PSOS Revisited

Peter G. Neumann
Computer Science Laboratory
SRI International
Menlo Park CA 94025-3493 USA
neumann@CSL.sri.com
1-650-859-2375

Richard J. Feiertag
Cougaar Software, Inc.
833 North Shoreline Blvd #C200
Mountain View CA 94043 USA
rfeiertag@cougaarsoftware.com
1-650-969-6960 ext 102

Abstract

This paper provides a retrospective view of the design of SRI's Provably Secure Operating System (PSOS), a formally specified tagged-capability hierarchical system architecture. It examines PSOS in the light of what has happened in computer system developments since 1980, and assesses the relevance of the PSOS concepts in that light.

1 Historical Introduction

The design of the Provably Secure Operating System began in 1973. Subsequent to interim reports in 1975 and 1977, the final design was described in the 1980 final report [33] — although an earlier summary of PSOS was published in 1979 [13]. This paper reconsiders that effort.

PSOS was designed as a useful general-purpose operating system with demonstrable security properties. Several advanced requirements were quite far-reaching at the time, such as formally provable trustworthiness of the system and its applications, confinement of information flow, and hierarchically layered monitoring. Every operation uniformly requires possession of an appropriate hardware-generated capability, which is nonforgeable and nonbypassable. The hardware-software design of the capability-based operating system provided an early example of hierarchically layered abstraction. Strong typing was deeply embedded in the hardware-software architecture. A rigorous development methodology was used to design the system with typical operating system functionality, good performance expectations, and well-defined security properties. Because operating systems were (and still are) large and complex combinations of hardware and software prone to reliability and security vulnerabilities, PSOS was designed using a combination of disciplined engineering processes in order to provide a sound basis for claiming that the resulting system could meet its security requirements.

The approach represents an instance of good software engineering design practice, with (for example) abstraction, modularity, encapsulation and inheritance (as in the object-oriented paradigm), information hiding (including internal data and state), least privilege, and simplicity of module interfaces, specifications, and interdependencies. These techniques helped reduce the otherwise overwhelming problem of demonstrating the security of a large and complex system, reducing it to the problem of proving the security of a composition of small subsystems.

The PSOS design was strongly motivated by the formal approach that was developed early in the project — namely, the Hierarchical Development Methodology (HDM) and its formally based SPECIfication and Assertion Language (SPECIAL), which was used to precisely specify each module at each system layer, as well as interlayer state mapping functions and abstract implementations that allowed an abstraction at one layer to be formally related to lower-layer abstractions and that rigorously defined the system as a composition of modules. Several illustrative application layers were also formally specified.

Many of the characteristic design flaws still common in today's systems were essentially avoided by the methodology and the specification language. Although some simple illustrative proofs were carried out, it would be a incorrect to say that PSOS was a *proven* secure operating system. Nevertheless, the approach clearly demonstrates how properties such as security could be formally proven — in the sense that the specification could be formally consistent with the requirements, the source code could be formally consistent with the specifications, and the compiler could be proven correct as well.

Multilevel security (MLS) was shown to be implementable at an application layer rather than in a kernel, via provably cumulative trustworthiness, or alternatively could be embedded in the underlying capability mechanism. Various applications were sketched out, including (among others) a primitive but illustrative secure relational database management system, a confined-subsystem manager, secure e-mail, and a hierarchical monitoring facility.

Efficiency of execution was possible despite the hierar-

chical layering, because of the hardware-software nature of the capability architecture.

In addition to the two authors of this paper and [13], the PSOS team also included Bob Boyer, Karl Levitt, and Larry Robinson, with cameo contributions from Bob Fabry, Bernard Mont-Reynaud, Olivier Roubine, and Ashok Saxena.

After the design was completed in 1980, the PSOS project continued until 1983, encompassing the Feiertag flow analyzer for multilevel security [12] (which used the Boyer-Moore theorem prover to detect MLS design flaws), the initial development of Extended HDM (EHDM), and the Goguen-Meseguer work on noninterference [15], unwinding, and inference control [16]. However, this later work is not discussed further here, where we concentrate on the system architecture.

2 The PSOS Design

The PSOS hierarchical layering is summarized in Table 1, showing 17 system layers (0 to 16), with arbitrary extendability to higher layers for applications and user software.

Table 1: PSOS Design Layers

Layer	PSOS Abstraction or Function
17+	applications and user code (-)
16	user request interpreter *
15	user environments and name spaces *
14	user input-output *
13	procedure records *
12	user processes*, visible input-output*
11	creation and deletion of user objects*
10	directories (*)[c11]
9	extended types (*)[c11]
8	segmentation (*)[c11]
7	paging [8]
6	system processes, input-output [12]
5	primitive input/output [6]
4	arithmetic, other basic operations *
3	clocks [6]
2	interrupts [6]
1	registers (*), addressable memory [7]
0	capabilities *
Note:	
*	user-visible interface
(*)	partially visible interface
(-)	user-restrictable as desired
[c11]	creation/deletion hidden by layer 11
[i]	module hidden by layer i=6,7,8, or 12

Layering. Each layer defines one or more abstract objects or services. The abstract objects of each layer are intended to be implemented in terms of abstract and primitive objects of lower layers. The capability layer (layer 0) is the most primitive layer, with each higher layer providing additional services.

In HDM, the functionality of each of these layers is formally specified in SPECIAL, as is the composition of modules – in terms of how each module at each layer is (abstractly) implemented in terms of the lower layers. These formal specifications can then be used to prove or demonstrate desired security properties of the system. Each layer is defined as a single module or a small set of modules, which can straightforwardly be implemented in hardware or software and can be verified to be consistent with its formal specifications.

Table 2: PSOS Generic Hierarchy

Group	PSOS Abstraction	Layers
G	application/user activities	17–...
F	user abstractions	14–16
E	community abstractions	10–13
D	abstract object manager	9
C	virtual resources	6–8
B	physical resources	1–5
A	capabilities	0

It is convenient to group the layers of Table 1 together as shown in Table 2, collecting together abstractions that satisfy similar goals. At the base of the hierarchy is the capability mechanism, from which all other abstractions in the system are constructed. Above the basic capability mechanisms are all the physical resources of the system — for example, primary and secondary storage, processors, and input/output devices. From the physical resources are constructed the virtual resources. These virtual resources present a more convenient interface to the programmer than the physical resources, permit multiplexing of the physical resources in a manner largely invisible to the user, and allow the system to allocate the physical resources so as to maximize their efficient use. (Note that mapping asynchronous physical operations into convenient virtual abstractions is generally a tricky operating system challenge.) The abstract object manager provides the mechanism by which higher-layer abstractions may be created. It is possible to construct higher-layer abstractions based solely on the capability mechanism; however, the abstract object manager provides services that make construction of such abstractions easier. The top two system groups (E and F) in the generic hierarchy of Table 2 include community abstractions and user-created

abstractions. The community abstractions are intended to be used by a large group of users — for example, all users or just those within a particular organization. Such abstractions can include utility routines such as compilers and editors, and virtual resources that create and control access — such as directories. The user abstractions are those intended for use by a limited and potentially more carefully controlled group of individuals.

PSOS Capabilities. In PSOS, capabilities are the means by which all system objects are referenced and accessed. Each object in PSOS can be accessed only upon presentation of an appropriate capability to a module responsible for that object. Capabilities can be neither forged nor altered. As a consequence, capabilities provide a controllable basis for implementing the operating system and its applications, as there is no other way of accessing an object other than by presenting an appropriate capability designating that object. Each PSOS capability consists of two parts: a unique identifier (uid) and a set of access rights (represented as a Boolean array). By definition, neither part is modifiable, once a capability is created.

Unique Identifiers. PSOS generates only one original capability for each uid. Copies of a given capability can be made (wherever not restricted — see below). Therefore, a procedure or task that creates a new capability with some uid knows that the only capabilities that can have that uid must have been copied either directly or indirectly from the original. In other words, the creator of a capability with a given uid is able to retain control over the distribution of capabilities with that uid (including revocation, for example, as in Redell [39]).

Access Rights. The set of access rights in a capability for an object is interpreted by the module responsible for that object to define what operations may be performed by using that capability. For example, the access rights for a segment capability as interpreted by the segment manager indicate whether that capability may be used to write information into the designated segment, to read that information, to call that segment as a procedure, and to delete that segment. The access rights for a directory capability might indicate whether that capability may be used to add entries to the designated directory, to remove entries, and to use the capability contained in that entry. The interpretation of the access rights is constrained by a monotonicity rule, namely that the presence of a right is always more powerful than its absence. The interpretation of the access rights differs for different object types, but the monotonicity rule must always apply.

Creating Capabilities. PSOS has only two basic operations that involve actions upon capabilities (as opposed to actions based on capabilities, which is the normal mode of accessing objects), as follows.

> **c = create-capability** creates a new capability with a never-previously used uid and all access rights enabled.
>
> **cl = restrict-access(c, mask)** creates a capability with the same uid as the given capability c and with access rights that are the intersection of those of the given capability c and the given maximum (mask); that is, it creates a possibly restricted copy that can under no circumstances have any access rights that c does not possess.

The second capability operation described above appears to permit unrestricted copying of capabilities. For certain types of security policies this unrestricted copying is too liberal. For example, one may wish to give the ability to access some object to a particular user but not permit that user to pass that ability on to other users. Because simplicity of the basic capability mechanism is extremely important to achieve the goals of PSOS, any means for restricting the propagation of capabilities should not add complexity to the capability mechanism. A few access rights (only one was used by PSOS itself) are reserved as store permissions. This is the only burden placed on the capability mechanism. The interpretation of the store permissions is performed by the basic storage object manager of PSOS, namely the segment manager. Each segment in the system is designated as to whether or not it is capability store limited for each store permission. If a segment is capability store limited for a particular store permission, then it can contain capabilities only if they have that store permission. This restriction can be enforced by a simple check on all segment-modifying operations. By properly choosing the segments that are capability store limited, some very useful restrictions on the propagation of capabilities can be achieved. The restriction used in PSOS is not allowing a process to pass certain capabilities to other processes or to place these capabilities in storage locations (e.g., a directory or interprocess communication channel) accessible to other processes. More general means for restricting propagation of capabilities and for revoking the privilege granted by a capability can be implemented as subsystems of PSOS. The store permission mechanism was selected as primitive in the system because it achieves the desired result with negligible additional complexity or cost.

Tagging of Capabilities. In PSOS, capabilities can be distinguished from other entities because they are tagged

throughout the system, that is, in processors and in both primary and secondary memory, by means of a tag bit inaccessible to software or ordinary hardware instructions. In particular, there are no processor operations (other than the two capability-creating instructions noted above) that can make alterations to an existing capability or fabricate a bogus capability. Consequently, hardware can enforce the nonforgeability and unalterability of capabilities. (Capabilities are generally intended to be local to a particular system, but could be meaningful in a networked sense outside of a particular system context within a trustworthy enclave. This is discussed in [33].)

2.1 PSOS Design Principles

The design of PSOS relies heavily on rigorous engineering principles. Although some use of modularity is evident in the hierarchical layering presented above, PSOS further divides each layer into modules that implement specific abstract objects and services of each layer. In addition, each module specification enforces strong encapsulation in the design as a direct result of constraints within SPECIAL. PSOS modules provide well-defined functionality and are responsible for assuring that functionality by protecting themselves from interference (both intentional and unintended) from outside the module and from side-effects on other modules. One of the main reasons that capabilities are embedded in the lowest layers of the PSOS hierarchy is to further enforce encapsulation in implementation. Capabilities also allow modules to provide access control for the abstract objects that they implement, even for objects that are the most basic.

Defining appropriate design layering is an exercise in abstraction and information hiding, both of which are fundamental to the PSOS design. Several instances of certain types of system objects appear several times at differing levels of abstraction in the PSOS layering. For example, input-output devices appear in layers 5, 6, 12, and 14, each higher layer providing a more abstract view of I/O and hiding the specifics of the lower-layer I/O. Similarly, storage objects and object referencing each appear at several layers in the hierarchy. This use of multiple layers of abstraction to gradually build the interfaces necessary to support applications allows each layer in the PSOS hierarchy to be small and simple, and straightforward to implement and verify. It also facilitates information hiding.

PSOS represents one of the earliest uses of all four techniques of abstraction, modularity, encapsulation, and information hiding. These techniques are all in general use in the design of systems today, although they may have different names and may be implemented differently than in PSOS. In particular, encapsulation is often defined in contemporary systems by linguistic constructs and enforced by compilers rather than at runtime by the operating system. Thus, PSOS was prescient in the uniform application of these techniques within the system.

Hierarchical layering is also a fundamental principle of the PSOS design. It is important because it significantly simplifies how components can interact. Hierarchical layering dictates that modules can be dependent only on functions of modules at the same or lower layers in the hierarchy. If hierarchical layering is not imposed, then relationships between components can be arbitrarily complex — making it difficult for a developer to understand how components interact.

Use of hierarchical layering is one PSOS design principle that has not found widespread application in contemporary systems, for various reasons. Hierarchical layering requires strict control over the design of the system. Contemporary development techniques such as spiral development or rapid prototyping that promote continual incremental design modifications at all layers of the design make it difficult to maintain a strict hierarchical layering. Also, it can be difficult to maintain the integrity of the hierarchy in the implementation. Most systems provide many ways in which components can interact, and often such interactions are implicit and not explicitly stated in the code. It is easy to unintentionally introduce interactions between components that violate the hierarchical layering. In addition, client-server and agent-based systems promote unconstrained and dynamic component interactions. Maintaining a hierarchical layering in such a system appears to be self-contradictory. However, it is not necessary to totally constrain relationships in systems to obtain a hierarchical layering. It is necessary to find an appropriate layering that somehow allows localization of arbitrary relationships and constrains the relationships between the abstract layers of the system. Finding such a layering was a rewarding challenge for PSOS 30 years ago, but it would probably not be so easy to do this today for client-server and agent-based systems. Nevertheless, it can significantly reduce the errors that arise in such systems and is well worth further research.

Incidentally, the PSOS design tends to observe the security principles provided by Saltzer and Schroeder [43] and the Generally Accepted System Security Principles [36].

2.2 PSOS Implementation Considerations

The PSOS architecture effectively dispels the popular myth that hierarchical structures must be inherently inefficient. It provides a counter-example to claims by Clark [5] and Atkins [1], who suggest that layering is generally undesirable because it leads to poor performance. For example,

the PSOS layers enable user-process operations (layer 12) to execute as single capability-based hardware instructions (layer 0) whenever processes and dynamic segment linkages had been previously established, without requiring interpretation by the system process abstraction (layer 6). Similarly, a capability-based operation in user-generated software at or above layer 17 could execute hardware instructions directly. (The bottom seven layers were conceived to be implemented directly in hardware, although the hardware could also encompass higher-layer functionality as well.) Thus, repeated layers of nested interpretation are not necessarily a consequence of layered abstraction, given a suitable architecture. Furthermore, the PSOS layering is in part conceptual; some of the layers can realistically be collapsed into single layers as long as the abstraction integrity, encapsulation, and relative dependence are maintained. For example, UNIX-like and other modern operating systems tend to collapse layers 12 and higher into one layer.

3 Hierarchical Development Methodology

HDM and SPECIAL played a vital role in the design, specification, and analysis of PSOS. The methodology strongly facilitated the hierarchical design with abstraction, encapsulation, and information hiding. In particular, the specification language explicitly defines all relevant definitions, necessary assumptions, exception conditions, and desired effects, and then factors them appropriately into the proof process as preconditions, conditions, and postconditions. As one particular benefit of this approach, module execution memory residues that are not explicitly part of the module state information (that is, nonexplicit state information that transcends module invocation and that could be accessed by another module invocation) cannot be specified, as there is no way of referring to them. Similarly, buffer overflows are easily excluded from specifications. Both types of flaws are demonstrably avoidable in the corresponding code.

Abstract implementations are metaprograms in which a particular module function is specified in terms of functions and data arguments of lower layers of abstraction. Mapping functions define how higher-layer state information is represented in terms of lower-layer states. The formal approach enables the analysis (such as formal proofs) of system compositions and systems in the large.

See the 1980 final report [33], a series of HDM reports including [42], and the Robinson-Levitt paper [40] for details. A general overview of HDM is also available [41], although it uses an earlier version of the PSOS design for illustration.

Table 3 indicates a few of the properties that can be associated with some of the layers and that could be subjected to formal analysis. For example, at layer 0, a design proof of nonforgeability of capabilities relies on the fact that there are only two machine instructions that create capabilities, and that the hardware design does not allow manipulation of the tag bits. An implementation proof of that property requires reasoning about the hardware implementation itself. Similarly, a design proof that there can be no memory residues at any layer after completion of execution is almost trivial — because those residues cannot be specified. An implementation proof that there are no such residues in the code falls out of the proof of consistency of code and specification, and is meaningful at various layers (for example, relating to processes and procedures). Applying HDM's analytic abilities to the PSOS specifications and mapping functions, it is thus possible to prove properties about higher layers of abstraction that encompass the fulfillment of those properties in the design and the implementation all the way down to the bottom layer. This enables the feasibility of analyzing systems in the large, based on formal analysis of the hierarchical specifications of the modules, mappings, and design composition.

Table 3: Illustrative PSOS Properties

Layer	Property
17+	Application-relevant properties (e.g., MLS separation, secure e-mail)
16	Soundness of user types
15	Search-path flaw avoidance
12	Process isolation, input-output soundness, no process memory residues
11	No lost objects (missing capabilities)
9	Generic type soundness
8	Segmentation integrity and independence (HW)
6	Interrupts properly masked (HW)
4	Correctness of basic operations (HW)
0	Nonforgeable, nonbypassable, nonalterable capability mechanism (in hardware, MLS only if desired)

4 Background

We briefly summarize some related work.

PSOS Spinoffs. The PSOS methodology was used directly in the Ford Aerospace Kernelized Secure Operating System (KSOS) [2, 27, 37], for the specification, formal flow analysis of multilevel security (using the Feiertag

tool), and an analysis of what would be needed to carry out code proofs if desired.

The PSOS project led to subsequent parallel implementation feasibility studies by Ford Aerospace and Honeywell (also funded by NSA), which subsequently resulted in further projects by Honeywell and its outgrowth, Secure Computing Corporation, particularly with respect to SCC's dependence on pervasive typed objects, layered designs, and layered analyses. The Honeywell/SCC systems included the Secure Ada Target (SAT) [19], the LOgical Coprocessor Kernel (LOCK) [14, 18, 44], and the SideWinder firewall, each of which used typed enforcement to enhance protection. SAT pursued the PSOS observation that the MLS property could be embedded into the capability mechanism (see Boebert et al. [50]), thus gaining the benefits of enforcing a mandatory security policy. Honeywell/SCC also took a different approach to mandatory integrity than the Biba multilevel integrity policy [3] (see Boebert and Kain [4]), implementing it via enforcing strong typing. (Proofs of this enforcement of integrity are given in [19].)

Capabilities. There have of course been many other capability-based architectures, both before and after PSOS. A classification of many possible architectures is included within the taxonomy presented by Landwehr and Kain [20]. To mention just a few relevant efforts, see CAL TSS [23, 25], Lampson's paper on the confinement problem [24], Fabry [11], Redell's thesis [39], KeyKOS [38] and its reincarnation as EROS [45], Linden [26], CAP [49], Karger's thesis [21], and Gong [17]. In addition, Karger and Herbert [22] reconsider the incorporation of lattice-based mandatory security into a capability-based system. On the practical side, IBM developed System/38, a tagged capability-based system in the late 1970s. (Its successors, the AS/400 and iSeries servers are not genuinely capability based [46], but commercially more successful.)

Layered Abstraction. Earlier uses of layered abstraction include Multics [7, 8, 34] (with rings of protection, layering of system survivability and recovery, and directory hierarchies) and Dijkstra's THE system [9] (with hierarchical locking protocols that demonstrably avoided interlayer deadlocks).

Formally Based Design and Proofs. There are of course many software development methodologies. However, only a few of those are formal, and very few are mathematically based design methodologies approaching HDM. Another effort in the 1970s done in the SRI Computer Science Laboratory also made effective use of formal methods for specification and proof: SIFT, the Software Implemented Fault Tolerant system [28, 48, 30], was a sevenfold-redundant fly-by-wire system prototype developed for NASA with a systemwide five-order-of-magnitude increase in reliability over a single processor. Note that most of the functionality, expressive power, and proof capabilities associated with HDM have now been superseded by its successor, SRI's PVS [35]. See [10, 47] for recent work on verifying a compiler. Also, as a remarkable illustration of a hierarchical proof process, see the five papers in the December 1989 special issue of the *Journal of Automated Reasoning* [29], edited by J Strother Moore, which demonstrates hierarchical system verification from bottom to top, including microprocessor design verification, a verified language implementation, and a verified code generator.

Principled Design. As further background on principled architectures and development, see Neumann's report [32] for the DARPA CHATS program (Composable High-Assurance Trustworthy Systems); that report distills some of the experience gained from Multics, PSOS, and subsequent efforts, into some further guidance for future system developers, and should be of interest to open-source as well as proprietary code developers. (An earlier summary version of the report is found in [31].)

5 Retrospection

According to David Redell, Butler Lampson is credited with saying somewhen in the 1970s that "Capability systems are the way of the future, and always will be." There seems to be considerable truth in that statement, although for reasons that may have little to do with the technical merits of capabilities.

One obvious reason that capability-based addressing has not made it more widely into commercial practice involves the inherent demands of the marketplace. The inertia of having to stay within the mainstream mitigates against radical departures, and the demand for meaningfully secure systems has remained surprisingly small until recently. Our 1980 PSOS report showed that a tagged capability-based system such as PSOS could be retrofitted relatively easily onto hardware architectures that existed then, but that might be less likely today.

However, some of the concepts of PSOS and other capability architectures have found their way into wider practice, beyond the IBM System/38 noted above. In particular, digital certificates are in essence universal capabilities, and of considerable potential use in distributed and networked systems. Similarly, strong typing in computer systems has found a place in the Honeywell/SCC systems and in var-

ious programming languages. (It would be interesting to contemplate the incorporation of a capability concept into programming languages such as Java; that is currently being done in the E language.)

On the other hand, hierarchical abstraction as in PSOS has not been widely used elsewhere (network protocols are an important exception), even though it offers enormous benefits in software development — particularly for systems that must satisfy critical requirements. One of our main hopes for the future is that this paper can inspire greater awareness of those benefits in system design.

Incidentally, what Butler Lampson said about capability systems has also been said about formal methods. We believe that the formal methodology and formal specifications greatly improved the understandability and security of PSOS, and analyzability in the large. Some of the significant benefits of formal methods are being realized today in other contexts, most notably with respect to algorithms for critical systems and to hardware designs and implementations, but there is still much that can be learned from PSOS.

In the 1980s, the momentum in the DoD research community was toward supposedly small kernels that could enforce multilevel security, underlying trusted computing bases (TCBs) that took care of all the practical matters that the kernels could not. Unfortunately, the kernels tended to expand, and the TCBs became bloated when they had to support general-purpose applications. However, whether we use capabilities or access-control lists or user/group/world rights or multilevel security or other forms of access control, it is important that the underlying mechanisms be largely invisible (through programming-language and higher-layer system abstractions) and easy to use. Capability systems enforcing strong typing offer a possible alternative, by having a very simple addressing mechanism upon which secure environments can be built in a layered hierarchical way.

Taking a broader view than just capabilities, there are still many historically learned lessons that need to be conveyed to today's developers tasked with producing dependably secure and robust systems and networks. For example, there is much wisdom in the Schroeder-Saltzer principles [43] and Corbató's Turing paper [6], if suitably applied — and hopefully also in the PSOS experience summarized here. Observance of what is explored in [32] can greatly enhance the effectiveness and controllability of development efforts: well-defined requirements, highly principled system architectures that honor the most important principles (for example, achieving sound structural layering, abstraction, isolation, compartmentalization, etc.), sensible implementation that adheres to principled software engineering practice, judicious use of sound development tools that help avoid flaws, and wise administration.

References

[1] M.S. Atkins. Experiments in SR with different upcall program structures. *ACM Transactions on Computer Systems*, 6(4):365–392, November 1988.

[2] T.A. Berson and G.L. Barksdale Jr. KSOS: Development methodology for a secure operating system. In *National Computer Conference*, pages 365–371. AFIPS Conference Proceedings, 1979. Vol. 48.

[3] K.J. Biba. Integrity considerations for secure computer systems. Technical Report MTR 3153, The Mitre Corporation, Bedford, Massachusetts, June 1975. Also available from USAF Electronic Systems Division, Bedford, Massachusetts, as ESD-TR-76-372, April 1977.

[4] W.E. Boebert and R.Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth DoD/NBS Computer Security Initiative Conference*, Gaithersburg, Maryland, 1–3 October 1985.

[5] D.D. Clark. The structuring of systems using upcalls. *Operating Systems Review*, pages 171–180, 1985.

[6] F.J. Corbató. On building systems that will fail (1990 Turing Award Lecture, with a following interview by Karen Frenkel). *Communications of the ACM*, 34(9):72–90, September 1991.

[7] R.C. Daley and J.B. Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5), May 1968.

[8] R.C. Daley and P.G. Neumann. A general-purpose file system for secondary storage. In *AFIPS Conference Proceedings, Fall Joint Computer Conference*, pages 213–229. Spartan Books, November 1965.

[9] E.W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5), May 1968.

[10] A. Dold, F.W. von Henke, V. Vialard, and W. Goerigk. A mechanically verified compiling specification for a realistic compiler. Technical report UIB 03-02, Universität Ulm, Fakultät für Informatik, Ulm, Germany, December 2002. http://www.informatik.uni-ulm.de/ki/Verifix/initcomp_uib-02-03.html.

[11] R.S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[12] R.J. Feiertag. A technique for proving specifications are multilevel secure. Technical Report CSL-109, Computer Science Laboratory, SRI International, Menlo Park, California, January 1980.

[13] R.J. Feiertag and P.G. Neumann. The foundations of a Provably Secure Operating System (PSOS). In *Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979. http://www.csl.sri.com/neumann/psos.pdf.

[14] T. Fine, J.T. Haigh, R.C. O'Brien, and D.L. Toups. An overview of the LOCK FTLS. Technical report, Honeywell, 1988.

[15] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20, Oakland, California, April 1982. IEEE Computer Society.

[16] J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 75–86, Oakland, California, April 1984. IEEE Computer Society.

[17] L. Gong. A secure identity-based capability system. In *Proceedings of the 1989 Symposium on Research in Security and Privacy*, pages 56–63, Oakland, California, May 1989. IEEE Computer Society.

[18] J.T. Haigh. Top level security properties for the LOCK system. Technical report, Honeywell, 1988.

[19] J.T. Haigh and W.D. Young. Extending the noninterference model of MLS for SAT. In *Proceedings of the 1986 Symposium on Security and Privacy*, pages 232–239, Oakland, California, April 1986. IEEE Computer Society.

[20] R.Y. Kain and C.E. Landwehr. On access checking in capability-based systems. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, April 1986.

[21] P.A. Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, England, October 1988. Technical Report No. 149.

[22] P.A. Karger and A.J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 95–100, Oakland, California, April 1984. IEEE Computer Society.

[23] B.W. Lampson. On reliable and extendible operating systems. In *Proceedings of the Second NATO Conference on Techniques in Software Engineering*, Rome, Italy, 1969. NATO.

[24] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[25] B.W. Lampson and H. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–265, May 1976.

[26] T.A. Linden. Operating system structures to support security and reliable software. *ACM Computing Surveys*, 5(1), March 1973.

[27] E.J. McCauley and P.J. Drongowski. KSOS: The design of a secure operating system. In *National Computer Conference*, pages 345–353. AFIPS Conference Proceedings, 1979. Vol. 48.

[28] P.M. Melliar-Smith and R.L. Schwartz. Formal specification and verification of SIFT: A fault-tolerant flight control system. *IEEE Transactions on Computers*, C-31(7):616–630, July 1982.

[29] J S. Moore, editor. System verification. *Journal of Automated Reasoning*, 5(4):409–530, December 1989. Includes five papers by Moore, W.R. Bevier, W.A. Hunt, Jr, and W.D. Young.

[30] L. Moser, P.M. Melliar-Smith, and R. Schwartz. Design verification of SIFT. Contractor Report 4097, NASA Langley Research Center, Hampton, VA, September 1987.

[31] P.G. Neumann. Achieving principled assuredly trustworthy composable systems and networks. In *Proceedings of the DARPA Information Survivability Conference and Exhibition, DISCEX3, volume 2*, pages 182–187. DARPA and IEEE Computer Society, April 2003.

[32] P.G. Neumann. Principled assuredly trustworthy composable architectures. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 2003. Final report, SRI Project 11459, www.csl.sri.com/neumann/chats4.html; also chats4.ps and chats4.pdf.

[33] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[34] E.I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, Massachusetts, 1972.

[35] S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001. http://www.csl.sri.com/~owre.

[36] W. Ozier. GASSP: Generally Accepted Systems Security Principles. Technical report, International Information Security Foundation, June 1997. web.mit.edu/security/www/gassp1.html.

[37] T. Perrine, J. Codd, and B. Hardy. An overview of the Kernelized Secure Operating System (KSOS). In *Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference*, pages 146–160, Gaithersburg, Maryland, September 1984.

[38] S.A. Rajunas, N. Hardy, A.C. Bomberger, W.S. Frantz, and C.R. Landau. Security in KeyKOS. In *Proceedings of the 1986 IEEE Sympsium on Security and Privacy*, April 1986.

[39] D.D. Redell. *Naming and Protection in Extendible Operating Systems*. PhD thesis, University of California at Berkeley, 1974. Also MIT Project MAC TR-104.

[40] L. Robinson and K.N. Levitt. Proof techniques for hierarchically structured programs. *Communications of the ACM*, 20(4):271–283, April 1977.

[41] L. Robinson, K.N. Levitt, P.G. Neumann, and A.R. Saxena. A formal methodology for the design of operating system software. In *R. Yeh (editors), Current Trends in Programming Methodology I, Prentice-Hall, 61–110*, 1977.

[42] L. Robinson, K.N. Levitt, and B.A. Silverberg. *The HDM Handbook*. Computer Science Laboratory, SRI International, Menlo Park, California, June 1979. Three Volumes.

[43] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975. (http://www.multicians.org).

[44] O.S. Saydjari, J.M. Beckman, and J.R. Leaman. LOCKing computers securely. In *10th National Computer Security Conference, Baltimore, Maryland*, pages 129–141, 21-24 September 1987.

[45] J.S. Shapiro and N. Hardy. EROS: a principle-driven operating system from the ground up. *IEEE Software*, 19(1):26–33, January/February 2002.

[46] F.G. Soltis. *Fortress Rochester: The Inside Story of the IBM iSeries*. 29th Street Press, Loveland, Colorado, 2001.

[47] D.W.J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness*. PhD thesis, Department of Computer Science, University of York, 1998.

[48] J.H. Wensley et al. SIFT design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[49] M.V. Wilkes and R.M. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier North Holland, New York, 1979.

[50] W.D. Young, W.E. Boebert, and R.Y. Kain. Proving a computer system secure. *Scientific Honeyweller*, 6(2):18–27, July 1985. Reprinted in Tutorial: Computer and Network Security, M.D. Abrams and H.J. Podell, editors, IEEE Computer Society Press, 1987, pp. 142–157.

Peter G. Neumann and Richard J. Feiertag, Classic Papers track, Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, Nevada, 8-12 December 2003, pages 208–216.