Security is an important factor if the programs of independent and possibly error-prone or malicious users are to coexist on the same computer system. In this paper, we show that a hierarchically structured operating system, such as produced by a virtual machine system, that combines a virtual machine monitor with several independent operating systems (VMM/OS), provides substantially better software security than a conventional two-level multiprogramming operating system approach. This added protection is derived from redundant security using independent mechanisms that are inherent in the design of most VMM/OS systems. Such a system can be obtained by exploiting existing software resources.

Hierarchical approach to computer system integrity

by J. J. Donovan and S. E. Madnick

Computer systems have taken on essential roles in many organizations. As such, concern for system integrity has become increasingly important. At the same time, economies of scale and centralization of operation often make it desirable to merge many separate applications onto a single computer system. In this paper, we explore operating system software approaches to improving system integrity in a shared facility.

Operating system *integrity* may be said to exist when an operating system functions correctly under all circumstances. It is helpful to further divide the concept of integrity into two related concepts: security and reliability. By *security*, we mean the ability of the operating system to maintain control over the system resources and thereby prevent users from accidentally or maliciously modifying or accessing unauthorized information. By *reliability* we mean the ability of the operating system to continue to supply useful service in spite of all abnormal software (and most abnormal hardware) conditions—whether accidental or malicious. That is, we expect the operating system to be able to prevent "crashes."

Unlike hardware, which can have manufacturing defects, physically age, wear out, and change properties over time, software is not influenced by these phenomena. Thus, given an operating

system with complete integrity, there is no way that security or reliability flaws can creep in. The difficulty lies in producing such a system. Modern operating systems may have hundreds of thousands or even millions of instructions. The specific sequence of instructions being executed is influenced by numerous parameters including the precise timing of events. Thus, a "bug" in a program may go unnoticed for several years. In this context, the term "wearing out" is sometimes used to describe the fact that bugs are continually discovered. Note that the software is not physically wearing out but rather new circumstances keep occurring—some exposing previously unknown flaws. In addition, as a system grows older and undergoes continual modification, its quality tends to deteriorate, and it becomes more error-prone. Furthermore, the process of fixing old bugs provides an opportunity to introduce new bugs, thereby guaranteeing an unending source.

Development of software integrity

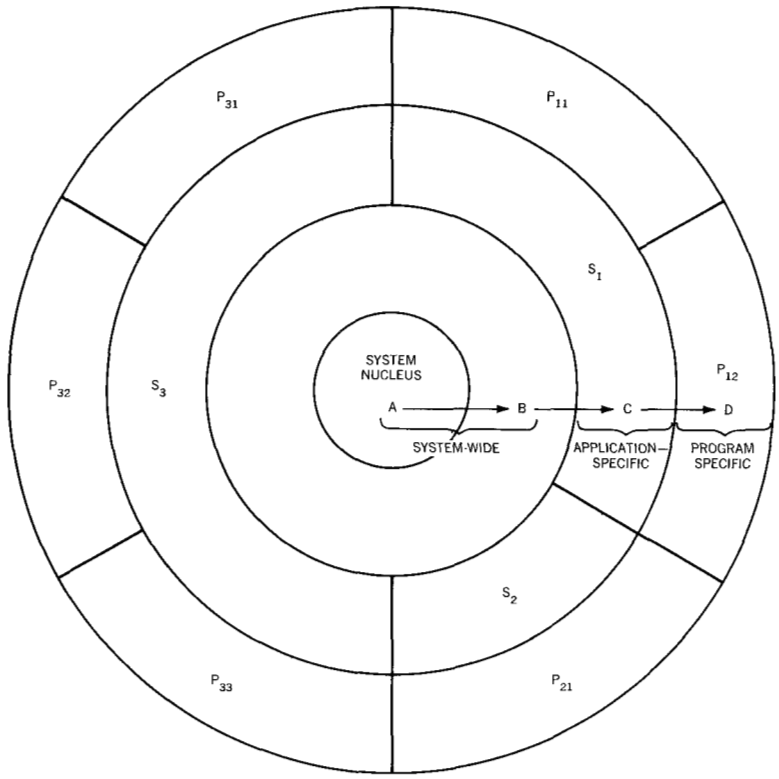
There has been considerable research and numerous attempts to develop "perfect" software ranging from hiring clever programmers, to having every program proofread by two or three programmers, to formal theorem proving.[1-3] None of these approaches have been completely successful for projects as large as a general purpose operating system. As Popek noted;[2] "Unfortunately, there currently does not exist a major software system that has withstood determined penetration efforts. We seem to have no secure systems." Although new and improved techniques will be found, the methodology for the development of perfect bug-free software is not likely to arrive for quite some time.

Under these circumstances, there are at least two things that can be done: (1) try to develop as much security and reliability into the software as possible, and (2) minimize the impact of a malfunction. In this latter connection, we would be suspicious of the integrity of a power system that would allow a simple malfunction at a Niagara Falls transformer to propagate throughout the entire Northeast. In a like manner, most software failures in an operating system would have minimal impact upon overall security and reliability if the propagation of these failures could be limited—a similar idea is involved in the use of bulkheads on ships.

Hierarchically structured operating systems

Numerous computer scientists have observed that it is possible to simplify the design of an operating system and improve its

Figure 1 Hierarchically structured operating system



integrity by a careful decomposition, separating the most critical functions from the successively less critical functions as well as separating system-wide functions from user-related functions. This approach has been termed hierarchical modularity,[4] extended machines,[5] multiple-state machines,[6] and levels of abstraction.[7] The broad concept of structured programming also encompasses this approach.

Figure 1 illustrates a hierarchically structured operating system.[5] User programs, P_{ij}, are supported by specific application subsystems, S_i. A failure in the software of region A (the system nucleus) would have wide-ranging impact upon all user programs. On the other hand, region D only impacts program P_{12}, and a failure in region C only impacts application subsystem S_1 (which in turn impacts P_{11} and P_{12}).

It is necessary to exploit certain hardware facilities to efficiently enforce the above hierarchical structure. OS/VS2 Release 2[8,9] utilizes the storage key values 0-7 of the System/370 hardware to segregate portions of the operating system, though not in a strictly hierarchical manner. The storage key hardware is avail-

able for this function because OS/VS2 Release 2 uses the Dynamic Address Translation feature to separate user address spaces. (OS/360 needed to use the storage key hardware for this function.) By exploiting the storage key hardware, the most critical privileged operating system facilities operate under key 0, less privileged system functions run under nonzero keys. In this way, the notion of "supervisor" state and "user" state is extended into several levels of supervisor state. This approach could be generalized to allow several levels of user state also.[10]

Other examples of operating systems with hardware-enforced hierarchical structure include: (1) the Honeywell MULTICS system[10] which employs sophisticated *ring structure* hardware to provide up to four levels of user state as well as four levels of supervisor state, (2) the Digital Equipment Corporation PDP-10 *three-state machine*[11] which separates kernel, supervisor, and user states, and (3) the Burroughs B6700[12,13] utilizing the *display-* or *lexical-level* hardware inherent in its stack structure.

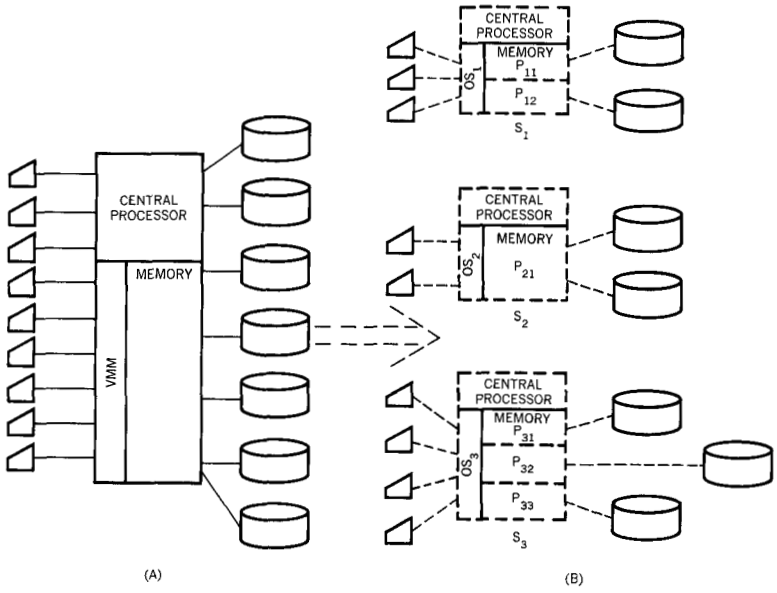
In the remainder of this paper, we will study and attempt to quantify the security and reliability attainable by means of the Virtual Machine Facility/370 (VM/370) operating system. This system has a clean three-level structure which can be easily extended to more levels. It has a simple design, as well as practical usage experience that extends back to 1966 (starting with earlier versions named CP-40 and CP-67).

Review of virtual machine concepts

Because virtual machines and their applications have been described extensively in the literature,[5,14-17] we will only briefly review the key points. A virtual machine may be defined as a replica of a real computer system simulated by a combination of a Virtual Machine Monitor (VMM) software program and appropriate hardware support. (See Goldberg[18,19] for a more precise definition.) For example, the VM/370 system enables a single System/370 to appear functionally as if it were multiple independent System/370s (i.e., multiple "virtual machines"). Thus, a VMM can make one computer system function as if it were multiple physically isolated systems as depicted in Figure 2. A VMM accomplishes this feat by controlling the multiplexing of the physical hardware resources in a manner analogous to the way that the telephone company multiplexes communications enabling separate and, hopefully, isolated conversations over the same physical communications link.

By restricting itself to the task of multiplexing and allocating the physical hardware, the VMM presents an interface that appears identical to a "bare machine." In fact, it is usually necessary to

Figure 2 Real and virtual information systems—(A) Real information system hardware, (B) Virtual information system hardware



load a user-oriented operating system into each virtual machine to provide the functions expected of modern operating systems, such as Job Control Language, command processors, data management services, and language processors. Thus, each virtual machine is controlled by a separate, and possibly different, operating system. The feasibility of this solution has been demonstrated on the VM/370 system and the earlier CP-67 and CP-40 systems. The extra VMM software and hardware do introduce additional overhead, but this overhead can be kept rather low (e.g., 10 to 20 percent). It has been reported[20] that a measurement of two DOS job streams run under control of VM/370 produced better throughput, due to increased multiprogramming, than running the same jobs serially on the same physical hardware equipment. Depending upon the precise economics and benefits of a large-scale system, the VMM approach is often preferable to the operation of the multiple physically isolated real systems.[21,22]

In addition to VM/370 and its predecessors, several other operational virtual machine systems have been developed, such as the DOS/VM of PRIME Computer, Inc.,[23] the virtual machine capability provided under the Michigan Terminal System (MTS),[24] and a virtual machine system for a modified PDP-11/45 used by UCLA for data security studies.[25]

Analysis of security and reliability in a virtual machine environment

In this section, we will analyze security and reliability in a virtual machine environment. We will show why the virtual machine approach should result in a system that is much less susceptible to security and reliability failures than a conventional two-level multiprogramming operating system. Recall that a *reliability failure* is any action of a user's program that causes the system to cease correct operation (i.e., the system "stops" or "crashes"), whereas a *security failure* is a form of reliability failure that allows one user's program to access or destroy the data or programs of another isolated user or gain control of the entire computer system. The reader may wish to refer to previous work on virtual machine security by Madnick and Donovan[26] and Attanasio[27] and on virtual machine reliability by Buzen, Chen, and Goldberg.[28]

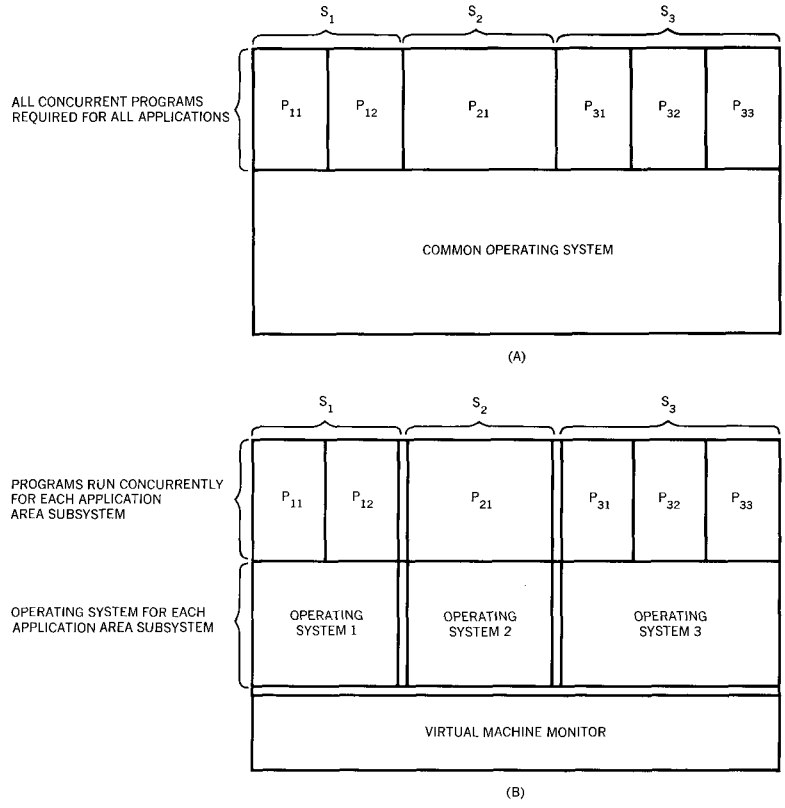
contemporary operating system environment

Most contemporary two-level operating systems, in conjunction with appropriate hardware support, provide mechanisms to prevent reliability and security failures (e.g., supervisor/problem state modes of operation). In this paper, we are only concerned about complete isolation security (i.e., no user is allowed access to any other user's information).

Under ideal circumstances, most current operating systems can provide isolation security. OS/360, for example, uses the System/360's storage key protection to insulate user programs from each other and from the operating system. The supervisor/problem state modes further prevent users from gaining control of the system. Thus, it should be possible to isolate users.

Figure 3A illustrates the coexistence of multiple programs on the same information system. Such a system is susceptible to a security violation if a single hardware or software failure were to occur. One factor contributing to the difficulty of validating entire operating systems is that user programs interface with the operating system through hundreds of parameterized entries (e.g., supervisor calls, program interruptions, I/O requests and interruptions); there is no presently known way to systematically validate the correct functioning of the operating system for all possible parameters for all entries. In fact, most systems tend to be highly vulnerable to invalid parameters. For example, a popular form of sabotage is to issue certain data-returning supervisor calls, for example, a "what time is it?" request, providing an invalid address as a parameter. The operating system, running with protection disabled and assuming that the address parameter corresponds to a user's data area, transfers the return data to that location. If the address provided actually corresponds to lo-

Figure 3 Comparison of OS and VMM/OS approaches—(A) Conventional two-level operating system approach, (B) Virtual machine approach



cations within the operating system, the system can be made to destroy or disable itself. Most "secure" systems, of course, attempt to detect this kind of error, but there are many other sabotage techniques and complete security is unlikely (see Popek[2] for additional examples).

Referring again to Figure 3A, we can see some of the factors contributing to the problem. In order to provide sufficient functionality to be effective for a large and heterogeneous collection of user programs and application subsystems, the operating system must be quite comprehensive and, thus, more vulnerable to error. In general, a single logical error in the operating system software can invalidate the entire security mechanism. Furthermore, as depicted in Figure 3A, there is no more protection between the programs of differing application subsystems (e.g., P_{12} and P_{21}) or the operating system than there is between the programs of a single application subsystem (e.g., P_{11} and P_{12}). The security of such conventional operating systems is sufficiently weak that the military has strict regulations that appear

to forbid the use of the same information system for both *secret* and *top secret* use—even though using separate systems is more costly. Similarly industrial competitors or different functions in the same company (e.g., payroll and engineering) are often reluctant to share the same computer.

level of security

Figure 3B illustrates the virtual machine approach to a physically shared system. This arrangement has numerous security advantages. If we define $P_s(\mathrm{P})$ to be the probability that a given run of program P will cause a security violation to occur, Equations 1 and 2 below would be expected to hold:

$$P_s(\mathrm{P} \mid \mathrm{OS}\ (n)) < P_s(\mathrm{P} \mid \mathrm{OS}\ (m)) \text{ for } n < m \qquad (1)$$

OS (i) refers to a conventional two-level operating system designed to support i user programs. The probability of system failure tends to increase with the load on the operating system (i.e., the number of different requests issued, the variety of functions exercised, the frequency of requests, etc.). In particular, a monoprogramming system, OS (1), tends to be much simpler and more reliable than a comprehensive multiprogramming system. Furthermore, the m-degree multiprogramming system often requires intricate alterations to support the special needs of the m users, especially if m is large. These problems have been experienced in most large-scale multiprogramming systems. These problems are diminished in a virtual machine environment because each virtual machine may run a separate operating system. Each operating system may be simpler and less error-prone than a single comprehensive all-encompassing operating system.

$$P_s(\mathrm{OS} \mid \mathrm{VMM}\ (k)) < P_s(\mathrm{P} \mid \mathrm{OS}\ (m)) \text{ for } k < m \qquad (2)$$

VMM (i) means a virtual machine monitor, VMM, supporting i virtual machines. The operating system, OS, on a particular virtual machine has the same relationship to the VMM (k) as a user's program, P, has to a conventional multiprogramming operating system, OS (m). In accordance with the same rationale as in Equation 1 above, the smaller the degree of multiprogramming (i.e., $k < m$), the smaller the probability of a security violation. Because virtual machine monitors tend to be shorter, simpler, and easier to debug than conventional multiprogramming operating systems, even when $k = m$, the VMM is less error-prone. For example, the VM/370 resident nucleus is about one-third the size of that required for MVT (multiprogramming with a variable number of tasks) with TSO. When the total privileged code of the two systems, resident and nonresident, is considered, the ratio is even more extreme.

Since the VMM is defined by the hardware specifications of the real machine, the field engineer's hardware diagnostic software can be used to check out much of the functional correctness of the VMM.

If we assume that the events represented by Equations 1 and 2 are independent, we can define the probability of a program P on one virtual machine violating the security of another concurrent program on another virtual machine as:

$$P_s(\mathrm{P} \mid \mathrm{OS}\ (n) \mid \mathrm{VMM}\ (k)) = P_s(\mathrm{P} \mid \mathrm{OS}\ (n)) \times P_s(\mathrm{OS} \mid \mathrm{VMM}\ (k)) \quad (3)$$

Based on the inequalities of Equations 1 and 2 above and the multiplicative dependency in Equation 3, we arrive at the conclusion:

$$P_s(\mathrm{P} \mid \mathrm{OS}\ (n) \mid \mathrm{VMM}\ (k)) << P_s(\mathrm{P} \mid \mathrm{OS}\ (m)) \text{ for } n, k < m \quad (4)$$

$P_s(\mathrm{P} \mid \mathrm{OS}\ (n) \mid \mathrm{VMM}\ (k))$ is the probability of the simultaneous security failure of P's operating system and the virtual machine monitor. If a single operating system's security fails, the VMM isolates this failure from the other virtual machines. If the VMM's security fails, it exposes information of other virtual machines to the operating system of one virtual machine. But, if functioning correctly, P's operating system will not take advantage of the security breach. This assumes that the designers of the individual operating systems are not in collusion with malicious users, which seems to be a reasonable hypothesis; otherwise, using the same collusion, $P_s(\mathrm{P} \mid \mathrm{OS}\ (m)) = 1$ could be attained by subverting the conventional operating system.

We are particularly concerned about the overall system security, that is, the probability that a security violation occurs due to any program in the system. This situation can be computed by:

$$\begin{aligned} P_s(\mathrm{P}_{11}, \mathrm{P}_{12}, \cdots, \mathrm{P}_{33}) &= P_s(\mathrm{P}_{11}) \times (1 - P_s(\mathrm{P}_{12})) \times \cdots \times (1 - P_s(\mathrm{P}_{33})) \\ &\quad + (1 - P_s(\mathrm{P}_{11})) \times P_s(\mathrm{P}_{12}) \times \cdots \times (1 - P_s(\mathrm{P}_{33})) \\ &\quad + \cdots \\ &\quad + P_s(\mathrm{P}_{11}) \times P_s(\mathrm{P}_{12}) \times \cdots \times P_s(\mathrm{P}_{33}) \end{aligned} \quad (5)$$

Alternatively, it can be represented as:

$$\begin{aligned} P_s(\mathrm{P}_{11}, \mathrm{P}_{12}, \cdots, \mathrm{P}_{33}) &= 1 - (1 - P_s(\mathrm{P}_{11})) \times (1 - P_s(\mathrm{P}_{12})) \\ &\quad \times \cdots \times (1 - P_s(\mathrm{P}_{33})) \end{aligned} \quad (6)$$

We note that $P_s(\mathrm{P}_{11}, \mathrm{P}_{12}, \cdots, \mathrm{P}_{33})$ is minimized when the individual Ps are minimized. The effect is accentuated due to the multiplicative nature of Equation 5. Thus, from the inequality of Equation 4, we conclude:

$$P_s(\mathrm{P}_{11}, \mathrm{P}_{12}, \cdots, \mathrm{P}_{33} \mid \mathrm{OS}\ (n) \mid \mathrm{VMM}\ (k)) <<< P_s(\mathrm{P}_{11}, \mathrm{P}_{12}, \cdots, \mathrm{P}_{33} \mid \mathrm{OS}\ (m)) \text{ for } n, k < m \quad (7)$$

That is, the security in a virtual machine environment is *very much better* than in a conventional multiprogramming operating system environment. This conclusion, as noted earlier, depends upon the probabilistic independence of the security failures. In a later section, we show that the independence condition is reasonable and applicable.

work effort

Equations 3 and 4, $P_s(\text{P} \mid \text{OS } (n) \mid \text{VMM } (k))$, are based upon the probability of two independent events occurring—a security failure in P's operating system (OS) and in the virtual machine monitor (VMM). This type of analysis is reasonable for considering the many sources of accidental reliability failures. In the case of an attempt to deliberately violate security, the penetrator would usually try to subvert the OS first and then, having taken control of the OS, attempt to subvert the VMM.

In the situation of deliberate penetration, it is useful to consider the work effort, $W_s(\text{P} \mid \text{OS } (n))$, which is a measure of the amount of work required to find a way for program P to take control of the operating system. The work effort may be in terms of mandays, number of attempts, or other such measures. Expressed in terms of work effort, Equation 3 becomes:

$$W_s(P|\text{OS } (n))|\text{VMM } (k)) = W_s(P|\text{OS } (n)) + W_s(\text{OS}|\text{VMM } (k)) \quad (8)$$

Note that unlike the probabilities of Equation 3, work efforts are additive rather than multiplicative. The overall conclusions of the preceding section also apply to a work effort analysis.

redundant security mechanisms

If the individual operating systems, OS, and the virtual machine monitor, VMM, used identical security mechanisms and algorithms, then any user action that resulted in penetration of one could also penetrate the other; that is, first take control of the OS and then, using the same technique, take control of the VMM. This penetration is logically analogous to placing one safe inside another safe—but having the same combination on both safes. To combat this danger, the OS and VMM must have *redundant security* based upon *independent mechanisms*. Similar reasoning has been applied in the specification of the PRIME modular computer system being constructed at the University of California, Berkeley. The constructors of PRIME use the term *dynamic verification* to mean "that every time a decision is made there is a consistency check performed on the decision using independent hardware and software."[29]

Table 1 illustrates redundant security mechanisms possible in a VMM/OS environment using VM/370 and OS/360 as example systems. Let us consider main memory security first. OS/360 uses the System/360-System/370 storage key hardware to isolate one user's memory area from invalid access by another user's

Table 1 Examples of redundant security mechanisms in a VMM/OS environment

Function	*VMM Mechanism (e.g., VM/370)*	*OS Mechanism (e.g., OS/360)*
Main storage security	Dynamic address translation (DAT)	Storage protection keys
Storage device security	Device address mapping	Volume label verification and data set passwords
Process allocation security	Clock comparator and time-slicing	Priority interruption (and, optionally, interval timer)

program. VM/370, on the other hand, uses the System/370 Dynamic Address Translation (DAT) hardware to provide a separate virtual memory (i.e., address space) for each virtual machine—independent of the storage keys. Thus, a malicious user would have to overwhelm both the storage key and the DAT mechanisms to violate the isolation security of another coexisting program on another virtual machine. The software algorithms, of course, used by OS/360 and VM/370 for memory security are quite different because the mechanisms that are used are so different. Thus, it is highly unlikely that they would both be susceptible to the same penetration techniques.

We find the same kind of redundant security in the area of secondary storage devices. OS/360, especially with the Resource Security System (RSS) option,[30] provides an elaborate set of mechanisms to restrict access to data sets (files). Each storage volume has a recorded label that is read by OS/360 to verify that it is the correct volume to be used (i.e., Automatic Volume Recognition, AVR). Furthermore, under RSS, the specific data sets on the volume may be individually protected by means of password codes or user authorization restrictions. VM/370, on the other hand, may have the volumes assigned to the virtual machines by the computer operator or a directory on the basis of the physical storage device address being used. Once again, the logical mapping of OS/360 is independent of the physical mapping of VM/370. These redundant security mechanisms can be found in other areas.

Although most existing VMMs were not designed specifically to provide such comprehensive isolation, they frequently include substantial redundant security mechanisms. In order to provide the needed isolation, future VMMs may be designed with increased redundant security. Using these techniques, the independence of OS and VMM penetration, assumed in Equation 3, can be attained.

Use of VM/370 to develop high-integrity systems

The various techniques described in the section on hierarchically structured operating systems for development of high-integrity systems, although encouraging, do not provide an immediate panacea for most users concerned about security and reliability. Most of the current hierarchically structured operating systems (e.g., OS/VS2 Release 2, MULTICS, etc.) are presently either experimental, in limited use, or require large or specialized hardware configurations. Even when these systems become more readily available, the user will probably still be faced with a substantial conversion effort. In this section, we explore a simpler and more immediate approach to increasing the integrity of current systems by exploiting the virtual machine concept.

The following are three example situations requiring high-integrity operations.

Departments A and B are two groups in the same company that operate separate computer facilities (e.g., a System/360, Model 30 and a System/360, Model 40). Due to increased processing loads and increased need for data interchange between departments A and B, it is recommended that they share a single larger facility (such as a System/370, Model 145). This situation is quite common. At M.I.T., for example, the Registrar's Office (processing student records) operated a separate facility from the Bursar's Office (processing payroll, etc.) both of which operate separately from the central research computer facility, which operates a System/370, Model 165.

Department A decides to add a substantial new application, such as on-line data acquisition. This can be handled by procuring an additional computer to be dedicated to this application or upgrading the present computer facility. The economy of scale usually gained by consolidating with one computer must be counterbalanced by the reliability required by the on-line application coexisting with the current batch operation.

In many situations, Examples 1 and 2 may occur at the same time forming a third example.

The vast majority of current-day computer installations have small-to-medium-size hardware configurations using fairly simple operating systems. For example, it is estimated that over 50 percent of the current System/360 and System/370 installations use some form of the Disk Operating System (DOS).

OS/VS2 Release 2, which potentially provides greater integrity, requires a minimum configuration of from 768K to 1024K bytes which is probably beyond the capabilities of most small-to-me-

dium-size configurations and, furthermore, represents a sizable conversion for an installation moving from a DOS environment.

The virtual machine approach, such as provided by VM/370, provides an attractive interim alternative. Referring to Example 1 above, departments A and B can each run a separate copy of DOS on separate virtual machines under VM/370. In addition to eliminating the need for any massive conversion, the departments are protected from each other by VM/370's security in addition to the facilities provided by DOS. In a similar manner, Example 2 can be handled by running the new on-line data acquisition application on a separate virtual machine from the current DOS batch processing work load. In fact, the on-line application may even utilize a different operating system, such as OS/360 or CMS, if that facilitates the implementation or improves integrity.

The VM/370 software insulates the application subsystem in one virtual machine from an integrity malfunction in the virtual machine of another application subsystem. This insulation is especially important when new applications are being tested concurrently with the use of existing applications.

Conclusion

In this paper, we have shown how a hierarchically structured operating system can provide substantially better software reliability and security than a conventional two-level multiprogramming operating system approach. A virtual machine facility, such as VM/370, makes it possible to convert a two-level conventional operating system into a three-level hierarchically structured operating system. Furthermore, by using redundant security mechanisms, a high degree of security is attainable.

ACKNOWLEDGMENT

The authors wish to acknowledge the suggestions offered by the reviewers of this paper. As noted by the reviewers, several of the concepts advocated by the authors, such as hierarchical systems and virtual machines, are controversial. The reader should use the references for further information on these subjects.

This paper is an extension of work originally reported in the *Proceedings of the ACM Workshop on Virtual Computer Systems* by Madnick and Donovan.[26]

This work was supported, in part, by the IBM Data Security Study, by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01), and by the Center for Information

Systems Research at the Massachusetts Institute of Technology Alfred P. Sloan School of Management.

CITED REFERENCES

1. G. Kahn, "An approach to system correctness," *Third ACM Symposium on Operating System Principles*, Stanford University, Stanford, California, 86–94 (October 1971).
2. G. J. Popek, "Protection structures," *Computer* **7**, No. 6, 22–33 (June 1974).
3. J. Scherf, *Data Security: A Comprehensive and Annotated Bibliography*, Master's Thesis, Massachusetts Institute of Technology, Alfred P. Sloan School of Management, Cambridge, Massachusetts (1973).
4. S. E. Madnick and J. W. Alsop, "A modular approach to file system design," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **34**, 1–14 (1969).
5. S. E. Madnick and J. J. Donovan, *Operating Systems*, McGraw-Hill Book Co., Inc., New York, New York (1974).
6. G. Smith, *The State of the Art of Computer Security*, IBM Study Report, Massachusetts Institute of Technology, Alfred P. Sloan School of Management, Cambridge, Massachusetts (1974).
7. E. W. Dijkstra, "The structure of the T.H.E. multiprogramming system," *Communications of the ACM* **8**, No. 9, 341–346 (May 1968).
8. *Introduction to OS/VS2 Release 2*, Form No. GC28-0661, IBM Corporation, Data Processing Division, White Plains, New York (February 1973).
9. A. L. Scherr, "Design of IBM OS/VS2 Release 2," *AFIPS Conference Proceedings, National Computer Conference*, **42**, 387–394 (1973).
10. E. I. Organick, *The MULTICS System: An Examination of its Structure*, MIT Press, Cambridge, Massachusetts (1972).
11. *PDP-10 Timesharing Handbook*, Digital Equipment Corporation, Maynard, Massachusetts (1973).
12. *B6700 Information Processing Systems*, Burroughs Corporation, Detroit, Michigan (1972).
13. E. I. Organick, *Computer System Organization—The B7500/B6700 Series*, Academic Press New York, New York (1973).
14. R. P. Goldberg, "Survey of virtual machine research," *Computer* **7**, No. 6, 34–45 (1974).
15. *IBM Virtual Machine Facility/370: Introduction*, Form No. GC20-1800, IBM Corporation, Data Processing Division, White Plains, New York (July 1972).
16. S. E. Madnick, "Time-sharing systems: Virtual machine concept vs. conventional approach," *Modern Data* **2**, No. 3, 34–36 (March 1969).
17. R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual storage and virtual machine concepts," *IBM Systems Journal* **11**, No. 2, 99–130 (1972).
18. R. P. Goldberg, "Virtual machines: Semantics and examples," *Proceedings of IEEE Computer Society Conference*, 141–142 (September 1971).
19. R. S. Goldberg, *Architectural Principles for Virtual Computer Systems*, Ph.D. dissertation, Harvard University, Cambridge, Massachusetts (November 1972).
20. C. J. Young, "Extended architecture and hypervisor performance", *Proceedings of the ACM Workshop on Virtual Computer Systems*, Cambridge, Massachusetts (1973).
21. F. R. Horton, "Virtual machine assist: Performance," *Guide 37*, Boston, Massachusetts (1973).
22. *IBM Virtual Machine Facility/370: Release 2 Planning Guide*, Form No. GC20-1814, IBM Corporation, Data Processing Division, White Plains, New York (1973).

23. *DOS/VM Reference Manual,* PRIME Computer, Inc., Framingham, Massachusetts (1974).
24. J. Hogg and P. Madderom, "The virtual machine facility—How to fake a 360," *Internal Note,* University of British Columbia and University of Michigan Computer Center (1973).
25. G. J. Popek and C. Kline, "Verifiably secure operating systems software," *AFIPS Conference Proceedings, National Computer Conference* **43,** 145–151 (1974).
26. S. E. Madnick and J. J. Donovan, "An approach to information system isolation and security in a shared facility," Working Paper 648–673, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Massachusetts (March 1973).
27. C. R. Attanasio, "Virtual machines and data security," *Proceedings of the ACM Workshop on Virtual Computer Systems,* Cambridge, Massachusetts (1973).
28. J. P. Buzen, P. P. Chen, and R. P. Goldberg, "Virtual machine techniques for improving system reliability," *Proceedings of the ACM Workshop on Virtual Computer Systems,* Cambridge, Massachusetts (1973).
29. R. S. Fabry, "Dynamic verification of operating system decisions," *Communications of the ACM* **16,** No. 11, 659–668 (November 1973).
30. *OS/MVT With Resource Security—Installation and System Programmers Guide,* Form No. GH20-1021, IBM Corporation, Data Processing Division, White Plains, New York (December 1971).

Reprint Form No. G321-5010