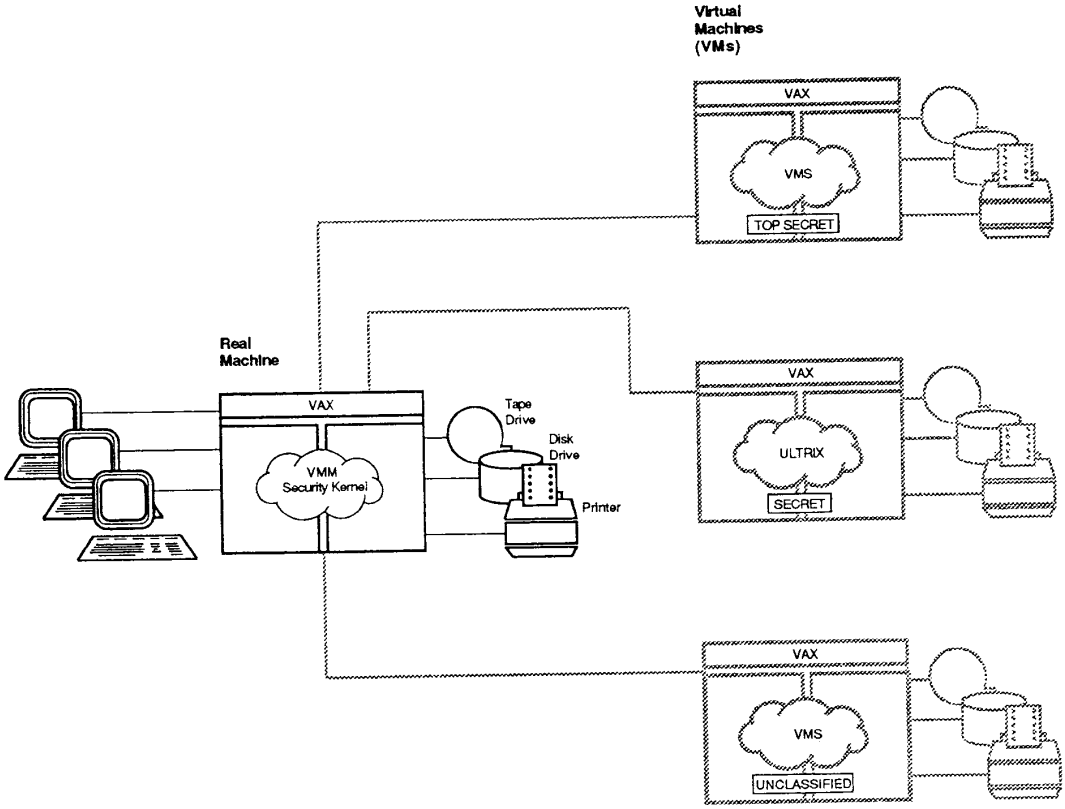

Fig. 1. VAX VMM Security Kernel configuration.

ods to assure correctness of design and implementation:

Class (A1): Verified Design: Class (A1) systems require the preparation and verification of a mathematically formal top-level specification (FTLS) of the security kernel design. Informal techniques must be used to show correspondence between the FTLS and the implemented software.

Beyond Class (A1): Classes beyond A1 will probably require formal verification of the code of the security kernel and some considerations of microcode and hardware correctness. However, constructing systems at this level of security is still beyond current technology, so requirements have not yet been stated.

III. Kernel Overview

The VAX Security Kernel is a virtual-machine monitor that runs on the VAX 8530, 8550, 8700, 8800, and 8810 processors.[1] It creates isolated virtual VAX processors, each of which can run either the VMS or ULTRIX-32 operating system. If desired, virtual machines running each of the operating systems can run simultaneously on the same computer system.[2]

[1]The VMM does not run on VAX 8820, 8830, or 8840 processors, due to microcode and console differences.

[2]At least one virtual machine must always run the VMS operating system, to carry out certain system management functions.

The VAX architecture was not virtualizable, and therefore extensions were made to the architecture and to the processor microcode to support virtualization.

Fig. 1 shows a typical VAX Security Kernel configuration. While the VAX Security Kernel is a VMM, it is primarily a security kernel. Therefore, certain features traditionally seen in VMM's, such as self-virtualization or debugging of one VM from another, have been omitted to reduce kernel complexity.

The VAX Security Kernel applies both mandatory and discretionary access controls to virtual machines. Each virtual machine is assigned an *access class*, which consists of a *secrecy class* and an *integrity class*, similar to those in the VMS Security Enhancement Service (VMS SES) [7]. The secrecy and integrity classes are based on the Bell and LaPadula security [8] and Biba integrity [9] models, respectively. The VAX Security Kernel also supports access control lists (ACL's) on all objects, similar to those in the VMS operating system [10].

The VMM security kernel is *not* a general-purpose operating system. The principal subjects and objects are virtual machines and virtual disks, rather than conventional processes and files. That is the inherent difference between a VMM and a traditional operating system. Processes and files are implemented within the virtual machines by either the VMS or ULTRIX-32 operating systems.

The VAX Security Kernel can support large numbers of simultaneous users.[3] Once a basic system was operational, all software development of the VAX Security Kernel was carried out on several virtual machines running on the VMM on a VAX 8800 system. On a typical day, about 40 software engineers and managers were logged in running a mixed load of text editing, compilation, system building, and document formatting. The system provided adequate interactive response time and is sufficiently reliable to support an engineering group that must meet strict milestones and schedules. As far as we know, the VAX Security Kernel was the first security kernel to support its own development team. The Multics Access Isolation Mechanism [11] was developed on Multics itself, but Multics with AIM was not a security kernel and only received a B2 rating.

At the time of the cancellation, the VAX Security Kernel was about to enter the Formal Evaluation Phase with the NCSC for an A1 rating. It was formally specified in Ina Jo and formal proofs were underway on the specifications.

IV. Design Approach

This section describes several of the design choices in the VAX Security Kernel, including details about the virtual machine approach to security kernels, virtualizing the VAX architecture, subjects and objects, access classes, our layered design, and other software engineering issues.

A. Virtual Machine Approach

The choice to build the VAX Security Kernel as a VMM was driven by two goals: to maintain compatibility with existing software written for the VAX architecture, and to keep software development and maintenance costs to a minimum.

We began plans to enhance the security of the VAX architecture in mid-1979. Our initial effort was the design of security enhancements to the VMS operating system, first prototyped in 1980 and available today in the base VMS operating system and in the VMS Security Enhancement Service [7].

At the time of the initial prototype of the VMS security enhancements [12], Digital considered a traditional kernel/emulator security kernel to support VMS applications. However, it quickly became clear that the software development costs of a VMS emulator would be comparable to the cost of development of the VMS operating system itself. Worse still, the emulator would have to track all changes made to the VMS operating system, resulting in ongoing costs that would be unacceptably high for the limited market for A1-secure systems. The kernel/emulator system could not replace the existing VMS operating system, because its performance would not be as good, and it would likely be export-controlled. Furthermore, the growing demand for UNIX-based software would force development of a UNIX emulator at still more development cost.

To resolve these development cost and compatibility problems, we chose a VMM security kernel approach. A VMM security kernel presents the interface of a computer architecture that is comparatively simple and not subject to frequent change. Thus the VAX Security Kernel presents an interface of the VAX architecture [13] and supports both the VMS and ULTRIX-32 operating systems with relatively few modifications.

The idea of a VMM security kernel is not a new one. Madnick and Donovan [14] first suggested the merits of VMM's for security, and Rhode [15] first proposed VMM security kernels. From 1976 to 1982, System Development Corporation (now a part of the UNISYS Corporation) built a kernelized version of IBM's VM/370 virtual-machine monitor, called KVM/370 [16]. While the design of the VAX Security Kernel is very different from KVM/370, we have applied some of the lessons learned in the KVM/370 project [17]. Section VIII compares the VAX Security Kernel with KVM/370. Gasser [18, sec. 10.7] provides more detail on some of the trade-offs between a VMM security kernel approach and a kernel/emulator approach.

B. Virtualizing the VAX

The requirements for virtualizing a computer architecture were specified by Popek and Goldberg [19]. In essence, they require that all sensitive instructions and all references to sensitive data structures trap when executed by unprivileged code. A *sensitive* instruction or data structure is one that either reveals or modifies the privileged state of the processor.

1) Sensitive Instructions: Unfortunately, the VAX architecture does not meet Popek and Goldberg's requirements. Several instructions, including Move Processor Status Longword (MOVPSL), Probe (PROBEx), and Return from Exception or Interrupt (REI) are sensitive, but unprivileged. Furthermore, page table entries (PTE's) are sensitive data structures that can be read and written with unprivileged instructions.

As a result, we made a number of extensions to the VAX architecture to support virtualization. In particular, we added a VM bit to the processor status longword (PSL) that indicated whether or not the processor was executing in a virtual machine. A variety of sensitive instructions were changed to trap based on the setting of the VM bit, so that the VMM security kernel could emulate their execution. Space does not permit a full discussion of the instruction changes. More complete descriptions can be found in Karger, Mason, and Leonard's patent [20] and in Hall and Robinson's paper [21] on virtualization of the VAX architecture.

2) Ring Compression: The most significant and security-relevant change to the VAX architecture was to virtualize protection rings. In the past, only processors with two protection states (such as the IBM 360/370 architecture) had been virtualized. Goldberg [22, sec. 4.3] described the difficulties of virtualizing machines with protection rings and therefore more than two protection states. He proposed several techniques for mapping ring numbers, some in software and one with a hardware ring-relocation register, but he recognized that none of his techniques were satisfactory. His software techniques broke down because the physical ring number remained visible, and his hardware ring-relocation technique broke down because virtualizing a machine with N rings always required $N + 1$ rings.

[3] Exact numbers depend on the precise hardware configuration.

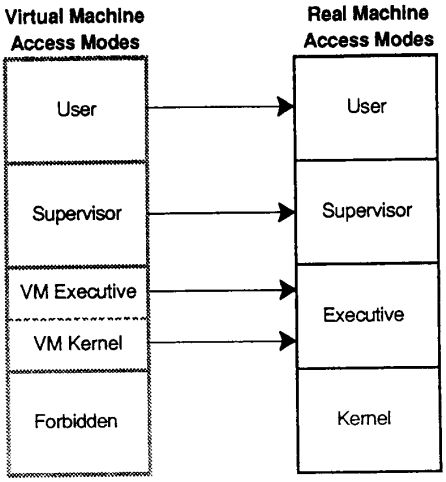

Fig. 2. Ring compression.

Since the VMS operating system uses all four of the protection rings of the VAX architecture, it was essential that we develop a new technique for virtualization of protection rings. That technique is called *ring compression*.

Fig. 2 shows how the protection rings of a virtual VAX processor are mapped to the rings of a real VAX processor. Virtual user and supervisor modes map to their real counterparts, but virtual executive and kernel modes both map to real executive mode. The real ring numbers are concealed from the virtual machine's operating system (VMOS) by three extensions to the VAX architecture: the addition of the VM bit to the PSL, the addition of a VM processor-status longword register (VMPSL), and the modification of all instructions that could reveal the real ring number. Those instructions, either trap to the VMM security kernel for emulation or obtain their information from the VMPSL, which contains the virtual ring numbers rather than the real ring number. Additional details can be found in Karger *et al.*'s patent [20] and in Hall and Robinson's paper [21].

Ring compression also requires that the security kernel change the memory protection of pages belonging to virtual machines so that their kernel-mode pages become accessible from executive mode. This change of memory protection could adversely affect security within a given virtual machine, because the virtual machine's kernel mode is no longer fully protected from its executive mode.

For the two operating systems of interest to the VAX Security Kernel, there is no effective loss of security within the virtual machines themselves, although there is a loss of robustness against the potentially bug-laden executive mode code. Fortunately, the VMS operating system grants all programs that run in the executive mode the right to change mode to kernel at will, and uses the kernel/executive mode boundary only as a reliability mechanism. Furthermore, the ULTRIX-32 operating system does not use the executive mode at all.

Of course, the compression of kernel and executive modes in the virtual machines in no way compromises the security of the VMM, as the security kernel runs only in real kernel mode, and no virtual machine ever is granted access to real kernel mode pages. If there were some other VAX operating system which actually used all four rings for security purposes, it would lose some of its own security, much as IBM operating systems lose some of their security when run in VM/370. However, no such operating systems exist for the VAX architecture.

3) I/O Emulation: Traditional virtual-machine monitors, such as IBM's VM/370, have virtualized not only the CPU, but also the I/O hardware. Virtualization of the I/O hardware allows the VMOS to run essentially unmodified. Virtualization of the VAX I/O hardware is particularly difficult, because its I/O devices are programmed by reading and writing control and status registers (CSR's) that are located in a region of physical memory called I/O space. This type of I/O originated on the PDP-11 series of computers and caused performance difficulties in the UCLA PDP-11 virtual-machine monitor [23], because the VMM must somehow simulate every instruction that manipulates a CSR. Vahey [24] proposed a complex hardware performance assist, but such a device would add excessive complexity and development cost to the VAX Security Kernel.

Instead, the VAX Security Kernel implements a special I/O interconnection strategy for virtual machines. The VAX architecture [13] does not specify how I/O is to be done, and different VAX processors have implemented very different I/O interfaces. The VAX Security Kernel I/O interface is a specialized kernel call mechanism, optimized for performance, rather than traditional CSR-based I/O. In essence, a virtual machine stores I/O-related parameters (such as buffer addresses, etc.) in specified locations in its I/O space, but no I/O takes place until the virtual machine executes a Move to Privileged Register (MTPR) instruction to a special kernel call (KCALL) register. This MTPR traps to security kernel software that then performs the I/O. Thus the number of traps to kernel software is dramatically less than would be required for CSR emulation.

This special kernel I/O interface means that special untrusted virtual device drivers had to be written for both the VMS and ULTRIX-32 operating systems, but this effort was no more than is typically required to support a new VAX processor, a small number of engineer-years.

Because the virtual VAX processors have an I/O interface different from that of any existing VAX processors, the VAX Security Kernel does not fall into any of Goldberg's traditional categories of VMM's. Goldberg [22, pp. 22–26] defines a Type I VMM as a VMM that runs on a bare machine. He defines a Type II VMM as a VMM that runs under an existing host operating system. Goldberg [22, sec. 3.3] also defines a Hybrid Virtual-Machine Monitor as one in which all supervisory-state instructions are simulated, rather than just the privileged instructions. The VMM security kernel is essentially a cross between a self-virtualizing Type I VMM for all non-I/O instructions and a Hybrid Virtual-Machine Monitor for I/O instructions.

4) Self-Virtualization: As we designed the extensions to the VAX architecture, we ensured that the architecture would permit *self-virtualization*. Self-virtualization is the ability of a virtual-machine monitor to run in one of its own virtual

machines and recursively create second-level virtual machines. Self-virtualization is very useful for developing and debugging the virtual-machine monitor itself, but it is of little value to actual users. Since self-virtualization would have added significant complexity to the Trusted Computing Base (TCB), no software support has been done.[4]

C. Subjects

There are two kinds of subjects in the VAX Security Kernel: users and virtual machines (VM's). A user communicates over the trusted path with a process called a *Server*. Servers are trusted processes, but unlike the trusted processes in other systems such as KSOS-11 [26], servers run only within the kernel itself. User subjects cannot run user-written code; servers execute only trusted code that is part of the TCB.

The powers of a Server are determined by:

- The user's minimum and maximum access class
- The terminal's minimum and maximum access class
- The user's discretionary access rights
- The user's privileges
- The privileges exercisable from the terminal.

A virtual machine is an untrusted subject that runs a VMOS. A user interacts with the VMOS in whatever fashion is normal for that operating system; for example, by logging into that VMOS and issuing commands. A user may write and run code inside a VM and even penetrate the VMOS, all without affecting the security of other virtual machines or the security kernel itself. At worst, a penetrated virtual machine could only affect other virtual machines with which it shared disk volumes.

On login to the security kernel, the VMM establishes a connection between the user's terminal line and the user's Server, called a *session*. When the user wants to use some virtual machine, the user issues the `CONNECT` command to his or her Server, specifying the name of that VM. If the connection is authorized, the system suspends the user's existing session with the Server and establishes a new session between the user's terminal line and the requested virtual machine. Thus the Servers and the VM's have distinct identities and distinct security attributes.

Virtual machines may be run in a single-user mode to provide maximum individual accountability. Alternately, they can be run in a multiuser mode. In such a case, individual accountability might be achieved by running a VMOS with at least a C2 rating, as suggested by the proposed Trusted VMM Interpretation [25] of Trusted Information Systems, Inc.

Virtual machines can also be treated as objects, because a user may request that the TCB provide a connection between the user's terminal and some VM. For this operation, the user is the subject and the VM is the object.

[4]The software changes needed for self-virtualization primarily consist of changes to the virtual device drivers and some changes in the emulation of certain sensitive instructions. Under the proposed Trusted VMM Interpretation [25], it might even be possible for a self-virtualized security kernel to itself remain A1 rated. To achieve that goal, the first-level VMM would map the second-level VMM's kernel mode to real executive mode, while the VM's running on top of the second-level VMM would have their supervisor, executive, and kernel modes all mapped to the real supervisor mode. Of course, as one continues to recursively self-virtualize, one runs out of protection rings at the fourth-level VMM, and that VMM would no longer be protected from its virtual machines.

D. Objects

The VAX Security Kernel supports a variety of objects, including real devices and volumes and security kernel files.

One group of objects comprises the real devices on the system: disk drives, tape drives, printers, terminal lines, and single access-class network lines. As these devices can contain or transmit information, access to them must be controlled by the TCB. Another object is the primary memory which is allocated to each VM when it is activated.

Disk and tape volumes are also objects. The contents of some disk volumes are completely under the control of a virtual machine. They may contain a file system structure of just an arbitrary collection of bits, depending on the method used by the VMOS to access the volume. Such volumes are called *exchangeable volumes*, because they may be exchanged with other computer systems running conventional operating systems. Other disk volumes contain a VAX Security Kernel file structure and are called *VAX Security Kernel volumes*. These volumes must not be directly accessed by a VMOS or exchanged with other systems, as an untrusted subject could subvert the kernel's file system or read information to which it was not entitled. The VAX Security Kernel does not provide trusted tape volumes; all tape volumes are exchangeable.

VAX Security Kernel volumes contain VAX Security Kernel files which are organized as a flat file system. VAX Security Kernel files are used for a variety of purposes in the system and are considered objects by the TCB. One use for VAX Security Kernel files is to hold long-term system databases such as the audit log and authorization file. These files are considered part of the TCB and, with the exception of the audit log, error log, and crash dump files, cannot be directly referenced by virtual machines.

Another use of VAX Security Kernel files is to create virtual disk volumes, loosely analogous to mini-disks in IBM's VM/370 [27, pp. 549–563]. Mini-disks allow a physical disk to be partitioned, so that one need not dedicate an entire physical disk to a small virtual machine that only requires a small amount of disk space. Such virtual disks may contain the file structure of some VMOS, such as a VMS file structure or an ULTRIX-32 file structure. However, the VMM deals with virtual disks only as a whole. The contents of a virtual disk are all part of a single object as far as the VMM is concerned.

E. Access Classes

The VAX Security Kernel enforces mandatory access controls, as required of all A1 systems. Both secrecy and integrity models are supported, based on the work of Bell and LaPadula [8] and of Biba [9], respectively. To implement mandatory access controls, each kernel subject and kernel object is assigned a sensitivity label, called an *access class*.[5] An access class consists of two components: a *secrecy class* and an *integrity class*. These components are each further divided into

[5]Some objects, such as terminal lines, may be assigned a range of access classes.

TABLE I
USER PRIVILEGES

Privilege	Powers
CLASSIFY_DEVICE	Assign access classes to I/O devices and privileges to terminals
CLASSIFY_SUBJECT	Assign access classes and privileges to subjects; name levels and categories
CLASSIFY_VOLUME	Register and assign access classes to volumes
DELETE_AUDIT	Delete audit data
DOWNGRADE_SECRECY	Downgrade secrecy of text after human inspection
DOWNGRADE_SECRECY_NOINSPECT	Downgrade secrecy of data without inspection
ENABLE_DEBUGGER	Enable untrusted kernel debugger
OPERATE	Mount volumes, change printer paper, boot and shutdown system
REGISTER	Register and change non-security attributes of devices, virtual machines, and users
SET_AUDIT	Control audit log and real-time alarms
SET_COVERT_CHANNEL_DEFENSE	Enable or disable covert channel defenses
SET_FILE	Create, delete, or copy kernel files
SET_PASSWORD	Change users' passwords and password parameters
UPGRADE_INTEGRITY	Upgrade integrity of text after human inspection
UPGRADE_INTEGRITY_NOINSPECT	Upgrade integrity of data without inspection

TABLE II
VIRTUAL MACHINE PRIVILEGES

Privilege	Powers
OPERATE	Dismount volumes; activate and deactivate other virtual machines; set login limits
SET_ACL	Change any object's ACL, if access class permits

a level and a category set. A *secrecy level* is a hierarchical classification. The *secrecy category set* is the set of nonhierarchical secrecy categories which represents the sensitivity of the access class. The integrity level and integrity category set are defined analogously. For compatibility with VMS SES [7], the kernel supports 256 secrecy levels, 256 integrity levels, 64 secrecy categories, and 64 integrity categories.

Given the complex structure of access classes, two definitions must be carefully constructed:

Definition 1

An access class A is *equal* to an access class B if and only if:

- The secrecy level of A is equal to the secrecy level of B
- The secrecy category set of A is equal to the secrecy category set of B
- The integrity level of A is equal to the integrity level of B, and
- The integrity category set of A is equal to the integrity category set of B.

Definition 2

An access class A *dominates* an access class B if and only if:

- The secrecy level of A is greater than or equal to the secrecy level of B
- The secrecy category set of A is a superset of the secrecy category set of B
- The integrity level of A is less than or equal to the integrity level of B, and
- The integrity category set of A is a subset of the integrity category set of B.

It is important to note that if two access classes are equal, each also dominates the other. This is because if P is a subset of Q, then P may contain some or all of the Q's members, while if P is a proper subset of Q, P must contain fewer members than Q. This terminology for subset relationships is based on Halmos [28, p. 3]. The corresponding relationships apply for supersets and proper supersets.

The secrecy and integrity models define that a subject may reference an object depending on the access classes of the subject and object and on the type of reference. A subject may read from an object only if the subject's access class dominates the access class of the object. A subject may write to an object only if the object's access class dominates the access class of the subject.[6] Thus, for example, a virtual machine may mount for read–write access an exchangeable volume only if the VM's access class is equal to that of the volume. However, the VM may mount for read-only access any exchangeable volume where the VM's access class dominates that of the volume.

F. Privileges

System managers, security managers, and operators gain their powers by having *privileges*. The privileges allow great flexibility in the assignment of powers and responsibilities, including a measure of two-person control and separation of duties. Privileges restrict access beyond the protections provided by mandatory and discretionary access controls. A privileged user cannot see data that would be otherwise inaccessible. Only the downgrading privileges allow bypassing of access controls, and the use of those privileges is audited.

Most privileges can be exercised only through the trusted path, and are called *user privileges* (see Table I). Two privileges can be exercised by virtual machines, and are called *virtual-machine privileges* (see Table II).

[6] In general, write access is even further restricted; a subject may write to an object only if the subject's and object's access classes are equal. This disallows blind writes to an object that cannot be read.

G. Layered Design

The VAX Security Kernel was implemented following the strict *levels of abstraction* approach originally used by Dijkstra [29] in the THE system. Janson [30] developed the use of levels of abstraction in security kernel design as a means of reducing complexity and providing precise and understandable specifications. Each layer of the design implements some abstraction in part by making calls on lower layers. In no case does a lower layer invoke or depend upon higher layer abstractions. By making lower layers unaware of higher abstractions, we reduced the total number of interactions in the system and thereby reduce the overall complexity. Furthermore, each layer can be tested in isolation from all higher layers, allowing debugging to proceed in an orderly fashion, rather than haphazardly throughout the system. This type of layering is called out in the requirements for B3 and A1 systems when the NCSC evaluation criteria [6, p. 38] state that: "The TCB shall incorporate significant use of layering, abstraction and data hiding. Significant system engineering shall be directed toward minimizing the complexity of the TCB . . . ."

The layered design of the VAX Security Kernel was based heavily on the Multics kernel design work of Janson [30] and Reed [31], and to a lesser extent on the Naval Postgraduate School kernel design [32]. Fig. 3 shows a diagram of the VAX Security Kernel. The arrows in the diagram indicate how each layer functionally depends on the abstract machine created by lower layers.

Each layer adds specific functions with security kernel, such that at the security perimeter, the secrecy and integrity models are enforced. The kernel itself is process-structured, as described in the summary of the various layers. Unlike many other kernels, all of the trusted processes run within the security perimeter and are included in the formal specifications of the system.

1) HIH: The Hardware-Interrupt Handler layer is immediately above the physical VAX hardware and modified microcode. It contains the interrupt handlers for the various I/O controllers and certain CPU-specific code.

2) LLS: The Lower-Level Scheduler is based strongly on Reed's two-level scheduler design [31]. It creates the abstractions of level one virtual processors (vp1's) that are the basic unit of scheduling for the system. The LLS supports symmetric multiprocessing by binding and unbinding real CPU's to individual vp1's. As shown in Fig. 4, there are three kinds of vp1's: *dedicated* vp1's that typically contain device drivers, *bindable* vp1's that can be bound to dedicated vp2's by the higher level scheduler, and *addressable* vp1's that can be bound to bindable vp2's and thereby to virtual machines. Vp1's are intended to be very inexpensive processes for use within the kernel. Only addressable vp1's have full address spaces; all other vp1's run out of the global address space of the kernel. Thus the lower-level scheduler can context switch in and out of most vp1's by merely saving registers and switching stack pointers. The lower-level scheduler also implements eventcounts [33] as the basic synchronization mechanism of the kernel. Eventcounts can be awaited or advanced in the normal way, or a *processor interrupt* can be tied to an eventcount, such that a VM can be interrupted when an eventcount has reached a particular value. This processor-interrupt mechanism provides upward transfers of control that are otherwise forbidden in the kernel. Processor interrupts are only delivered when the CPU is executing outside the security kernel.

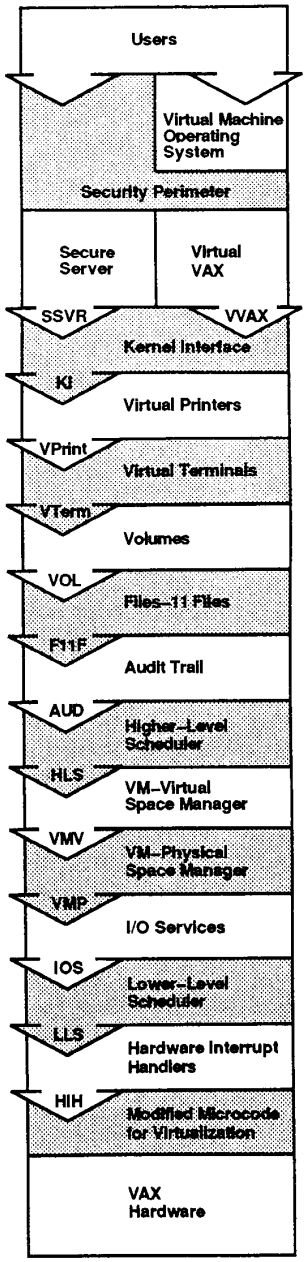

Fig. 3. VAX Security Kernel layers.

3) IOS: The I/O services layer implements device drivers that control the real I/O devices. The current version supports only directly connected terminals and storage devices.

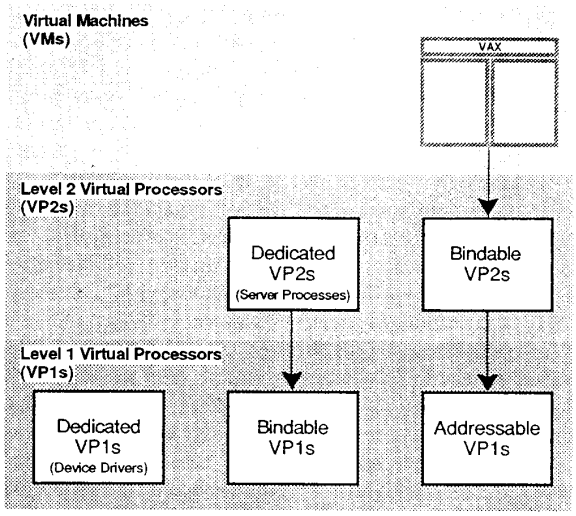

Fig. 4. Level-one and level-two virtual processors.

4) VMP: The VM physical memory layer manages real physical memory, and assigns it to virtual machines.

5) VMV: The VM virtual memory layer implements the *shadow page tables* needed to support virtual memory in the virtual machines.[7] VMV implements a primary-memory-only strategy, requiring that all the physical memory that a virtual machine sees be physically resident when that virtual machine is active. While this technique limits the number of simultaneously active virtual machines to the number that can fit into physical memory simultaneously, it significantly reduces kernel complexity by eliminating the need for a demand-paging mechanism in the kernel. It also eliminates the phenomenon of *double paging* that is often seen in other VMM's, where the demand paging mechanisms of the VMM and of the VMOS can thrash against one another, leading to serious performance degradation. In the VMM security kernel, the virtual machines are allocated a fixed amount of physical memory and do all their own paging.

6) HLS: The Higher-Level Scheduler is also based on Reed's two-level scheduler [31]. Unlike Reed's design, our higher level scheduler is extremely simple, because it does not need to schedule access to primary memory. The HLS does create the abstraction of level-two virtual processors (vp2's). There are two kinds of vp2's: *dedicated* vp2's which are used primarily by the SSVR layer described below, and *bindable* vp2's which are used for virtual machines. Fig. 4 shows the relationships between vp1's and vp2's.

7) AUD: The auditing layer provides the facilities for security auditing and security alarms. It is described in detail in [34].

8) F11F: The Files-11 Files layer implements a subset of the ODS-2 file system that is also used in the VMS operating system.[8] The most significant restrictions on the VAX Security Kernel implementation of ODS-2 are that all files must be preallocated and contiguous. This reduces kernel complexity by eliminating the need for dynamic file extensions. F11F implements ODS-2 files only as a flat file system.

9) VOL: The Volumes layer implements VAX Security Kernel and exchangeable volumes and provides registries of all subjects and objects. These registries are much simpler than ODS-2 directories.

10) VTerm: The Virtual Terminals layer implements virtual terminals for each virtual machine and manages the physical terminal lines. Each user may have multiple sessions connected to different virtual machines, and VTerm provides the session management functions and also implements asynchronous network lines to allow virtual machines to connect to single-access-class networks via specially dedicated terminal lines. The current version of the system has no support for higher-speed network connections.

11) VPrint: The Virtual Printers layer implements virtual printers for each virtual machine and multiplexes the real physical printers among the virtual printers. It provides top and bottom labeling, as well as trusted banner pages to delimit listings of different access classes and different VM's.

12) KI: The Kernel Interface layer implements virtual controllers for the various virtual I/O devices and the security function controller, which implements such functions as loading virtual disks into virtual drives.

13) VVAX: The Virtual VAX layer completes the virtualization process by emulating sensitive instructions, delivering exceptions and interrupts to the virtual machine, etc.

14) SSVR: The Secure Server layer implements the trusted path for security kernel, log users in and out, and provides security-related administrative functions. There is a dedicated vp2 for each terminal line to provide a Server process for each logged in user.

15) VMOS: The VMOS layer is the virtual machine's operating system.

16) USERS: The users in Fig. 3 include both the untrusted applications programs that run on top of the VMOS, and the human beings who communicate directly with the secure server via the trusted path.

H. Software Engineering Issues

A number of interesting software engineering issues arose during the development of the VAX Security Kernel. While space does not permit discussing all of them, this section highlights a few of the most significant.

1) Programming Language Choice: Perhaps the most critical software engineering issue in the VAX Security Kernel design was the choice of a programming language. From the problems that KSOS-11 [26], [36] had with its choice of compilers, it was clear that we needed high-quality compilers to develop our security kernel. While we wanted as strongly typed a language as possible, it was much more critical that the compiler correctly compile very large programs, produce high-

[7] Shadow page tables are created by a VMM, because the physical addresses in page table entries must be relocated. Shadow page tables are described in detail by Madnick and Donovan [27, sec. 9-5]. Shadow page tables are also where ring compression occurs.

[8] A brief summary of the Files-11 ODS-2 structure can be found in [35, appendices].

TABLE III
EXECUTABLE STATEMENTS PER LAYER

Layer	MACRO	PASCAL	PL/I	Total
VVAX	3371	1502	0	4873
SSVR	0	6876	330	7206
KI	10	3354	0	3364
VPRINT	0	1455	0	1455
VTERM	0	1419	0	1419
VOL	0	2553	0	2253
F11F	0	2962	0	2962
AUD	0	543	0	543
HLS	0	0	430	430
VMV	129	0	1069	1198
VMP	0	0	352	352
IOS	0	4725	0	4725
LLS	1289	13	3839	5141
HIH	815	2393	174	3382
COMMON	244	0	0	244
PMM	0	0	176	176
SVSBOO	2541	734	0	3275
VMMBOOT	55	213	430	698
VMMLIB	3021	503	1265	4789
Total	11475	29245	8065	48785

quality VAX object code, and be supported by an organization that could quickly respond to any problems we might find.

At the time the VAX Security Kernel prototype effort began, there were only a small number of systems programming languages available for the VAX architecture: BLISS-32, PL/I, PASCAL, and C. BLISS-32 was rejected because of its lack of data typing facilities. PASCAL was rejected because the V2.0 compiler that generated high-quality code was not yet available. This left PL/I and C, both of which used the same good quality code generator. We chose PL/I because of its slightly better data-typing support, because of its better support for character string manipulation, and because the first prototype developers had extensive prior experience in coding operating systems in PL/I.

We were not happy with the choice of PL/I, because its data types were not strongly enforced. When the high-quality V2.0 PASCAL compiler became available, we began writing new code for the kernel in PASCAL. PASCAL provides much stronger data-type checking than PL/I, and the VAX calling standard made interlanguage calls easy to implement.

Higher level language compilers cannot generate optimal code for all programs. Therefore we found it necessary to implement those modules that actual measurements had shown to be performance-critical in the MACRO-32 assembly language. Table III shows how much code was written in each of the languages for each layer of the kernel.[9] The table shows the number of executable source code statements (excluding comments, declarations, and white space) and per-layer and per-language totals.

In retrospect, the use of both PL/I and PASCAL has led to certain difficulties. Software engineers must be trained in both languages, and some kernel bugs have resulted from misunderstandings of how to pass parameters from one language to the other. Future security kernel developers would do well to choose one systems programming language and stick to it.

2) Coding Strategies: A number of coding strategies proved very useful in the development of the VAX Security Kernel. For example, we avoided all use of global pools within the kernel to minimize the possibility of storage channels. The maximum size of data structures is determined at system boot time (based on system-generation parameters), and memory is allocated for that maximum size during kernel initialization.

Different sections of memory within the kernel are separated by no-access guard pages to detect run-away array or string references. Unused memory is set to all ones to increase the chance of detecting the use of uninitialized variables, because zeros are less likely to generate exceptions.

The layers of the kernel are coded defensively with sanity checks to protect each layer from higher layers. If irregularities are detected, the system crashes to avoid the possibility of a security compromise. These sanity checks were devised to aid in the debugging of the kernel and do not themselves provide security assurance mechanisms. However, many of the checks remain enabled in the finished kernel to help detect any remaining bugs.

The actions of a user or a virtual machine cannot crash the kernel. They can cause error messages, exception conditions raised in the virtual machine, or in extreme cases, the halting of an offending subject.

Since the entire TCB runs in kernel mode, there are no hardware-enforced firewalls between layers. However, the layering methodology forbids lower layers from calling higher layers. To help us spot layer violations, we applied both automatic and manual techniques. Using the features of the VAX DEC/Module Management System (VAX DEC/MMS) and the VAX DEC/Code Management Systems (VAX DEC/CMS), we were able to isolate all dependencies of a layer on other layers. By visual inspection, we could immediately spot upward references. In fact during development, we did detect and fix several such occurrences.

V. HUMAN INTERFACES

High-security systems have developed a reputation for being hard to use, primarily due to their limited user interfaces. We believe that it is essential that a human interface meet the expectations of today's commercial computer users. However, we faced the same obstacles faced by other developers of high-security systems:

- Development resources are limited, and satisfying the A1 criteria takes precedence over all other efforts
- The kernel must be small and verifiable. User interface features such as a sophisticated command parser are large and often difficult to verify. Consequently, an interface built entirely on trusted code cannot match the usability of an interface built on untrusted code.

We overcame these obstacles by creating two separate command sets: the Secure Server commands, and the SECURE commands. The Secure Server commands are implemented

[9]Table III includes a number of entries that are not shown in the layer diagram in Fig. 3. These layers, COMMON, PMM, SVSBOO, VMMBOOT, and VMMLIB provide certain booting and runtime library support functions. The normal runtime libraries for the PL/I and PASCAL languages are not linked into the kernel, because they would have added a large amount of code that would need to be evaluated and placed under configuration control.

```
$ SECURE DELETE TLS:STATUS.RPT
Press SECURE ATTENTION to complete execution of this command.
User presses SECURE ATTENTION to establish a trusted path.
Delete VAX Security Kernel file TLS:STATUS.RPT
Confirmation [Yes or No]:  Y
VMM: File deleted
Resuming...
```

Fig. 5. Example of a User SECURE command.

entirely in trusted code. The administrative commands, the SECURE commands, are parsed in the VMS and ULTRIX-32 operating systems. With this approach we reduce the amount of trusted code and gain the well-developed command interfaces of these mature commercial operating systems. SECURE commands are normally only issued by the system manager, the security manager, the operators, and the auditors, although ordinary users may need to issue a few of them at times. By contrast, all users must issue some Secure Server commands to login and connect to virtual machines.

A. Secure Server Commands

The Secure Server is the user's direct interface to the kernel. A user invokes a trusted path to the Secure Server by pressing the *Secure Attention Key.* This key operates at all times and cannot be intercepted by untrusted code. We have chosen the BREAK key to be the Secure Attention Key.

The Secure Server's commands control terminal connections to virtual machines in the same way that a terminal server controls terminal connections to physical machines, using commands such as: `CONNECT`, `DISCONNECT`, `RESUME`, and `SHOW SESSIONS`. A user can create sessions with several virtual machines at different access classes and can quickly switch from one to another.

The interface for the Secure Server commands is built entirely in trusted code and offers only minimal command-line editing functions.

B. SECURE Commands

The tools for managing the system are the SECURE commands. The SECURE commands and utilities are implemented just as are other commands in the VMS and ULTRIX-32 command languages, except that they issue kernel calls to do their work. The complete set of SECURE commands and utilities is installed in the VMS operating system. A subset of the SECURE commands is offered by the ULTRIX-32 operating system.

The SECURE commands, unlike the Secure Server commands, are parsed by the VMS and ULTRIX-32 command language interpreters. The user can take advantage of such features as command-line recall and command procedures.

There are two types of SECURE commands: *VM SECURE* commands, and *User SECURE* commands. Both types of SECURE commands are issued from the VM's operating-system command level. VM SECURE commands are executed in the context of the issuing VM. User SECURE commands are submitted to the Secure Server for execution. The commands are distinguished by the type of subject, a user or a virtual machine, that holds the access class and privileges necessary to issue the command.

C. Command Confirmation

While both the User and VM SECURE commands are administrative commands, only the User SECURE commands must be trusted. For such security-relevant commands, we require A1 assurance that:

- The command was issued by a user and not by a Trojan horse in a VM
- The command received by the Secure Server is exactly the same command typed by the user, and not a command that was covertly modified by a Trojan horse
- The user who issued the command can be identified in the audit log.

Our design for the User SECURE commands provides both trust and individuality accountability, even for commands issued from an untrusted environment. Upon receipt of a valid User SECURE command, the VM instructs the user to press SECURE ATTENTION. This key invokes a trusted path between the user's terminal and the Secure Server. A SECURE ATTENTION signal can be sent to the Secure Server only by manually pressing the BREAK key. This prevents a Trojan horse from completing the execution of a User SECURE command.

To prevent a VM from spoofing the user by passing a different command from what the user typed, the Secure Server displays the action which will be taken by the command and prompts the user to approve or reject the operation. Fig. 5 is an abbreviated example of a User SECURE command issued from a VMS virtual machine. *Resuming* indicates that control of the terminal will be returned to the virtual machine.

D. SECURE Utilities

Managing the VMM security kernel requires a number of utilities. Our SECURE utilities are modeled after VMS utilities and are summarized in Table IV.

E. Reclassifying Information

Users can be permitted to change the access class of the contents of a VAX Security Kernel file or an exchangeable volume with the SECURE RECLASSIFY command. This command copies the contents of a kernel file or volume to an existing kernel file or volume labeled with a different access

TABLE IV
SECURE UTILITIES

SECURE Utility	Purpose
Authorize	Registers users and virtual machines, etc.
Register/Device	Registers I/O devices
Register/Volume	Registers disk and tape volumes.
Sysgen	Sets limits on system resources.
Crash Dump Analyzer	Provides data for determining the cause of a system crash.

class. The source and destination objects must lie within the user's access-class range. In addition, privileges are required if the reclassification downgrades the data's secrecy class or upgrades its integrity class.

Reclassification normally requires trusted inspection by the user. Inspection is required to be sure that a Trojan horse has not inserted additional information that the user did not intend to reclassify. To make inspection easier, the user can opt to print the VAX Security Kernel file or display the file on the terminal, one screen at a time. Once the complete file is printed or displayed, the user is prompted to approve the reclassification. To prevent the covert passing of information from the source file to the target file in the form of invisible escape sequences, inspected files must contain only printing characters, spaces, and form feeds. A line may not end with a space, because a trailing space would be invisible. The reclassification is terminated if any illegal character is encountered.

F. Difficulties with Command Confirmation

Command confirmation was intended to simplify the TCB and make implementation easier. While it eliminated the need for a complex parser within the TCB, it introduced a form of asynchronous communication between VM's and Server processes that was even more complex than a parser would have been. In retrospect, a menu interface to user SECURE commands combined with a mechanism for creating and checking precompiled scripts would have been simpler than the asynchronous approach and could have significantly shortened the development time and further improved the overall human factors of the system.

VI. ASSURANCE

The principal reason for building an A1 security kernel is to provide a high degree of assurance that the security features of the system actually work correctly. This section describes some of the techniques which we have used in the VAX Security Kernel to provide the necessary assurance of security, to meet both the requirements of an A1 evaluation and the requirements of real-world users. It is this integration of both A1 requirements and real-world requirements which is of particular research interest, since previous security kernels have not succeeded at integrating the A1 requirements with good performance and compatibility with large amounts of existing commercial software.

Gasser [18, p. 163] describes Honeywell's STOP kernel for the SCOMP [37] and Gemini Computers' GEMSOS [38] as commercial-grade security kernels. However, STOP does not provide software compatibility with existing operating systems, and GEMSOS to date has only been used in specialized environments. Shockley *et al.* [38] report that research is under way to provide both UNIX and MS-DOS environments for GEMSOS, but it is not clear whether those environments are yet working. If Gemini succeeds in providing both UNIX and MS-DOS environments in GEMSOS, they will have succeeded at integrating A1 requirements with real-world requirements. The VAX Security Kernel supported both the VMS and ULTRIX-32 operating systems with their layered applications by late 1989.

A. Design and Code Changes

Every change to our code underwent both design and code review, regardless of whether the code was trusted or untrusted, or whether it was a whole new layer or a bug fix. Design reviews for even the smallest fixes ensured that system-wide effects were considered. Each layer had an owner, who participated in the design review and was responsible for the quality of that layer. Each code change was reviewed both in the context of its own layer and in the contexts of its calling and called layers so as to catch interlayer problems.

Reviewers learned from the code they reviewed, as well as sharing their knowledge through review comments. Reviewers addressed readability and clarity, security, performance, elegance, and adherence to guidelines. Much like access controls, design and code guidelines were either mandatory or discretionary. Mandatory guidelines were based on prior experience in security kernel developments. Discretionary guidelines were used to avoid well-known traps in the programming language and to produce consistent, readable code. This consistency made it easier for an engineer to pick up and debug in a new area, reducing engineering costs and time.

The code review results, along with the design and test plan, were publicized for the entire group to check. This practice provided a last review of the entire change by a large audience. Code review results also served as examples from which engineers could learn good coding practices.

The development team made extensive use of VAX Notes online conferences to publicize design and coding guidelines, to discuss specific design issues, to track bug reports, and to record and publicize the results of the above-mentioned design and code reviews.

Each coding task was integrated with the current working system as soon as it was complete. This integration was constrained to always produce a working system. Continual and incremental integration avoided major unexpected failures by identifying design and/or coding errors as soon as possible.

B. Development Environment

As mentioned in Section III, we developed the VAX Security Kernel on a VAX Security Kernel system. Thus our group did its daily work on a system designed to meet A1 security requirements, using most of its features and controls.

Our VM's ran at meaningful access classes. Different versions of the kernel were maintained on different VM's to keep orthogonal tasks from impinging on each other. We also used VM's for developing and testing the untrusted code which must run in the VMS and ULTRIX-32 operating systems. We separated the roles of our own system manager and security manager, as recommended in the NCSC Evaluation Criteria [6].

The CPU and console of the development machine were kept inside a lab that only members of the VAX Security Kernel development group could enter. Within that lab, the development machine was protected by a *cage*, which consists of another room with a locked door. Physical access to both the lab and to the cage within the lab was controlled by a key-card security system. Finally, our development machine was not connected to Digital's internal computer network, so as to minimize the external threat to our development environment and our project.

C. Testing

Integrating a coding task required that a developer run a standard regression test suite. Integration occurred usually at least once a week, and as often as twice a day.[10] This regression suite consisted of two portions: *layer tests*, and *KCALL tests*. Layer tests were linked directly into the kernel, and tested layer interfaces and internal routines by calling them directly and checking their outcome. KCALL tests ran in a VM, issuing legal, illegal, and malformed requests so as to check the VM interface.

A separate suite of tests, issued via the VAX DEC/Test Manager (DTM), was run once every two weeks to test the user command interface. These tests ran for 30 h. They consisted of commands that are successful, commands that produce errors, and commands that send malformed packets to the SSVR layer. DTM checked both the results of each command and the displays it produces.

We also ran the standard VAX architecture exerciser (AXE) that verified that a particular CPU correctly implements a VAX computer. We ran AXE to test the accuracy of the VAX virtualization. AXE tests were run extensively during the development of the CPU microcode extensions and the VVAX layer. They would have been run again when the kernel reached final completion.

At the time of the cancellation, we were developing test plans for fully exercising all of the access control decisions and other security-relevant checks made by the system and for system-penetration testing. Some of these new tests would be developed from scratch, and some would be based on the formal specifications.

D. Formal Methods

The requirements for an A1 security evaluation state that a formal security policy model must be written, that a formal top-level specification (FTLS) of the system design must be written and proven to satisfy the security policy model, that the system implementation must be informally shown to be consistent with the FTLS, and that formal methods must be used in covert channel analysis of the system. The FTLS must accurately model system external interfaces, externally visible behavior, and security-relevant actions. A descriptive top-level specification (DTLS) is also required as a complete natural language description of the system.

[10] Developers, of course, ran individual tests prior to integration.

TABLE V
LINES OF FORMAL SPECIFICATIONS

Level of Specification	Lines of Ina Jo	
	Total	Transforms
TLS	650	294
FTLS	11758	8410
Total	12408	8704

We used the Formal Development Methodology (FDM) specification and verification system [39] to help meet these requirements. We wrote both our security policy model (which consists of criteria and constraints and the top-level specification (TLS) of the various transforms) and our FTLS in the FDM specification language, Ina Jo. We used the FDM interactive theorem prover (ITP) to show that the TLS obeys the policy and that the FTLS maps to the TLS. The DTLS consisted of our internal design documentation, plus some special *glue documents* that tie the DTLS and the FTLS together, particularly describing areas of the kernel which are not formally modeled in the FTLS.

Table III shows the number of executable statements in the security kernel. For comparison, Table V shows an estimate of the total number of lines of Ina Jo (comments excluded) and the number of lines of transforms (declarations excluded) required to specify that kernel. The numbers are estimates, because the FTLS was not yet complete when the project was canceled. The totals show that the number of lines of transforms are about one-sixth of the number of executable statements in the security kernel.

Formal methods do not make the system secure by themselves. Successful proof that our specifications met security policy did not guarantee that there were no lurking implementation bugs. However, the use of formal methods significantly improved the overall quality of the security kernel. When combined with the informal testing procedures, the use of formal methods improved the assurance that the security features are effective. Indeed, the very act of formally specifying the security kernel in Ina Jo detected several kernel bugs, both because of constraints imposed by proof procedures and because the process of code correspondence provides a thorough method for reviewing the TCB code and informal design specifications. The separation of duties between the software engineer and the verifier, by itself, provided valuable extra assurance, even if no proofs had ever been done.

E. Covert Channel Analysis and Countermeasures

We performed extensive analysis of covert channels throughout the development of the VAX Security Kernel. These analyses were done partially on an informal basis by engineers

closely studying the system design, and on a formal basis using a new technique for automating the Shared-Resource Matrix approach [40] with code-level flow analysis tools. It is interesting to note that the majority of the covert channels found were identified by the informal method of engineers carefully thinking about the design of the software and hardware.

The majority of storage channels were eliminated from the system during the design phase by the technique of always preallocating resources to avoid resource exhaustion channels. This preallocation had the side-benefit of significantly improving the overall robustness of the system, since it was impossible for any virtual machine to run the system out of resources.

Optimization of disk arm movements has been well-known as a source of storage channels since Schaefer *et al.*'s analysis of covert channels in KVM/370 [41]. Our analysis of disk arm optimization found that the storage channels were far more complex than had been previously thought, and that new optimizing disk controllers could make countermeasures much more difficult than those in the time of KVM/370. However, detailed analysis of the disk arm optimization storage channels revealed new techniques which could completely close such channels, without losing the throughput benefits of disk arm-motion optimizations, even in the presence of hardware controllers whose optimizations cannot be simply turned off. Furthermore, the countermeasures did not adversely affect overall system performance, but actually improved throughput in certain disk-I/O-intensive benchmarks. The details of these countermeasures are described in [42].

Timing channels proved a much more serious problem, because they tended to have higher bandwidths than the storage channels and because many of them were inherent in the underlying hardware. Since timing channels appeared numerous, difficult to enumerate, and difficult to close, we instead turned out attention to clocks. Timing channels can only be exploited in the presence of accurate clocks.

Analysis of clocks proved very fruitful, and Wray [43] developed a new technique for identifying timing channels based on *dual-clock analysis.* Essentially, a timing channel consists of two clocks, one of which must be accurate, while the Trojan horse attempting to leak information modulates the rate of the other clock.

We found that identifying all of the sources of accurate clocks was much easier than finding all of the possible timing channels in the system. If we could make the clocks less accurate, then the effective bandwidth of all timing channels in the system would be lowered. We called this new approach *fuzzy time*, because we attempted to fuzz the accuracy of all clocks available in the system.

Fuzzy time must address several different classes of clocks, including those provided by the system itself through system calls or clock interrupts, clocks provided by the time-sharing of the CPU, clocks provided by I/O devices, and external clocks which can be used to time the arrival of output on terminals, network connections, etc. The techniques for dealing with all of these clocks are very complex and are beyond the scope of this paper. They are described in detail by Hu [44]. Fuzzy time reduced the bandwidth of the worst timing channel in the VAX Security Kernel by over two orders of magnitude to well under the 10 b/s guideline of the NCSC [6]. The performance degradation due to fuzzy time was only 5–6% of CPU usage on multiprogrammed benchmarks.

F. Configuration Control

We maintained strict configuration control on many items, including design documents, trusted kernel code, test suites, user documents, and verification documents. All of our code was maintained under the VAX DEC/Code Management System (CMS) to maintain a history of each change to each module. Security reviews checked each item against the specific NCSC criteria requirements [6] it fulfills, and checked among the items for internal consistency. Items that had been reviewed were stored on a master pack which was physically protected against modification.

Our hardware, firmware, and software development tools were developed by other groups within the corporation. We reviewed hardware and firmware ECO's, prior to supporting them in the VAX Security Kernel. New versions of software development tools were tested on a stand-alone laboratory system prior to use on the kernel development machine. We used only the standard, released versions of software development tools, the same versions that had been checked out for shipment to our customers. With rare exceptions, no field-test versions were permitted on the kernel development machine.

G. Trusted Distribution

The end user of a security kernel must have some assurance that no one has tampered with or substituted counterfeit copies of the hardware and software which make up the system. Hardware and software have different trusted distribution requirements.

1) Hardware Trusted Distribution: To assure that the hardware systems would arrive at the customer's site meeting the trusted distribution criteria, we developed a security-seal program. If someone tampered with the seal, evidence would be provided of the attempted entry. A locking device would combine with the security sealing procedures to ensure a trusted shipment. Full individual accountability would be provided, including logs of the delivery.

2) Software Trusted Distribution: Installation of an A1 system involves achieving a trusted state. The steps to do this on VAX 8800 hardware are complex. The console processor software and CPU microcode must be installed and cryptographically check-summed with stand-alone software to detect any possible tampering. If a secure site loses its trusted state for any reason, they must reinstall the console software and the CPU microcode. The trusted state could be lost just by running an untrusted operating system or hardware diagnostics on the system.

Next, the trusted code is installed via untrusted code (VMS) and the result is cryptographically check-summed to verify that the untrusted code has not tampered with the trusted code. The result of the check-sum is checked against a *message authentication code* to verify correct installation. The check-

summing software is shipped separately from the rest of the software, so that a single failure of the trusted distribution system could not compromise both the check-sum program and the authentication code.

For software, there would also be an option of using trusted couriers instead of the separate delivery paths.

VII. Production-Quality Kernels

A production-quality security kernel is designed to protect and ensure the quality of real-world information. This section describes some of the differences between research and production-quality security kernels that are required to meet general user requirements, as well as to satisfy the NCSC criteria for an A1 operating system.

A. Producing the Kernel

The primary tools for creating a security kernel are compilers. Quality compilers must work for large programs, produce efficient object code, and be reliably supported. We sacrificed programming language elegance in favor of compilers with a strong track record: the VAX PASCAL and PL/I compilers. We maintained contact with the compiler developers throughout the development, and they provided much needed help to us, including occasional changes to the actual compiler code.

A second tool, a symbolic debugger/crash dump analyzer, is needed to develop and debug the system. It would also be needed by users and support personnel to diagnose problems, and by users who might wish to add functions to the kernel.

A production-quality security kernel must have adequate performance to justify its purchase in the face of other options such as multiple separate computers or periods processing. To help ensure attention to performance, we did our own development work on a VAX Security Kernel system. Performance-critical paths were written in a high-level language, and then rewritten in assembly language for speed. We added meters to find performance-critical routines, and a rudimentary performance monitor to gather statistics on CPU and I/O usage.

Bug-tracking mechanisms are needed both to satisfy NCSC configuration management guidelines and to give us a means to respond to problems on a timely basis. They also provided a means to check against our definition of quality: having no security bugs and no bug that keeps production work from running. Statistics on the number of bugs and their severity provide concrete feedback on stability.

B. Robustness

For a system to be widely used, it must be robust; that is, it must not fail very often. Robustness was an explicit nongoal of the development of the VAX Security Kernel, because we feared that adding fault tolerance would increase the complexity of the TCB, leading to a possible failure to meet A1 security requirements. Furthermore, attempting to recover from some classes of faults could actually conceal or propagate security penetrations. Therefore, the VAX Security Kernel was designed to shutdown on the discovery of any sort of fault.

Despite these apparent steps to reduce robustness, the VAX Security Kernel regularly remained up for nearly three weeks while under a heavy production load of real users! Such robustness is unheard of for field test versions of brand-new operating systems. Most new operating systems (including virtual machine monitors) consider themselves lucky to stay up for a few hours when in initial field test.

This unexpected robustness of the VAX Security Kernel comes from the strict software engineering discipline required by the A1 security criteria. Such a high level of discipline has rarely been required in industry, and this level of robustness confirms the value of the A1 development requirements far beyond the limited domain of computer security. A1 secure systems are also good bases for the development of highly available systems.

C. Documentation

A real security kernel requires extensive documentation for its users and for its system and security managers. These documents must not only meet the content requirements of the NCSC; they must also be clear and understandable to both novice and sophisticated customers. The VAX Security Kernel documentation set consists of nine manuals and a reference card. The manuals include a user's guide, guides to both system security and system management, a command reference manual, both basic and advanced programmer's manuals, an installation guide, a master index, and release notes. These manuals were written to the same quality standards as the manuals for the VMS and ULTRIX-32 operating systems.

VIII. Comparison with KVM/370

While the VAX Security Kernel superficially bears a strong resemblance to KVM/370, in that both systems create virtual machines which run at different access classes, the internal structures of the two systems are very different.

Most significantly, KVM/370 was designed as a retrofit to the existing VM/370 product, with a specific goal of leaving at least half of the original code intact [17]. As a result, KVM/370 was structured as shown in Fig. 6. The KVM/370 security kernel used a variation on self-virtualization to create a series of NKCP's (Non-Kernel Control Programs), each at a distinct mandatory access class. The NKCP's ran unmodified VM/370 code to create multiple virtual machines that then ran the CMS (Conversational Monitor System), a single-user operating system designed to run in a virtual machine. The disadvantage of this approach is that many functions executed by a virtual machine required two context switches, first into the NKCP and then into the security kernel. By comparison, VAX Security Kernel achieves a higher performance level by allowing the virtual machines to communicate directly with the security kernel. This makes the VAX Security Kernel larger than the KVM/370 security kernel, but we believe that the performance gains justify the increase in size.[11]

KVM/370 never implemented support for VMOS's that supported virtual memory. It implemented demand paging

[11] This comparison is not strictly fair to KVM/370, because the KVM/370 team was required to maintain compatibility and a large body of original code from VM/370, while the VAX Security Kernel team had the liberty of designing and coding from scratch.

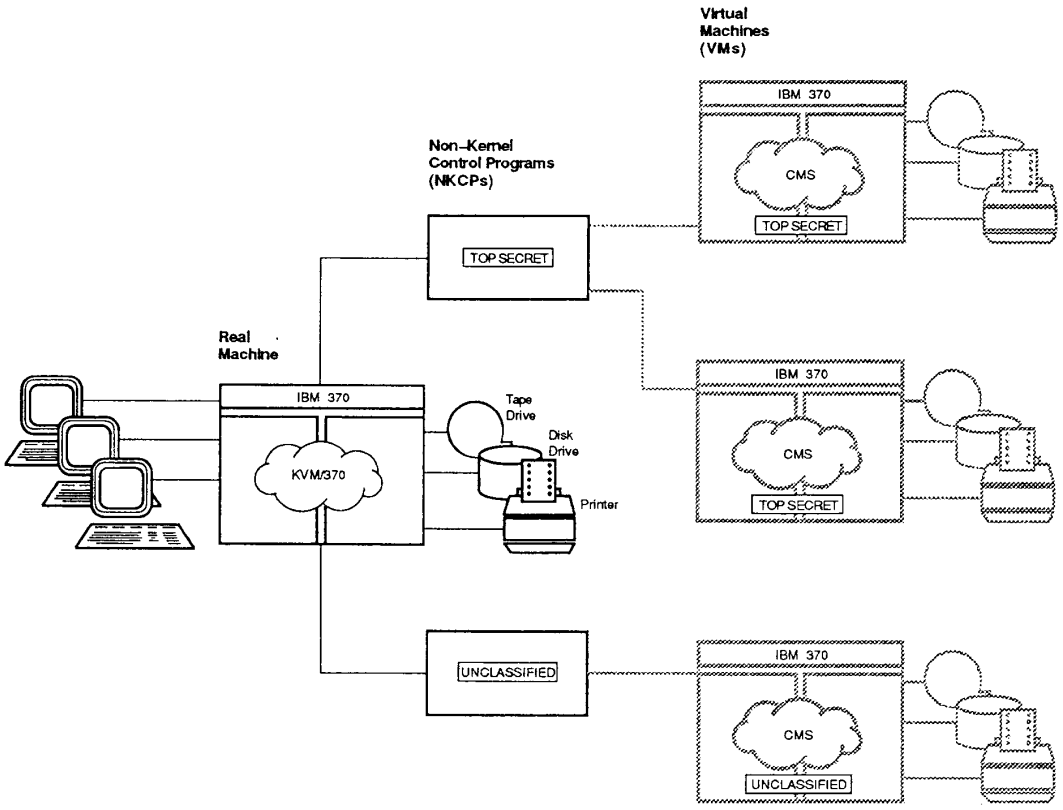

Fig. 6. KVM/370 configuration.

within its TCB. By contrast, the VAX Security Kernel leaves virtual memory support to the VMOS's. Eliminating demand paging reduces kernel complexity and improves performance at the cost of limiting the number of simultaneously active virtual machines.

Another major difference is that KVM/370 had a very limited interface for system management and security management functions. For example, new users could not be added during online operation. By contrast, the VAX Security Kernel offers a full complement of system and security management tools, such as are required in a general-purpose system (see Section V).

While performance comparisons are very tricky to make, the relative performance of the VAX Security Kernel seems better than that of KVM/370. KVM/370 reports [17] performance ranges from 10–50% of VM/370, depending on the workload. By contrast, the VAX Security Kernel exhibits performance ranges from 30–90% of VMS capacity, again depending on the workload. The KVM/370 measurements were of an untuned system, while the VAX Security Kernel measurements were of a system with a limited amount of tuning. The KVM/370 comparisons were to VM/370, itself a virtual-machine monitor with performance degradation compared to a native operating system. The VAX Security Kernel comparisons were to the native VMS operating system. KVM/370 reported a number of desirable performance optimizations that had not been done, and likewise, we know of a number of optimizations that had not yet been applied to VAX Security Kernel at the time of the cancellation.

IX. History of the Project

The idea of a virtual-machine monitor security kernel for the VAX, similar in concept to KVM/370, was first conceived by Karger and Lipner in a Mexican restaurant in Palo Alto, CA, immediately after the 1981 Symposium on Security and Privacy. An initial design study [45] concluded in 1982 that such a security kernel would be practical for the VAX architecture.

The security kernel was initially prototyped on a VAX-11/730 system. The VAX-11/730 CPU [46] was particularly attractive, because it was vertically microprogrammed and its microcode was executed from a writable control store (WCS) which could be reloaded from magnetic tape cassettes. This environment was ideal for experimenting with alternate microcode extensions to the VAX architecture, although the CPU itself was quite slow.

The VMS operating system first successfully booted in a virtual machine on 19 July 1984. That version of the security kernel was a research prototype and not a production-quality system. It was extremely slow (due in part to the choice of the

VAX-11/730, and in part to the initial software design which emphasized quick development and extensive self-checking, but not performance), and its user interface was extremely crude.

Once the VMM security kernel prototype was running reliably on the VAX-11/730 and we had accomplished some performance tuning (that improved system performance by at least an order of magnitude), we then began investigation of what a production-quality version would be like. The extensions to the VAX architecture were reimplemented on the VAX 8800 family of CPU's to provide a high-performance base for the system. Like the VAX-11/730, the VAX 8800 CPU [47] runs its microcode from a writable control store (WCS), so modifications were possible. The VAX 8800 microcode is organized horizontally, rather than vertically, and the microcode is pipelined, so the actual implementation of the extensions was much more complex than for the VAX-11/730.

Going from the research prototype to the practical version also gave us the opportunity to revisit a number of design decisions. In particular, the extensions to the VAX architecture to support virtualization were simplified, in part due to the limited availability of microcode memory in the VAX 8800. A performance study of the VAX Security Kernel prototype revealed that some of our architectural extensions did not provide the expected performance gains, while other extensions would be more valuable. For example, the prototype design included complex microcode assistance for delivering exceptions and interrupts to the virtual machines, but these microcode assists proved not to be useful and a much simpler scheme was implemented for the VAX 8800. Similarly, performance measurements of the prototype revealed that VAX operating systems (and VMS in particular) use the MTPR instruction to change their interrupt priority level (IPL) much more frequently than anyone had expected. Therefore the software was changed to optimize this particular path and microcode assistance was considered, although not implemented in this version.

The move to the production-quality kernel also marked the development of such features as user and system-management interfaces, auditing, and error logging. The prototype kernel, as a research kernel, had no need of such tools, but a real A1 system must have them so that the end users can manage and reliably run real applications on the system.

By January 1988 the kernel was sufficiently stable that some engineers could begin doing their development work on a VM. Also in January 1988 the first VAX Security Kernel was installed outside the kernel development group. That system was installed in the European ULTRIX Engineering Group in Reading, England, for porting the ULTRIX-32 operating system to a virtual machine. ULTRIX-32 first booted in a virtual machine on 15 February 1988, only two months after detailed design for the port began, and less than one month after a working VAX Security Kernel system was available for use in Reading.

By August 1988 VAX Security Kernel builds were being done on virtual machines, and by early 1989 essentially all software development work was being done on the kernel. By the Spring of 1989, the kernel was sufficiently stable that the VAX 8800 that had been running a conventional VMS time-sharing system for the kernel developers was released for other purposes. The VAX Security Kernel entered external field test in late 1989 at a number of government and aerospace industrial sites. Feedback from the external field test sites was very favorable, as the VAX Security Kernel allowed the sites to perform many multilevel secure tasks that were impossible on other systems. Performance was acceptable, except for the time needed to actually install the system. Installation time was very long, due to the extensive checks required to establish the initial secure state.

X. Cancellation

The project was formally canceled by the Digital Equipment Corporation on 1 March 1990. While the exact reasons for the cancellation remain confidential, some of the issues can be discussed here. Most importantly, the VAX Security Kernel was considered a technical success and there was significant customer demand for a product version of the security kernel and for a higher performance, nonsecure version to support VMS and ULTRIX-32 coexistence.

However, a significant fraction of the customer demand came from foreign countries who are allied with the United States under a variety of treaties. The current U.S. State Department export controls on operating systems at the B3 and A1 levels are extremely onerous and would likely have interfered with many potential sales, even to close NATO allies. These export controls do not achieve their goal of restricting access to high-security operating systems, as the technology needed to achieve such high security is primarily based on strict application of well-known software engineering practices, such as layering and information hiding. In the area of formal methods, European computer scientists are generally viewed as equal to or ahead of their U.S. counterparts. In the U.K., Data Logic has received a government contract [48] to design a secure UNIX system to meet the U.K. equivalent of an A1 rating. The National Research Council's recent report, *Computers at Risk* [49, pp. 158–159], suggests that the export controls on B3 and A1 systems are in fact discouraging U.S. industry from developing systems which employ such technology.

Certain management decisions also affected the decision to cancel. In the interest of shortening the development schedule, management chose not to implement Ethernet support in the initial versions of the VAX Security Kernel. This lack of good networking support was particularly critical, as most computer systems today require such support. The primary criticisms during the external field test came from the lack of Ethernet support.

The changes to the VAX architecture for virtualization were officially approved as ECO's (engineering change orders). However, the ECO's were not made mandatory on all new VAX processors, and most of the development groups for the VAX processors that followed the VAX 8800 did not choose to implement the ECO's. As many of those newer processors implemented their microcode in ROM's, adding the virtualization ECO's after the fact would have been difficult and expensive.

XI. Conclusions

The VAX Security Kernel is a working, production-quality VMM security kernel with performance sufficient to support a large number of time-sharing users. It is sufficiently fast and stable so that it supported its own development team. It supports vast amounts of existing user software that has been written for both the VMS and the ULTRIX-32 operating systems, and it supports both operating systems running simultaneously on the same CPU. The new covert-channel countermeasures in the VAX Security Kernel deal effectively and efficiently against entire classes of storage and timing channels that had previously been thought intractable [50].[12] As a research project in what is required to build a practical security kernel, it has been a major success.

The development of the VAX Security Kernel was long and arduous, and we learned a number of lessons during that time. Performance of a security kernel is extremely important, and getting good performance is very hard. It requires detailed analysis of what portions of the kernel are performance-critical and a willingness to redesign those portions for performance and possible recode them in assembly language or to provide microcode performance assistance.

Building the system twice—once as a research prototype and once as a study of a production-quality system—was extremely valuable. The second time around we were able to apply some of the performance lessons learned by adjusting our microcode assistance, and we developed the user and management interfaces which are essential in a real system.

One very important lesson learned is that it is not sufficient to just build a high-performance A1-secure system. That system must also support the features demanded by the user community. This means that the system must not only support many commercially available software systems (such as the VMS and ULTRIX-32 operating systems), but it must also support high-security, high-speed networking and high-security windowing systems. A high-security time-sharing system is no longer sufficient for the marketplace of the 1990's.

Developing a system to A1 standards is very hard work. Some of the A1 requirements can directly conflict with performance and usability goals, and the testing and review requirements are very time consuming. Furthermore, the export controls imposed on the A1 systems can seriously reduce the potential market for a system, making it difficult to recover the costs in achieving the A1 rating. On the other hand, the discipline required to meet A1 requirements definitely improves overall software quality and reliability.

Acknowledgment

A great many people were involved in making the VAX Security Kernel a success, and space does not permit mentioning them all here. The VAX-11/730 prototype was developed by a team of P. Karger, A. Mason, C. Kahn, and S. Thigpen. The VAX-11/730 microcode extensions were done by T. Leonard. Other engineers on the project included C. Anderson, D. Argo, M. Bokhan, D. Butchart, T. Casey, G. Champagne, E. Childs, R. Crane, D. Elfstrom, J. Ferguson, A. Gabriel-Reilly, R. Gonda, R. Govotski, E. Gugel, J. Hall, A. Hsu, W. Hu, L. Kendall, B. Kindel, C. Lo, M. McClintock, J. Melanson, L. Nasman, T. C. Pan, S. Pinkoski, T. Priborsky, J. Purretta, D. Raizen, P. Robinson, P. Sawyer, K. Seiden, G. Shapira, R. Shepardson, R. Simon, S. Stennett, H. Teng, T. Tierney, S. Troop, and M. E. Zurko. Verification work was done by C. Dermody, D. Ellis, R. Marsden, R. Modeen, D. Wittenberg, and J. Wray, with assistance from E. Cohen, T. Haley, S. Landauer, and T. Vickers Benzel of Trusted Information Systems and J. Thomas of the Aerospace Corporation. The technical writers included E. Aschkenasi, D. Bonin, E. Guth, J. Hurst, B. Laru, and P. Norton. In addition, the contributions of managers, supervisors, field test coordinators, compiler writers, members of the VMS and the European ULTRIX Engineering groups, product managers, customer service system engineers, marketing people, operations staff, our illustrator, and secretaries were all critical to the project. Finally, we must thank our team from the National Computer Security Center for their participation throughout the long development effort and the referees for their suggestions for improving this paper.

This paper presents the opinions of its authors, which are not necessarily those of Digital Equipment Corporation or the Open Software Foundation. Opinions expressed in this paper must not be construed to imply any product commitment on the part of Digital Equipment Corporation or the Open Software Foundation.

Ina Jo is a registered trademark of UNISYS Corporation. MS-DOS is a registered trademark of Microsoft Corporation. UNIX is a registered trademark of UNIX Systems Laboratory, Inc. Ethernet is a registered trademark of Xerox Corporation. The following are trademarks of Digital Equipment Corporation: DEC, DEC/CMS, DEC/MMS, DESNC, PDP-11, RSTS/E, TOPS-20, ULTRIX, VAX, VAX-11/730, VAX 8530, VAX 8550, VAX 8700, VAX 8800, VAX 8810, VAX 8820, VAX 8830, VAX 8840, VAX DEC/Test Manager, and VMS.

[12] Patent applications have been filed covering some of these countermeasures.

References

[1] R. R. Schell, "Computer security: the Achilles heel of the electronic air force?" *Air Univ. Rev.*, vol. XXX, pp. 16–33, Jan.–Feb. 1979.
[2] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, pp. 613–615, Oct. 1973.
[3] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Commun. ACM*, vol. 19, pp. 461–471, Aug. 1976.
[4] S. B. Lipner, "A comment on the confinement problem," *Operating Syst. Rev.*, vol. 9, pp. 192–196, Nov. 1975 (presented at the 5th Symp. Operating Syst. Principles, Univ. Texas, Austin, 19–21 Nov. 1975).
[5] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, pp. 236–243, May 1976.
[6] "Department of defense trusted computer system evaluation criteria," DOD, Washington, DC, DOD 5200.28-STD, Dec. 1985.
[7] S. Blotcky, K. Lynch, and S. Lipner, "SE/VMS: implementing mandatory security in VAX/VMS," in *Proc. 9th Nat. Comput. Security Conf.* (Gaithersburg, MD), 15–18 Sept. 1986, pp. 47–54.
[8] D. E. Bell and L. J. LaPadula, "Computer security model: unified exposition and Multics interpretation," MITRE Corp., Bedford, MA, HQ Electron. Syst. Div., Hanscom AFB, MA, Tech. Rep. ESD-TR-75-306, June 1975.
[9] K. J. Biba, "Integrity considerations for secure computer systems," MITRE Corp., Bedford, MA, HQ Electron. Syst. Div., Hanscom AFB, MA, Tech. Rep. ESD-TR-76-372, Apr. 1977.
[10] "Guide to VMS system security," Digital Equip. Corp., Maynard, MA, Order No. AA-LA40B-TE, June 1989.

[11] J. Whitmore *et al.*, "Design for Multics security enhancements," Honeywell Inform. Syst., Inc., HQ Electron. Syst. Div., Hanscom AFB, MA, Tech. Rep. ESD-TR-74-176, Dec. 1973.
[12] P. A. Karger, "Computer security research at Digital," in *Proc. 3rd Seminar on the DoD Comput. Security Initiative Program* (Gaithersburg, MD), 18–20 Nov. 1980, pp. E-1–E-6.
[13] T. E. Leonard, Ed., *VAX Architecture Reference Manual.* Bedford, MA: Digital, 1987.
[14] S. E. Madnick and J. J. Donovan, "Application and analysis of the virtual machine approach to information system security," in *Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Comput. Syst.* (Cambridge, MA), 26–27 Mar. 1973, pp. 210–224.
[15] R. Rhode, "Secure multilevel virtual computer systems," MITRE Corp., Bedford, MA, HQ Electron. Syst. Div., Hanscom AFB, MA, Tech. Rep. ESD-TR-74-370, Feb. 1975.
[16] B. D. Gold *et al.*, "A security retrofit of VM/370," in *AFIPS Conf. Proc.*, vol. 48, *1979 Nat. Comput. Conf.* (Montvale, NJ), 1979, pp. 335–344.
[17] B. D. Gold, R. R. Linde, and P. F. Cudney, "KVM/370 in retrospect," in *Proc. 1984 Symp. Security and Privacy* (Oakland, CA), 29 Apr.–2 May 1984, pp. 13–23.
[18] M. Gasser, *Building a Secure Computer System.* New York: Van Nostrand Reinhold, 1988.
[19] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974.
[20] P. A. Karger, T. E. Leonard, and A. H. Mason, "Computer with virtual machine mode and multiple protection rings," U.S. Patent No. 4 787 031, 22 Nov. 1988.
[21] J. S. Hall and P. T. Robinson, "Virtualizing the VAX architecture," *Comput. Architecture News*, vol. 19, pp. 380–389, May 1991 (presented at the 18th Int. Symp. Comput. Architecture Conf., Toronto, ON, Can., 27–30 May 1991).
[22] R. P. Goldberg, "Architectural principles for virtual computer systems," Ph.D. thesis, Div. Eng. and Appl. Phys., Harvard Univ., Cambridge, MA, Feb. 1973 (published as ESD-TR-73-105, HQ Electron. Syst. Div., Hanscom AFB, MA).
[23] G. J. Popek and C. S. Kline, "The PDP-11 virtual machine architecture: a case study," *Operating Syst. Rev.*, vol. 9, pp. 97–105, Nov. 1975 (presented at the 5th Symp. Operating Syst. Principles, Univ. Texas, Austin).
[24] M. D. Vahey, "A virtualizer efficiency device for virtual machines," M.S. thesis, UCLA, 1975.
[25] "A proposed interpretation of the TCSEC for virtual machine architectures," Trusted Inform. Syst., Inc., Glenwood, MD, Tech. Rep. draft, 31 Mar. 1989.
[26] T. A. Berson and G. L. Barksdale, Jr., "KSOS—development methodology for a secure operating system," in *AFIPS Conf. Proc.*, vol. 48, 1979 Nat. Comput. Conf., (Montvale, NJ), 1979, pp. 365–371.
[27] S. E. Madnick and J. J. Donovan, *Operating Systems.* New York: McGraw-Hill, 1974.
[28] P. R. Halmos, *Naive Set Theory.* New York: Van Nostrand Reinhold, 1960.
[29] E. W. Dijkstra, "The structure of the THE multiprogramming system," *Commun. ACM*, vol. 11, pp. 341–346, May 1968.
[30] P. A. Janson, "Using type extension to organize virtual memory mechanisms," Ph.D. thesis, Dept. Elect. Eng. and Comput. Science, MIT, Cambridge (published as Tech. Rep. MIT/LCS/TR-167, Lab. Comput. Sci., MIT, Sept. 1976).
[31] D. P. Reed, "Processor multiplexing in a layered operating system," S.M. thesis, Dept. Elect. Eng. and Comput. Science, MIT, Cambridge (published as Tech. Rep. MIT/LCS/TR-164, Lab. Comput. Sci., MIT, July 1976).
[32] L. A. Cox, Jr. and R. R. Schell, "The structure of a security kernel for a Z8000 multiprocessor," in *Proc. 1981 Symp. on Security and Privacy* (Oakland, CA), 27–29 Apr. 1981, pp. 124–129.
[33] D. P. Reed and R. K. Kanodia, "Synchronization with eventcounts and sequences," *Commun. ACM*, vol. 22, pp. 115–123, Feb. 1979.
[34] K. F. Seiden and J. P. Melanson, "The auditing facility for a VMM security kernel," in *Proc. 1990 IEEE Symp. Res. in Security and Privacy* (Oakland, CA), 7–9 May 1990, pp. 262–277.
[35] "VMS analyze/disk-structure utility manual," Digital Equip. Corp., Maynard, MA, Order No. AA-LA39A-TE, Apr. 1988.
[36] J. Nagle, "Update on the kernelized security operating system (KSOS)," in *Proc. 3rd Seminar on the DoD Comput. Security Initiative Program* (Gaithersburg, MD), 18–20 Nov. 1980, pp. Q-1–Q-7.
[37] L. J. Fraim, "SCOMP: A solution to the multilevel security problem," *Computer*, vol. 16, pp. 26–34, July 1983.
[38] W. R. Shockley, T. F. Tao, and M. F. Thompson, "An overview of the GEMSOS class A1 technology and application experience," in *Proc. 11th Nat. Comput. Security Conf.*, 17–20 Oct. 1988, pp. 238–245.
[39] J. Scheid, S. Anderson, R. Martin, and S. Holtzberg, "The Ina Jo specification language reference manual—release 1," System Development Corp., Santa Monica, CA, TM 6021/001/02, 1986.
[40] R. A. Kemmerer, "A practical approach to identifying storage and timing channels," in *Proc. 1982 Symp. Security and Privacy* (Oakland, CA), 26–28 Apr. 1982, pp. 66–73.
[41] M. Schaefer, B. Gold, R. Linde, and J. Scheid, "Program confinement in KVM/370," in *Proc. 1977 ACM Ann. Conf.* (Seattle, WA), 16–19 Oct. 1977, pp. 404–410.
[42] P. A. Karger and J. C. Wray, "Storage channels in disk arm optimization," in *Proc. 1991 IEEE Comput. Soc. Symp. on Res. in Security and Privacy* (Oakland, CA), 20–22 May 1991, pp. 52–61.
[43] J. C. Wray, "An analysis of covert timing channels," in *Proc. 1991 IEEE Comput. Soc. Symp. on Res. in Security and Privacy* (Oakland, CA), 20–22 May 1991, pp. 2–7.
[44] W.-M. Hu, "Reducing timing channels with fuzzy time," in *Proc. 1991 IEEE Comput. Soc. Symp. on Res. in Security and Privacy* (Oakland, CA), 20–22 May 1991, pp. 8–20.
[45] P. A. Karger, "Preliminary design of a VAX-11 virtual machine monitor security kernel," Digital Equip. Corp., Hudson, MA, Tech. Rep. DEC TR-126, 13 Jan. 1982.
[46] "VAX-11/730 central processing unit technical description," Digital Equip. Corp., Maynard, MA, EK-KA730-TD-001, May 1982.
[47] S. N. Mishra, "The VAX 8800 microarchitecture," *Digital Tech. J.*, pp. 20–33, Feb. 1987.
[48] S. Hill, "Secret service vets Unix," *Comput. Weekly*, p. 1, 26 Apr. 1990.
[49] *Computers at Risk: Safe Computing in the Information Age.* Washington, DC: Nat. Acad. Press, 1991.
[50] "Minutes of the first workshop on covert channel analysis," *Cipher: Newsletter IEEE Comput. Soc. Tech. Committee on Security and Privacy*, July 1990.

Paul A. Karger (S'71–M'74) received the S.B., S.M., and Eng. degrees from the Massachusetts Institute of Technology (MIT) in 1972, 1977, and 1980, respectively, and the Ph.D. degree from the University of Cambridge, England, in 1989.

He is a Senior Technical Consultant for the Open Software Foundation (OSF), responsible for all of OSF's work in computer security. Before joining OSF, he was a Consulting Software Engineer for the Digital Equipment Corporation, where he founded the Secure Systems Group and was the Chief Architect of the prototype security enhancements to the VMS operating system and of the prototype VAX VMM security kernel.

Mary Ellen Zurko received the S.B. degree in computer science from MIT in 1982, where she is currently working towards the S.M. degree in computer science under the sponsorship of the Digital Equipment Corporation's Graduate Engineering Education Program. She has worked in Digital's Secure Systems Group since 1986. Her research interests include distributed authorization and fusing security and usability.

Douglas W. Bonin began his career at the Digital Equipment Corporation in 1977 as a Test Technician in PDP-11 manufacturing. After working as a Technical Writer for the TOPS-20 operating system, he joined Digital's Secure Systems Group and became Project Leader for the VAX Security Kernel documentation team and author of the "VAX Security Kernel User's Guide" and the "VAX Security Kernel Guide to System Security." Currently, he is leading the development of Digital's Corporate ULTRIX Security Policy.

Andrew H. Mason received the S.B. degree in electrical engineering in 1974 and the S.M. degree in electrical engineering and computer science in 1977, both from MIT, and the S.M. degree in management from MIT's Sloan School in 1978.

He is currently a Consulting Software Engineer at the Digital Equipment Corporation. His interests include operating systems, virtual machines, computer architecture, and amateur astronomy.

Clifford E. Kahn received the B.A. degree in computer science and music in 1979 from the University of California, Santa Barbara. Since 1980 he has been with the Digital Equipment Corporation, where he early developed a command interpreter for RSTS/E. He was a main designer and implementor of the VMM Security Kernel, particularly its layer structure and its file system. He is now working on security for distributed environments.