Architectures and Formal Representations for Secure Systems

Peter G. Neumann
Computer Science Laboratory
SRI International EL-243
Menlo Park CA 94025-3493
Internet neumann@csl.sri.com
Telephone 1-415-859-2375
Fax 1-415-859-2844

2 October 1995

Final Report
SRI Project 6401
Deliverable A002

Prepared for the Department of Defense
Fort George G. Meade, Maryland 20755-6000
under Contract MDA904-94-C-6160.

Architectures and Formal Representations for Secure Systems

EXECUTIVE SUMMARY

As used in this report, the term *formal methods* encompasses mathematical and logical techniques for representing and analyzing computer systems, with the intent of increasing (1) the rigor with which a system can be defined, (2) the security and reliability that can be attained by system design and implementation, and (3) the dependability with which the requirements can be met.

This report considers the design and development of secure distributed systems and networks of such systems, and the use of formal methods for representing and analyzing those systems and networks. The report takes a broad, system-oriented view of the overall problem of attaining systems and networks that can enforce stringent security requirements with high assurance. It considers the problem from a variety of perspectives, representing a chicken-and-the-egg situation regarding practical secure systems and formal methods: Which comes first?

- How do we best achieve distributed-system architectures that can realistically result in cost-effective system developments and cost-effective operational systems?
- How do we best apply formal methods that realistically facilitate the development of systems with critical security requirements, and that provide increased levels of confidence in the security of the resulting systems?

This report draws several conclusions.

- Greater and more immediate practical benefits can be obtained if distributed system architectures are conceptually somewhat different from the more centralized views that have been the main focus in the past.
- Greater practical benefits can be attained by applying formal methods selectively in ways that are better adapted to the architectural choices.
- These benefits can be realized most effectively if the architectures and the formal methods go hand-in-hand — compatibly and mutually supportive. Neither the architectures nor the use of formal methods should totally dominate the other.

For example, the choice of system and network architectures can have an enormous influence on how effectively formal methods can be applied. Similarly, the ways in which formal methods are applied can have an enormous effect on the benefits that can be attained. In particular, being able to represent and reason about modular subsystems and their interactions (for example, through serial compositions, parallel combinations, hierarchical layerings, and network connections) is an essential ingredient of both the architectures and the formal methods.

Conventional commercial systems (that is, systems that are not multilevel secure) have generally gone through their life cycles under a fix-only-the-most-severe-bugs strategy, whereby patches are released only when absolutely necessary, and where serious security flaws generally remain ubiquitously. Customized systems seem to be even more prone to have vulnerabilities, perhaps because

they have received much less attention from friendly enthusiasts and would-be attackers. Overall, applications, systems, and network infrastructures are seriously at risk from both accidental and intentional causes.

Cryptographic techniques may superficially appear to be as strong as their key lengths, but their implementations may seriously compromise their effective strength. For example, using a two-character pass phrase to generate a 56-bit DES key is not going to result in an effective key length of 56 bits. Cryptographic implementations may also be compromised by their improper embedding into operating-system or network environments – for example, because the keys are exposed, or the unencrypted text can be accessed, or because the crypto can simply be bypassed.

Experience with past and present operating systems, networking software, and applications is not reassuring, and extrapolations into the future are also not reassuring. Bugs get fixed, but new bugs keep emerging.

Serious security vulnerabilities are continually being found (for example, see [159]), with recent widely known examples including the following cases (to name just a few):

- The demonstrated vulnerability of Netscape's security, subjected contemporaneously to exhaustive attacks of its 40-bit crypto by a French student and by a British team
- A second Netscape vulnerability involved the use of a randomization algorithm to create a crypto initialization offset (seed) based on the calendar clock and additional static information, subjected to an algorithmic approach that made the attack dramatically less than exhaustive (discovered by two Berkeley student "cypherpunks")
- A third Netscape vulnerability, in the Navigator software, which would cause the server to crash when presented with overly long numbers (discovered by the "cypherpunks")
- The vulnerability of Citibank's funds-transfer protocols, subjected to penetration attacks by a Russian intruder and 5 coconspirators (in the United States, The Netherlands and Israel), resulting from weak user authentication (reusable passwords), and resulting in 40 unauthorized transfers totalling more than $10 million, although actual losses were minimal. (These activities occurred in 1994, but became publically known only on 19 August 1995; delays and total stonewalling in reporting such incidents seem to be typical when losses and penetrations occur in financial systems.)
- An attack on the law-enforcement access field in the first version of Clipper [21]

Many additional vulnerabilities remain hidden, or known but not yet fixed. Consequently, other attacks are possible today, and new attacks will continue to be possible in the future. With respect to security, the infrastructure is weak, the operating systems and networking protocols are weak, and new systems continue to have serious flaws.

Multilevel-secure systems have traditionally relied on a centralized all-powerful kernel and a trusted computing base that together are supposed to enforce multilevel security. However, system purveyors have not been sufficiently eager to develop such systems; the evaluation process seems to have hindered progress because of its complexity and the long delays involved in the process, and because the systems are generally moving targets that undergo continuing improvements. Overall, formal methods have had only limited success in being applied to the development of widely used commercial systems. To combat these difficulties, we consider here parameterized modular architectures from which secure systems and secure networks can be readily configured and deployed, whereby the components can be analyzed in isolation and their combinations then analyzed, and whereby formal representations and analyses can become more useful in the real world.

Even ignoring the obstacles presented by the evaluation process and the difficulties experienced in applying formal methods to real systems, the system-development process has remained troglodytic. This is attributable to many causes, including the persistence of historically constrained architectures, historically limited personnel factors, and regressive system-development practices — which in some cases have been almost devoid of any serious discipline or methodology, and which have been driven more by compatibility with the past than by visions of the future. Many other factors are also relevant, including largely nontechnical issues such as perceived or real customer indifference, the lack of true national or even global disasters involving serious security breaches, and some negative effects of U.S. export control policy that are perceived by the system developers. At present, this combination of factors appears to be present considerable obstacles to both system security and the fruitful application of formal methods.

We focus here on a set of common principles that apply equally well to the establishment of system architectures and to the use of formal methods, and that can work to the advantage of both. We believe that this symbiotic approach can result in significantly greater progress toward the achievement of realistically secure systems.

The key aspects of the recommended unified approach are as follows:

- Modular, hierarchical, and distributed system structures can make architectures more implementable in terms of off-the-shelf components and at the same time can simplify the formal analyses. Concepts of good system development practice (such as the so-called field of software engineering, particularly referring to abstraction, modularity, information hiding, and some nontrivial aspects of object-oriented paradigms) can be pushed much more vigorously than they have in the past. (For example, these concepts are barely evident in existing sets of security criteria, poorly represented in university education and corporate training, and widely ignored in many commercial systems.) Parameterized components and parameterized interconnections can increase the ease with which such systems can be configured and put to productive use. Trustworthy servers are essential to this approach. In addition, some creative high-integrity strategies for accommodating modular implementations of cryptography are desirable, especially if they can improve the likelihood that good crypto can be available domestically without creating a disincentive to vendors that inhibits them from producing better domestic systems because of export-control problems that hinder their competition in the international marketplace.

- Formal methods have the highest payoffs when applied to the most critical system attributes. However, any localized properties should be related to properties of the overall architecture. As one example, security of networks and distributed systems should be relatable to the security and other properties of their components. As another example, fault tolerance and reliability are essential for the assurance of security, and critical properties relating to those attributes should also be addressed. Real-time and performance properties may also be critical, and should be included, as appropriate. The manner in which components are configured and interconnected should be rigorously represented, and the most critical properties of the most critical components should be evaluated. The formal methods should apply coherently

> but not necessarily uniformly throughout the development cycle, at various layers of abstraction (high, intermediate, and low), across distributed systems and networks — without being overly dedicated to particular implementation alternatives. These methods should apply to the combination of system components, and permit analyses of the combined systems in terms of the properties of the components. The intent of the appendix to the original draft Trusted Network Interface document [150] (the Red Book) should be reconsidered and new criteria established that represent the real complexities of distributed systems and networks, and that provide a basis for applying formal methods thereto. Because different formal methods are likely to be used in different contexts, these different methods should interoperate seamlessly (or at least without too much difficulty). The greatest payoffs tend to come up front, from having appropriate requirements and sound designs, although some involvement of formal methods in the implementation process can also be very valuable — if the requirements are appropriate and the design is sound.

Our intended audience is somewhat broad. The report is intended to be useful to its sponsoring organization, of which the Special Projects Office is engaged in carrying on both internally and funding externally research and development, whereas InfoSec groups are attempting to encourage the development of highly secure systems and networks. It is also intended for other government organizations engaged in funding research or procuring critical systems. The report is additionally intended to be useful to would-be users of formal methods — namely, system developers and customers with critical requirements — as well as researchers finding new ways to apply old techniques in formal methods, pursuing new directions, and developing tools that embody those methods and facilitate moving in those directions. It can serve as an introduction to the literature for those who are not well versed in past work, and as a review for those who are long-time participants in the application of formal methods to real systems.

The report suggests that some fundamental changes in approach can have significant benefits in the attainment of dependably secure systems, particularly with regard to how system and network architectures are conceived and how formal methods are applied. Major advances in the formal methods themselves are not generally required, although improvements in the ease of use of the corresponding tools would be helpful. In addition, integration of those methods and tools is needed, particularly as it relates to the ability to handle modular combinations of subsystems for security, and to accommodate other requirements that may be necessary for adequate security — such as fault tolerance in the infrastructure, real-time performance — or that might compromise security — such as emergency overrides.

To achieve the suggested benefits, some real changes in conceptual thinking are required; it is not sufficient just to pay lipservice to the principles discussed herein. It is easy for developers or institutions to say that these principles are indeed what they have been trying to follow all along. It is another thing to actually follow them.

The bottom line involves the need for discipline in how systems are conceived, designed, developed, modeled, analyzed, and used. That discipline should be applied to the architecture, to the development practice, and to any uses of formal methods. The best strategy is one in which the discipline is applied consistently throughout, and where the formal methods and the architectures serve to help each other.

Contents

List of Tables

Glossary

The following terms and abbreviations are used in this report:

Authentication The process of validating whether an object (user, agent, file, etc.) is indeed authentic, based on some cryptographically or hashcoding-based authentifying information (see *cryptobinding,* below). The term *authentication* is also used more generally (particularly, by others) to include the process of cryptobinding as well. Here, authentication involves primarily the validation of the cryptobinding. See Section 5.7.

BAN logic A belief logic for reasoning about cryptographic protocols. See Section 6.3.

Byzantine systems Systems that function properly despite arbitrary misbehavior on the part of some maximum number of component subsystems. See Sections 2.7 and 8.4.

Clark-Wilson integrity properties A collection of application-layer integrity properties related to well-formed transactions. See Section 2.5.

Composition The simplest form of composition involves serial hookups without feedback. Generalized forms of composition involve subsystems with mutual feedback, hierarchical layering in which a collection of modules collectively form a particular layer, and networked connections. See Section 8.

Compromisibility The possibility of being penetrated, bypassed, subverted, or otherwise subjected to attacks that cause violations of security policy. See Section 2.8.

Confidentiality An attribute of security implying that knowledge of information (for example, a hardware design, software source code, or stored or transmitted data) cannot be acquired without appropriate authorization.

Covert channel Flaws in the security of a system that permit the derivation of information that cannot otherwise be obtained — that is, information that is not directly readable. Appendix C considers only covert channels in multilevel-secure systems, although covert channels may also exist in non-MLS systems.

Cryptobinding The process of creating an object that can later be authenticated. See Section 5.7.

Dependence A logical relation implying (in varying ways) that the behavior of a function or the satisfaction of property is determined at least in part by other functions or properties. See Section 2.7.

DBMS Database management system. See Section 3.3.

DSM Distributed, Single-level-user-processor, Multilevel-secure, applied to multicomputer systems. See Sections 5.1 and C.6.

Emergence A characteristic of a property that is is not meaningful with respect to lower layers or to individual components at a particular layer, but that is meaningful with respect to a higher layer or with respect to the composition of components at the given layer. See Section 8.1.

Encapsulation The protection of an implementation from compromise. For example, the intent of crypto may be compromised (for example, by exposure of keys or of unencrypted information) if the crypto is not properly encapsulated. See Section 2.6.

Escrow binding The binding between keys that are escrowed and keys that are actually used in escrowed-key encryption. See Section 5.7.

Formal methods Mathematical and logical techniques for representing and analyzing computer systems, potentially resulting in much greater rigor, security, and dependability than otherwise attainable. See Parts Two and Three of this report.

IDES Intrusion Detection Expert System. See Appendix B.

Integrity A property that an entity is in a proper state — that is, the entity is genuine, or that the entity has not been accidentally or intentionally altered. Integrity properties include consistency, correctness, and authenticity.

ISPL Independent subsystem per level. See Section C.3.

Liveness (Lamport) A property that implies that a system will eventually do something. See Section 8.1.

MLA Multilevel availability. No entity can depend upon another entity that is not at least as trustworthy with respect to availability. See Section 2.4.

MLI Multilevel integrity. No entity can depend upon another entity that is not at least as trustworthy with respect to integrity. See Section 2.3.

MLS Multilevel security. Information must never flow from a more sensitive entity to a less sensitive entity. See Sections 2.3 and 2.9.

NIDES Next-generation Intrusion Detection Expert System, based on IDES. See Appendix B.

Nonrepudiation (strictly speaking, nonrepudiatability, which is too cumbersome a term) The credibility with which a supposedly authenticated (id)entity cannot later be subjected to a successful claim that the authentication had been erroneous, spoofed, or otherwise mishandled. See Sections 7 and 11.3.

PSOS A provably secure operating system that represents an early example of an object-oriented system hardware-software design, abstraction, layering, and formal specification. See Section 3.2.

RISSC Reduced Interfaces for Secure System Components, in which the security-critical interfaces are sharply constrained — through careful design, pervasive use of separation of duties, encapsulation, and other structural properties, as well as consistent implementation. See Section 5.

Safety (human) A property relating to a system's ability to protect human lives.

Safety (Lamport) A property that implies that if the system eventually does something, its behavior will be as specified. See Section 8.1.

SeaView A multilevel-secure database management system with no trustedness for MLS required of the DBMS. See Section 3.3.

Security Security attributes considered here include confidentiality, integrity, availability, authentication, and accountability.

TCB Trusted computing base. See Section 2.

Trusted Depending on context, either something that must be trusted because it is critical, or something that is trusted, whether or not it is trustworthy. See Section 2.6.

Trustworthiness The characteristic of having a measure of dependability with respect to the satisfaction of the stated requirements, that is, worthy of being trusted. See Section 2.6.

1 Introduction

This report summarizes the work of SRI Project 6401, which has considered the design and development of secure distributed systems and networks of such systems, and the use of formal methods for representing and analyzing those systems. Part of this work is a logical extension of the work begun under SRI Project 6402 — the final report [158] for which considered the extent to which assurance can be attained if cryptographic and cryptologic techniques are implemented in software, for each of various families of system architectures; however, that report did not attempt to elaborate on the specific techniques necessary to increase assurance of software implementations. It also considered only very narrowly the roles of cryptographic approaches to security, rather than the much broader view of security addressed here.

As used in this report, the term "formal methods" encompasses a wide variety of approaches that apply mathematical and logical techniques to the representation and analysis of computer and communication systems, in this case with particular emphasis on security. The report does not attempt to provide a complete survey of everything relating to the union of the fields of architecture, security, and formal methods. However, it does attempt to characterize the intersection of those fields by examining the underlying principles and concepts, assessing the current state of the art, and making recommendations about how significantly greater progress might be achieved.

1.1 Goals of the Project

The goals of the project are as follows:

1. To accelerate the learning curve with respect to secure distributed systems and the use of formal methods associated with those systems, for technically sophisticated individuals not familiar with the field and for newcomers to the field
2. To enhance the ease with which meaningfully secure distributed systems can be readily configured out of commercially available hardware, operating systems, and network components (to the maximum extent practical)
3. To increase the ease with which formal methods can be applied to practical secure systems
4. To enhance the ease with which secure-system products as well as distributed and networked systems can be readily evaluated
5. To increase the impact that carefully developed system architectures can have on the future — for example, aiding the development community, simplifying the implementation process, easing the evaluation process, and facilitating the incorporation of important advances into new systems

Toward those ends, this report does the following:

- Provides a historical context relating to secure-system architectures and formal methods applicable to those architectures, addressing the current state of the art, its limitations, the primary difficulties encountered in the past, and an analysis of why those limitations and difficulties arose and how they might be overcome in the future

- Examines a few representative secure-system architectures as illustrative of what has been done in the past and what might be done in the future
- Makes recommendations for specific families of system architectures that would be suitable for highly distributed and widely networked systems capable of overcoming the past difficulties
- Examines formal methods applicable to the representation and analysis of such architectures, and provides illustrations of how those methods might best be applied, for the evaluation and formal or informal analysis of models, designs, specifications, implementations, and configurations
- Identifies new directions for research and development relating to formal methods applicable to architectures for secure systems and networks, whose pursuit could substantively improve the development, configuration, and evaluation of readily available secure systems

1.2 Background

This study encompasses a range of topics, all of which are relevant to the design, development, and analysis of systems with stringent security requirements.

- *System and network architectures.* Many types of computer systems are considered here, representing a wide architectural spectrum. Several basic architectural families are examined, including trusted computing bases (TCBs) with centralized security kernels, layered TCBs, distributed systems with equivalent centralized kernels, and distributed systems with distributed TCBs or with gateways and firewalls. Of special interest from an architectural perspective are multilevel-secure distributed systems that adhere to the Randell-Rushby [210] and Proctor-Neumann concepts [192] — under which all end-user systems are single-level systems, and multilevel security is enforced by judicious use of multilevel-secure servers. That concept has subsequently been generalized into families of RISSC architectures (Reduced Interfaces for Secure System Components, Neumann-Gong, 1994 [163]), in which the security-critical interfaces are sharply constrained — through careful design, pervasive use of separation of duties, encapsulation, and other structural properties, as well as consistent implementation. The RISSC concept is considered in Section 5.
- *Uses of cryptography.* A recent SRI report [158] considers three basic types of applications of cryptography — for storage, for communication, and for integrity (including both authenticity and consistency). That report explores ten basic families of system architectures employing cryptographic techniques. It considers the relative merits of software and hardware implementations of the crypto. It also stresses the importance of integrating crypto into system architectures, a topic that is of vital importance to the present study. In particular, hierarchical structures, layering, and dependencies relating specifically to software implementations of cryptographic algorithms and protocols are fundamental to the assurance necessary for cryptographic implementations. Both that study and the present one examine the extent to which cryptographic implementations, operating systems, file servers, network servers, and other components must be trustworthy to ensure that the overall systems and networks provide adequate security with adequate assurance.

- *Evaluation criteria.* The DoD Trusted Computer Security Evaluation Criteria (TCSEC) [151] and the TCSEC Trusted Network Interpretation (TNI) [150] are the historically applicable criteria documents for evaluations of computers and their networks, but those criteria are incomplete when applied to secure distributed systems. The various newer criteria efforts from Europe and Canada and the proposed Federal Criteria and Common Criteria are somewhat less deficient, but still incomplete. Various deficiencies in the TCSEC, the European Information Technology Security Evaluation Criteria (ITSEC) [60], and the Canadian Trusted Computer Product Evaluation Criteria (CTCPEC) [34] have been discussed previously in [157]. For the TCSEC, greater emphasis is needed (for example) on applications, system integrity and availability, trusted distribution below A1, trusted recovery below B3, Trojan horse avoidance, and generalized composition. Furthermore, some unification of the differences between product evaluations and system evaluations is necessary to overcome the serious delays in evaluations.

- *Behavioral models.* Multilevel security of Bell and LaPadula [12, 13], multilevel integrity of Biba [19], and application-layer integrity of Clark-Wilson [42] are three examples of security-relevant models. They raise many issues of trust, trustworthiness, and dependence. However, such models are not sufficiently general and not complete enough to characterize all the essential properties of the various architectures that must be addressed.

- *Programming practice.* The roles of sensible programming languages (perhaps C++, for example) and sensible software-engineering techniques are very important. A poor choice of programming language or improper use of a good programming language can lead to serious security vulnerabilities. However, even a superior language can be misused.

- *Formal methods.* Formal methods for expressing requirements, specifying designs, and demonstrating consistency between specifications and requirements are fundamental to the development of dependably secure systems. Varying degrees of rigor can be applied, and each may be appropriate in certain circumstances. Specification methods and formal proof methods of interest here include SRI's HDM, EHDM, and PVS, Computational Logic Inc.'s (CLI) Gypsy and Rose, Odyssey Research's Romulus, the European Z (Zed), and many others. Criteria for usefulness of specification methods include understandability, readability, writability, and modifiability, as well as issues relating to the availability of a publication language and/or a specification language. Formal verification that code is consistent with its specification is initially less critical to the integrity of the system development process, wherein greater emphasis is placed on formal methods for establishing and analyzing specifications (for example, design validation with respect to stated requirements). Ongoing efforts at specification of security-related systems are being pursued by Secure Computing Corporation (SCC), Trusted Information Systems (TIS), the NSA evaluation group, DRA, the United Kingdom's CESG, and pursuits of Rushby's earlier work on separation kernels [203, 205]. In addition, there is significant recent progress in using formal methods in applications for which there are stringent requirements for reliability, fault tolerance, and human safety. Some of those results are discussed here, and related to what might comparably be done with respect to security.

This report documents various secure-system architectures and examines the applicability of formal methods to the analysis and representation of those and other architectures. The report contains extensive pointers to the literature, with recommendations regarding significant material for further study.

Part One: ARCHITECTURE

2 The Role of Structure in Enhancing Security

Section 2 illustrates the roles that system architecture and component structuring can play in system and network applications that must satisfy critical requirements such as security, reliability, and human safety. These concepts are seen to have significant implications on how effectively systems can be designed, implemented, evaluated, maintained, and operated. They affect the entire system life cycle.

These concepts also can have a major impact on the effectiveness with which formal methods can be applied to real systems. It is difficult to apply formal representations and formal analyses to very complex systems in the absence of cleanly delineated system structures, which are themselves facilitated by certain concepts of good software-engineering practice — such as abstraction of functionality at each external or internal interface, hierarchical layering, modular encapsulation, information hiding, separation of concerns (for example, duties or privileges), and the object-oriented paradigm (which embraces abstraction, encapsulation, polymorphism, and inheritance). This report revisits some of those concepts, in terms of what they offer to security and formal methods.

These concepts all can contribute to security, safety, reliability, and overall system verifiability, if they are used wisely. (For example, see [156, 154].) They can also be badly misused, especially when treated as panaceas or magic bullets — as in the recent craze to label everything with the "object-oriented" buzzterm. In general, there are no easy answers regarding system design, implementation, specification, design validation, and formal program verification. However, each of the techniques described here can add significantly to the structural integrity and the security of a system or network complex.

2.1 Complexity

One of the main tasks in creating a system architecture is to avoid unnecessary complexity. Analysis of the architecture can be greatly simplified by stripping away low-level details and obtaining an accurate high-level understanding. Considering that people tend toward oversimplification, Albert Einstein's familiar statement is important here: "Everything should be made as simple as possible, but no simpler."

Complexity is in the eye of the beholder. A system may seem complex to someone who does not understand it, and simple to someone who does. System designers and analysts often look for easy answers to difficult problems, beginning with system conceptualization and continuing through the development, use, and evolution of their systems, and culminating in the dismantling or replacement of those systems. But there usually are no easy answers.[1]

Many of the techniques addressed here (system structuring, software engineering, abstraction, de-

[1] For a document devoted to complexity and what can be done about it, see the ACM publication, *Managing Complexity and Modeling Reality: Strategic Issues and an Action Agenda,* D. Frailey (ed.), 1991, which includes, among other articles, Neumann's "Managing Complexity in Critical Systems" and Peter Denning's "Can Human Reality be Modeled Reliably?".

composition, composition, layering, and so on) can contribute directly to combatting complexity, and by doing so can also increase security. For example, specifications of abstractions (for requirements or functionality) and their interrelationships are common sources of security flaws. Formal methods are particularly valuable in detecting incompleteness and other errors in specifications, as discussed in Section 6.

2.2 Architecturally Interesting Systems

Multics [171] was perhaps the first operating system to make extensive use of structure within the operating system itself, as well as providing protection mechanisms that permitted multiple application layers to be separated from the operating-system layers. The concentric interprocess domains permitted the iterative implementation of policy-mechanism separation. The multilayered directory hierarchy enabled sensible directory structuring. The virtual memory implementation demanded separation of symbolic and physical addressing, where all real objects in memory are accessed by means of symbolically named virtual objects. Multics is considered in further detail in Section 3.1.

The hierarchically layered T.H.E. operating system [55] demonstrated that a strict hierarchical locking strategy could avoid deadly embraces *between* layers (although occasional deadlocks still occurred for many years *within* layers). This concept is important in preventing denials of service as well as ensuring high system availability.

Dijkstra [54, 56] and Parnas [174, 176, 177, 180, 185] each contributed significantly to the constructive structural decomposition of systems, and Parnas [44, 175, 178, 179, 181, 182, 183, 184, 186] provided definitive advances toward the formal specifications and analysis of real and complex systems, from the early 1970s to the present.

The concepts of abstraction and hierarchical layering, along with the work of Dijkstra and Parnas, were very influential in two SRI system efforts from the 1970s. These two efforts were the so-called Provably Secure Operating System (PSOS) [62, 161] and the Software-Implemented Fault-Tolerant System (SIFT) [141, 147, 250], both of which were specified according to the SRI Hierarchical Development Methodology (HDM) (Section 6.2). The object-oriented hierarchically layered PSOS design is considered further in Section 3.2. The SIFT design and prototype implementation represented a seven-processor fly-by-wire avionics computer system that was resistant to hardware faults; in the presence of faults, it was self-diagnosing and self-reconfiguring.

The PSOS design was the immediate ancestor of the sequence of developments beginning with the Honeywell Secure Ada Target (SAT), which then led to the Honeywell/Secure Computing Technology Corporation (SCTC)/Secure Computing Corporation (SCC) LOgical Coprocessor Kernel (LOCK) [39, 40, 65, 82, 83, 84, 85, 215] and the notion of trusted pipelines [23] — in which each stage in a pipeline can have its own security policy. PSOS is in several senses an early system adhering to the philosophy of Reduced Interfaces for Secure System Components (RISSC). PSOS is considered in further detail in Section 3.

A rather different lineage that also adheres to the RISSC philosophy begins with the Rushby-Randell Newcastle Distributed Secure System DSS [210], which involved a collection of single-level Unix[2] systems linked via trusted network interface units (TNIUs). A descendent of DSS is now also commercially available (DRA) in the United Kingdom. The Gemini GEMSOS and Trusted

[2] All product and company names mentioned in this report are trademarks of their respective holders.

Network Processor (GTNP) multilevel-security gateway efforts also bear considerable kinship with Multics and PSOS, and with the RISSC concept. Both LOCK and GTNP are operational systems with potential usefulness in B3/A1-rated applications.

A TCSEC-orthodox system lineage [151] includes end-user systems with multilevel-secure kernels and associated trusted computing bases (TCBs), such as the kernelized Unix system KSOS [16, 133], the MITRE [221] and UCLA Secure Unix kernels, and the kernelized virtual-machine system KVM for IBM's VM [72, 214]. Rushby's separation kernels [203, 205] represents a minimalized lower-layer version of that approach, in which certain isolation properties are enforced, and on which other properties such as multilevel security can be implemented.

Unix has led to many variants, such as CMU's Mach, TMach, TIS's DTMach (see [164] for an analysis of secure reliable distributed systems), Synergy, and Secure Computing Corporation's DTOS.

Synergy [216] is also of considerable interest (although it has very little available documentation). Synergy follows some of the careful abstraction, layering, and policy-mechanism separation found in PSOS. Its layered architecture incorporates, from the lowest layer upward, (0) an extended Mach microkernel, (1) a policy-enforcing security server, (2) a collection of functional servers (for files, devices, names, authentication, networking, keys, crypto, and audit), (3) operating system servers, and (4) applications, trusted or otherwise.

DTOS unifies the PSOS/LOCK object-oriented multiple-policy lineage with Unix-style operating systems based on a Mach kernel, and provides Mach compatibility as well. Security policies are implemented in a security-policy server, which is external to the Mach kernel — thus making it easier to support multiple policies. DTOS provides fine-granularity access control, and experimentally supports both a LOCK-style type-enforcement policy and a Chinese Wall policy.

Blacker [64, 249] provides a historically interesting approach to networked systems with end-to-end encryption, although its architecture is unwieldy by modern standards.[3]

More recent efforts such as the Digital Distributed System Security Architecture [68] (DDSSA) and Kerberos [14, 155, 236] are fundamental steps toward modern secure distributed systems, employing Needham-Schroeder-like certificates [152, 153, 172] for trusted authentication. (DDSSA is somewhat more secure than Kerberos, although both are susceptible to operating-system attacks.) See [223] for a summary of these and related protocols.

End-to-end encryption for electronic mail is provided by various forms of Privacy Enhanced Mail (PEM [108, 109, 110, 121]), including PGP [66, 224, 256]. In each case, the security depends on the operating system or other platform on which the mail software exists. More general applications of crypto are considered by Schneier [223].

Many other references could be cited for historical interest and for further approaches. For additional early references on security kernels, see [5, 219, 220]. Earlier references on security guards include [17, 35, 45, 78]. For futher background on secure distributed systems, see [8, 36, 58, 67, 194, 203, 211, 244], and for useful further works on distributed systems see [148, 149, 239, 240, 241]. For additional references on capability-based architectures, see [103, 104, 193].

In this report, we have not attempted to analyze efforts that are still in flux. However, anyone

[3] The Blacker design experience has generally been restricted; a paper on the underlying trade-offs [249] won the best-paper award at the 1988 IEEE Security and Privacy Symposium, but could not be published in the proceedings until 1992.

seriously interested in pursuing the concepts discussed in this report should attempt to evaluate those other efforts in the light of the concepts discussed herein.

2.3 Multilevel Security and Integrity

Intuitively, the basic multilevel-security (MLS) property is that information must never flow from a more sensitive entity (for example, file, database (DB), system, or user terminal) to a less sensitive entity. Multilevel integrity (MLI) is a strict mathematical dual of multilevel security. We oversimplify slightly for the moment, and elaborate on these two properties as follows.

- **MLS.** For reading, the multilevel-security flow property implies that the security level of the recipient must be at least as high as the security level of the object being read, for otherwise there would be a downward flow of information. For writing, the flow property implies that the security level of the information source must be at most the security level of the destination, for otherwise there would be a downward flow of information.
- **MLI.** Intuitively, the basic MLI property is that no entity (for example, process, program, or data) can depend upon another entity whose integrity level is not at least as trustworthy with respect to integrity.

The MLS/MLI afficianado is accustomed to lattice policies in which levels and categories are associated with each entity or user, and where flow and dependence are defined in terms of lattice inclusion. However, that refinement is not necessary for an understanding of the concepts here, and is obtained (if properties are stated appropriately) simply by replacing the linear relations with corresponding lattice set-inclusion relations — for example, replacing $\geq$ by $\supseteq$ (sometimes pronounced "dominates").

2.4 Multilevel Availability

In many systems, there is a need for priorities relating to the allocation of scarce resources. Furthermore, when operating under emergency situations, it may be necessary to alter the priorities or the way in which they are interpreted. Neumann et al. [164] have proposed a concept of *multilevel availability* (MLA) that is somewhat analogous to MLS and MLI. Each resource has associated with it a range of priorities within which it may operate, and each client is allocated (dynamically or statically, according to circumstances) a priority instance; the allocated priority could depend on the client's identity or role, the intended operation, the objects on which the operation is acting, and the destination of the results, although it could be assigned a preallocated value, as is the case with statically allocated MLS labels. The precedences defined by the priorities would then be enforced by a multilevel-availability TCB or server.

A mandatory multilevel-availability policy could also be viewed intuitively as a special kind of mandatory multilevel integrity: with respect to levels of trustedness for availability, there shall be no dependence on components that are less trustworthy with respect to availability. Note that trustedness for availability and trustedness for integrity need not be equivalent. A high-integrity component could attempt to guarantee correct results, but might not be able to control how long the task might take; indeed, the task might never complete (in the case of the ultimate denial of service for that component). If multilevel availability labels can be mapped onto integrity labels, then the MLA labels would become superfluous (particularly if the mapping is one-to-one); otherwise, they would be independently meaningful. However, in the case of independently chosen labels for MLS, MLI, and MLA, it is necessary to ensure that the three sets of labels are not mutually in conflict, to the extent that an intended operation might be impossible to perform. Discretionary availability policies might also be defined in addition to the mandatory MLA policy. However, for present purposes, we consider only the priority structuring that would result from a simple mandatory multilevel availability policy in which there is no reliance on less available resources, with the possibility of trusted exceptions similar to what is necessary to make MLS and MLI realistically workable.

Multilevel availability can be used to ensure that high-priority clients have sufficient precedence over lower-priority clients. In this way, lower-priority clients can be prevented from causing denials of services that affect higher-priority clients.

2.5 Software-Engineering Techniques

The software-engineering concepts noted at the beginning of Section 2 are fundamental to the development of well-structured computer and communication systems. We next consider the Clark–Wilson integrity properties, restated in Table 1 in terms more familiar to software engineers than the original representation. (This table is derived from [159].)

Of particular relevance are abstract data types and their proper encapsulation, atomic transactions, and various forms of consistency. It becomes clear that good software-engineering practice can contribute directly to the enforcement of the Clark–Wilson properties and to other generalized forms of system integrity.

Unless a Clark–Wilson type of application is implemented directly on hardware with no underlying operating system, the sound enforcement of the Clark–Wilson properties will typically depend upon lower-layer (e.g., operating system) integrity properties. Sound implementation is considered by Karger [104] using a secure capability architecture (SCAP [105]), and by Lee [114] and Shockley [228] employing an underlying trusted computing base that uses the notion of 'partially trusted subjects' and that enforces both multilevel security (MLS) and multilevel integrity [19] (MLI).

A useful report analyzing Clark–Wilson integrity is found in [4].

Table 2 summarizes the ways in which software engineering can potentially contribute to the architecture of sound systems, including the prevention and detection of the characteristic misuse techniques, and the avoidance of or recovery from unreliability modes (through reliability and fault-tolerance mechanisms). In the table, a plus sign indicates a positive contribution, whereas a minus sign indicates a negative effect; parentheses indicate that the contribution (positive or negative) is a second-order effect or a potential effect. (This table is derived from [159].)

The table illustrates in overview that good software engineering can contribute significantly to system security and integrity, as well as to system reliability and human safety, with respect to the underlying systems and to the applications. However, the mere presence of a technique is not sufficient. All these techniques are frequently touted as magical answers, which they certainly are not. Each can be badly misused.

Table 1: Summary of Clark–Wilson integrity properties

CW rule	Clark–Wilson Integrity Properties for Enforcement (E) and Certification (C)
CW-E1	Encapsulation of abstract data types
CW-E2	User authorization
CW-E3	User authentication
CW-E4	Nondiscretionary controls
CW-C1	External data consistency
CW-C2	Transformation (internal) consistency
CW-C3	Separation of privilege and least privilege
CW-C4	Complete nontamperable auditing
CW-C5	Atomic input validation

2.6 Structural Encapsulation

Table 3 illustrates the potential effects of hardware, software, environmental, and other problems in conventionally designed systems and networks, in which widespread disasters may be triggered at any layer — for example, as a result of security breaches or system malfunctions. For simplicity, only three layers are shown, denoted as 0, 1, and 2, which might typically correspond to an operating system, an application environment, and user environments, respectively. The 1980 4-hour collapse of the entire ARPAnet and the 1990 11-hour collapse of AT&T long-distance services are examples of how seemingly minor local events can trigger widespread havoc. (See [159] for details.)

In contrast to such an undifferentiated and not wisely layered design, it is desirable to have an architecture that places the most critical functions in the lowest layers, and then builds upon those functions. In such a structure, it is desirable that defensive design be able to ensure that failures at the lowest layer are very unlikely and if they do occur, their effects can be contained; that failures in the next higher layers are only locally serious; and that failures in the highest layers are usually inconsequential. An example is provided by the original ring structure of Multics, in which Ring 0 failures might crash the system, Ring 1 failures might crash the user process but not the system, and Ring 2 failures might abort the user command without damaging the user process. More recent examples are provided by architectures based on a kernel that enforces multilevel security, with a trusted computing base that encapsulates subsystems that must be trusted to violate the strict-sense multilevel security of the kernel and applications implemented on top of the TCB.

In the present context, *trust* may imply something other than just having to be trusted with respect to multilevel security; it might typically imply trustedness for certain security and reliability properties, for example, or more generally any properties that must be maintained — such as Clark-Wilson consistency.

In the view of the TCSEC Orange Book [151], there is an ambiguity in the definition of *trusted,* with one or both of two meanings: (1) a system *can be trusted* because it is believed to satisfy certain requirements (for example, for reliability or security); (2) a system *must be trusted* because deleterious consequences could result if it fails to satisfy its requirements. Whether the system is actually trust*worthy* is another matter. In this report, calling a system or component *trustworthy*

Table 2: Software-engineering principles and their potential effects

Principles	Properties to be ensured: Secure data	System integrity	System reliability, availability	Identity, authentication	Auditing	Application properties
Modular decomposition	+	+	+	[+]	+	+
Modular composition	[+]	[+]	[+]	+		+
Strict isolation	+	+	+	+	+	[+]
Abstraction, encapsulation, information hiding	+	+	+	+	+	+
Hierarchical layering	+	+	+	+	+	+
Type safety	+	+	+	+	[+]	+
Object orientation	+	+	+	+	+	+
Parameterization	[+]	+	[+]	[+]	[+]	[+]
Inheritance	+	+	+	+	+	+
Separation of duties	+	+	[+]	+	+	+
Least privilege	+	+	+	+	+	+
Virtualized location	+	+	+	+	[+]	+
Virtualized networking	+	+	+	+–	[+–]	+
Virtualized concurrency	+	+	+			+
Virtualized replication	+	+	+			+
Virtualized recovery	+	+	+		[+]	+
Fault tolerance	[+–]	+	+		[+]	+

Legend:
Secure data includes data confidentiality and data integrity
+ = a primary contribution
– = a potential negative implication
[ ] = a second-order or potential effect

Table 3: Conventional system architectures

Layer	Typical hierarchy	Effects on the system
2	Higher-layer functions	Global disaster possible from failure, penetration, or accidental misuse
1	Middle-layer functions	Global disaster possible from failure, penetration, or accidental misuse
0	Low-layer functions	Global disaster possible from failure, penetration, or accidental misuse

implies that its trustworthiness can or must be demonstrated in some convincing way; we tend to avoid ambiguous uses of the term *trusted.*

Table 4 illustrates such a conceptual hierarchy in which defensive measures are taken appropriate to each layer. In particular, at the lowest and potentially most vulnerable layer, the security and fault-tolerance measures are most stringent, in an effort to prevent widespread effects. At higher layers, such measures may be less necessary, because the design structure ensures that such effects cannot occur. This conceptual hierarchy applies equally to isolated systems, distributed systems, and networked systems where the networking must itself be robust — reliable and sufficiently secure. The significant point is that the "kernel" at any layer must be trustworthy with respect to whatever requirements are necessary, and that the functionality at one layer becomes the logical "kernel" for the next higher layer.

This structural encapsulation of criticality works very nicely for multilevel security and certain other security properties. In this way, secure systems can be composed out of components with some assurance that the higher layers cannot circumvent or compromise the properties enforced by the lower layers, and also cannot modify the lower-layer functionality — for example, by introducing a Trojan horse. (In this report, we use the term *compromise* in the negative sense of subverting. See Section 2.8, which elaborates on the notions of compromisibility and noncompromisibility.) Structural separation also works to a considerable extent for reliability, where mechanisms that are more reliable can be developed using less reliable components. In addition, it is relevant to life-critical applications, although in such environments there is always a potential danger that the highest-layer functionality may be sufficiently untrustworthy that it can result in loss of life despite the integrity of the lower-layer functionality. However, the goal of structural encapsulation in a functional hierarchy is to isolate the lower layers from tampering and other forms of compromise. (Tampering implies modification; compromise implies subversion, but does not necessarily imply modification — although it includes tampering as a special case.)

Structural encapsulation is also relevant in distributed systems in which the physical and logical separation of components can be used to advantage by enforcing certain additional controls, such as remote access controls and authentication of both users and components. In this case, the encapsulation is relevant horizontally (among comparable subsystems) and vertically (between hierarchically layered subsystems).

Encapsulation is particularly important with respect to the uses of cryptography, as discussed in [158]. Whether a crypto implementation is in hardware or in software is often less important than how the crypto is encapsulated. For example, a flaw in the algorithm exists in both cases; a flaw in the embedding of the crypto into an operating system or hardware chip may be compromisible in either case; the presence of unencrypted forms that are not suitably protected can be a risk in either case. The major advantage of a carefully encapsulated hardware implementation (including tamperproofing and internal keys that never emerge from the hardware) arises in connection with authentication that can be done completely in hardware, as in the case of the Fortezza chip (nèe Tessera). However, even such a completely encapsulated authentication chip must be protected against spoofing attacks that might simply bypass the chip or replay its positive acknowledgments.

2.7 Dependence and System Decomposition

The notion of *dependence* is a fundamental structuring concept. Parnas [176] has examined a variety of types of dependence relations, among which the relation `depends upon for its correctness`

Table 4: A constructive hierarchy

Layer	Hierarchy	Effects of hardware, software, and other problems
2	Noncritical functions, isolated	Disasters unlikely to be caused by noncritical (e.g., untrusted) software, due to sharply defined design isolation; assurance needed for correctness of certain functions and application properties
1	Somewhat-critical functions	Disasters sharply limited by layer separation and limited trust in this layer
0	Most-critical functions	Disasters possible but unlikely if systems are partitioned sensibly, with hardware fault tolerance, reliable software, rigorous authentication; must be simple enough to be analyzed thoroughly

is particularly relevant to many systems. This relation (which we abbreviate here as `depends upon`) can be used to induce an ordering on the components of a system, on subsystems, or on different networked systems. In some cases the ordering is purely hierarchical, in the sense that the components are linearly ordered or perhaps lattice ordered. In those cases in which the linear ordering reflects the levels of abstraction, we speak of the system as being hierarchically layered by its vertical dependencies; that is, the objects of a particular level of abstraction `depend upon` only lower layers. In other cases, there may be an ordering of different subsystems with comparable levels of abstraction, as in the case of a network of more-or-less coequal systems. In these cases, we can still determine the presence of horizontal dependencies. In other cases, some dependencies may be mutual (as in A `depends upon` B *and* B `depends upon` A), in which case no hierarchical ordering can exist unless those mutually dependent components are collapsed into a single layer, with just one component. In the worst case, when the totality of all dependencies is considered, an entire system may necessarily collapse into a single layer, implying that there is no possible substructure based on the `depends upon` relation. That is, the iterative closure of all dependencies among subsystems results in a single system that contains precisely the set of all subsystems. Largely unsubstructured systems are typical of the situation depicted in Table 3.

In general, in many system architectures, mutual dependence is a very bad practice — particularly if a depended-upon component is less trustworthy than the components that depend upon it. In a simplistic sense (which we do not insist on here), dependence on another component is acceptable only if that other component is at least as trustworthy, with respect to meeting its expected requirements. This simplistic sense is roughly equivalent to Biba's notion of strict multilevel integrity [19], in which dependence on less trustworthy entities is not permitted — at least not when trustworthiness and trustedness are equated. (Biba assigns multilevel integrity labels to subjects and objects, and enforces a strict lattice ordering that is a precise formal dual of the Bell and LaPadula multilevel security lattice.)

The notion of dependence used in this report is more general than that found in the strict hierarchical layerings induced by the `depends upon` relation. Indeed, the generalization used here includes the simple case in which the lower-layer (depended-upon) functionality is at least as trustworthy, but it also includes the more complex case in which the upper layer is able to tolerate untrustworthiness in lower-layer functionality. This generalized concept also extends to dependence on other subsystems at a comparable layer of abstraction. Furthermore, it also applies to properties at one layer depending on properties at lower layers.

As examples, here are three design techniques that permit such generalized dependence:

- Error-correcting codes, in which reliable communications or storage representations can be obtained by adding suitable redundancy, despite the presence of unreliable media
- Fault tolerance, whereby a system can continue to perform correctly despite the occurrence of certain types of simultaneous faults
- Byzantine algorithms, in which correct behavior can be achieved despite arbitrary misbehavior of a certain number of faulty or malicious components

These techniques are discussed subsequently.

In addition to encouraging a well-structured system design decomposition, this generalized notion of dependence can also increase the soundness of the system if the design and implementation are such that the depended-upon layers cannot be compromised from above (see Section 2.8) — for example, through security penetrations or accidental misbehavior. Although no pun is intended, consistent use of *dependence* in structuring a system can have beneficial effects in increasing the *dependability* of the resulting system.

The Biba notion of strict lattice-ordered nondependence is much too restrictive in many system contexts. Particularly in distributed systems, in which certain components may have unknown trustworthiness, it becomes difficult to structure functional dependence based on trustworthiness. In those cases, explicitly trusted mechanisms may desirably be invoked to mediate between trusted components and questionable components. Confidentiality, integrity, and prevention of denials of service are all relevant security issues. The gamut of safety issues must also be addressed, and fault-tolerance techniques or Byzantine algorithms may be required in certain cases. Thus, we seek here a structural framework in which reliable and trustworthy systems can be developed even if they must rely on less reliable and less trustworthy subsystems. This generalized dependence relation is denoted here simply as `depends on`, rather than the strict-sense Parnas relation, `depends upon [for its correctness]`. Not only is the prepositional simplification from *upon* to *on* consistent with modern English-language usage, it also is meant to suggest a broader sense of dependence. (The `depends on` relation is close to, but not identical to, Parnas's `uses` relation, and seems worthy of being identified on its own.)

It is useful to distinguish between design principles and implementation principles. Modular decomposition, hierarchical layering, fault tolerance and other constructive uses of redundancy, and separation of duties are examples of design principles that can pervasively affect the design of a system.

Decomposition can take on several forms. Vertical decomposition recognizes different layers of abstraction and separates them from one another. Horizontal decomposition (modularization) is useful at any particular design layer, identifying functionally distinct components. It can be implemented in many ways, through coordination from higher layers, local message passing, or networked interconnections. Refinement provides a less obvious form of temporal decomposition in which the representation of a particular function, module, layer, or system interface undergoes successive definitude — for example, evolving from a requirements specification to a functional specification to an implementation, and perhaps with additional functionality being added along the way.

Jim Horning (at Digital's Systems Research Center in Palo Alto) offered me the following guidance relating to system decomposition:

> Decomposition into smaller pieces is a fundamental approach to mastering complexity. The trick is to decompose a system in such a way that the globally important decisions can be made at the abstract level, and the pieces can be implemented separately with confidence that they will collectively achieve the intended result. (Much of the art of system design is captured by the bumper sticker "Think globally, act locally.")

Various implementation principles are also desirable to ensure that the design principles are properly enforced by an actual system implementation; for example, the intent of structural design properties must be preserved throughout the implementation. The distinction between design and implementation is especially important in attempts to protect against both accidental system misbehavior and intentional system misuse. For example, redundancy techniques are generally thought to be aimed at limiting accidental problems (for example, hardware fault modes), while separation of duties is generally thought to be aimed at controlling intentional compromises. The two concepts are in fact related, and each approach is applicable to both types of problems; the two concepts can complement each other. Unfortunately, shortsighted views of the design and its implementation generally result in inadequate solutions, and it is often in the design and implementation of the system structure that flaws are introduced.

To motivate the generalized `depends on` relation, we consider various specific efforts to develop reliable components out of less reliable building blocks, elaborating on the three bulleted design techniques on the previous page.

- Error-correcting codes with parameters (n, N, d) typically permit up to e random bit errors to be corrected out of a block length of n bits, where $d = 2e + 1$ is the minimum Hamming distance of a code containing N distinct code words. (The Hamming distance [86] between two equal-length binary code words is the number of bit positions in which the code words differ.) Codes have also been developed for arithmetic processes, for byte-oriented representations, and for certain classes of dependent errors. These codes provide a basic example of how reliability can be gained through the constructive use of redundancy [187, 195].

- An early paper by von Neumann demonstrated that reliable components could be built out of unreliable components [248], as long as the probability of failure is not precisely one-half. A contemporaneous work by Moore and Shannon [143] demonstrated essentially the same result for switching circuits built out of what were then affectionately referred to as *crummy relays*. These papers provided early examples of fault-tolerant designs.

- Even more stringent requirements are met by fault-tolerant and Byzantine algorithms [113, 222], although those requirements span a wide range. A typical fault-tolerant storage component could provide correct retrieval of a file despite the outage of an entire file server, by using duplicate servers. A typical fault-tolerant system could ensure correct and undegraded

> operation despite the failure of any one processor, and ensure degraded but fail-safe performance after the concurrent failure of a second processor, under suitable assumptions relating to tolerated failure modes. A typical Byzantine algorithm goes even further; for example, it could permit correct operation despite the arbitrary and completely unpredictable behavior (maliciously or accidentally) of up to f out of its n components, with no assumptions regarding failure modes. A Byzantine clock subsystem that can perform correctly despite arbitrary misbehavior within up to f of its constituent $3f+1$ clocks [113, 213] is discussed in Section 8.

To reiterate, we observe that, within this conception of layered abstraction, a reliable subsystem implemented out of less reliable components `depends on` the underlying abstractions, but does not necessarily `depend upon` them `for its correctness`. Similarly, the properties at one layer `depend on` the lower-layer properties, and in principle should be derivable from the lower-layer properties and the functionality at the given layer. (For example, see the mapping techniques described by Robinson and Levitt [199].)

Closely related to Byzantine algorithms that provide fault-tolerance and misuse protection are some of the multikey crypto algorithms. For example, Micali [142] demonstrates the possibilities of requiring a confluence of n key-holders to enable decryption or signing (for authenticity or integrity). Desmedt, Frankel, and Yung [52] have generalized that approach, whereby the crypto can function properly despite the presence of at most f Byzantine agents (for example, faulty, malicious, or simply not available) out of a total of n agents, whereby at least $n - f$ agents must be functioning properly.

Similarly, a trustworthy secure subsystem implemented out of less trustworthy components `depends on` the underlying abstractions, but does not necessarily `depend upon` them `for its correctness`. An example analogous to Byzantine algorithms is provided by the fragmented key-escrow scheme of Reiter and Birman [196], whereby at most f out of n escrow agents can be untrustworthy (maliciously or accidentally) or unavailable — that is, the contributing participation of at least $n-f$ out of n different escrow agents is required to enable successful decryption. Another example is provided by encryption, which converts an untrustworthy communications medium into a medium that does not have to be trusted for confidentiality or integrity. A further example involves an inherently flawed subsystem that is used by a higher layer, but for which the higher-layer encapsulation is able to completely mask its flaws, rendering those flaws unexploitable from the encapsulated interface and also ensuring that the higher-layer interface is nonbypassable — that is, that the flawed subsystem cannot be called on directly without going through the intermediation of the higher-layer interface.

The Desmedt-Frankel-Yung and Reiter-Birman approaches both provide examples of a separation of duties that incorporate a Byzantine-like resilience; each has architectural implications as well as analytic implications. Overall, we wish to encourage approaches in which trustworthiness can be increased through use of trustworthy mechanisms, despite the possible presence of untrustworthy components.

Section 8 considers the process of composing a system out of its subsystems and deriving the resulting system properties from its subsystem properties. This concept is of vital importance in distributed systems and networks of such systems.

2.8 Compromisibility and Noncompromisibility

To illustrate the importance of dependence on properties of underlying abstractions, consider the necessity of depending on a life-critical system for the protection of human safety. In such a system, safety ultimately `depends upon` the confidentiality, integrity, and availability of both the system and its data. It also may `depend upon` component and system reliability, and on real-time performance. Furthermore, it usually `depends upon` the correctness of much of the application code. In the sense that each layer in a hierarchical system design `depends upon` the properties of the lower layers, the way in which trusted computing bases are layered becomes important for developing dependably safe systems — particularly in those cases in which the `depends on` relation can be used more appropriately instead of `depends upon` to accommodate an implementation based on less trustworthy components.

The same dependence situation is true of secure systems, in which each layer in the hierarchy (for example, consisting of a kernel, a trusted computing base for primitive security, databases, application software, and user software) must enforce some set of security properties. The properties may differ from layer to layer, and various trustworthy mechanisms may exist at each layer, but the properties at a particular layer are generally derived from lower-layer properties.

In the security context, there are many notions of compromise. For example, compromise might entail accessing supposedly restricted data, inserting unvalidated code into a trusted environment, altering existing user data or operating-system parameters, causing a denial of service, finding an escape from a highly restricted menu interface, or installing or modifying a rule in a rulebase that results in subversion of an expert system.

There is an important distinction between having to depend on lower-layer functionality (whether it is trustworthy or not) and having some meaningful assurance that the lower-layer functionality is actually noncompromisible under a wide range of actual threats. Noncompromisibility is particularly important with respect to security, safety, and reliability.

Potentially, a supposedly sound system could be rendered unsound in any of three basic situations:

- Compromise from above (intuitively, outside)
- Compromise from within (intuitively, inside)
- Compromise from below (intuitively, underneath)

Each of these situations could be caused intentionally, but could also happen accidentally. (For descriptive simplicity, a *user* may be a person, a process, an agent, a subsystem, another system, or any other computer-related entity.)

- *Compromise from above.* Compromise from above is typically performed from an access point that is nominally external to the component being compromised, and is typically perpetrated by a completely unprivileged user, or by a privileged user gaining access to perpetrate a further compromise. In general, no authorization is required, possibly because of an exploitable flaw in the standard interface.
- *Compromise from within.* Compromise from within is typically performed by a user who has somehow gained access (with or without authorization) to the internals of a component, such

as privileged maintenance access to a database management system, a network controller, or an automatic teller machine. It could be perpetrated by an authorized user who is misusing privileges, or by a penetrator. Compromises from within include all the compromises possible from above, plus certain additional compromises. In addition, compromises from above may subsequently enable compromises from within.

- *Compromise from below.* Compromise from below is typically performed by a user who has somehow gained access (with or without authorization) to layers of abstraction underlying a particular component that is being compromised, which can then be undermined without attacking the component itself. Compromise from below may result from malicious action or accidental failure of an underlying mechanism on which the particular component depends. Examples include (1) obtaining the unencrypted form of an encrypted message by reading a temporary file in storage, (2) finding an occurrence of a particular word in a restricted database to which access is not permitted by scanning the disk on which that database is stored, and (3) editing an enqueued mail message after it is released by a user but before it is actually sent out by the mailer. Compromises from below include all the compromises possible from above or within, plus certain additional compromises. In addition, compromises from above or within may subsequently enable compromises from below.

The distinctions among these three modes tend to disappear in systems that are not well structured, in which inside and outside are indistinguishable (as in systems with only one protection state), or in which above and below are merged (as in flat systems that have no concept of hierarchy).

Certain attack modes may occur in any of these forms of compromise. Examples of Trojan-horse perpetrations are as follows:

- Compromise from above: a letter bomb (e.g., electronic mail) that when read or interpreted can result in unanticipated executions, or a spoofing attack that piggybacks on a line or replays a message
- Compromise from within: a surreptitious code patch that maintains a hidden trickle file of sensitive information within the program data
- Compromise from below: a wiretap implanted inside a telephone switch, or Ken Thompson's now-classical object-code modification of the C compiler that permitted a trapdoor routine to be planted in the login [242] (whereby it becomes clear that system security also `depends upon` the compiler)

Table 5 summarizes some properties whose nonsatisfaction could potentially compromise system behavior, by compromising confidentiality, integrity, availability, real-time performance, or correctness of the application code, either accidentally or intentionally. To illustrate such compromises, Table 5 also indicates possible compromises — whether they involve modification (tampering) or not — that can occur from above, from within, or from below, for each representative layer of abstraction. The distinctions are not always precise: a penetrator may compromise from above, but once having penetrated, is then in position to compromise from below or from within. Thus, one type of compromise may be used to enable another. For this reason, the table characterizes only the primary modes of compromise. For example, a user entering through a resource access control

Table 5: Illustrative compromises

Layer of abstraction	**Compromise from above**	**Compromise from within**	**Compromise from below**
Environment	Acts of God	Lightning Earthquakes	Chernobyl-like disasters
User	Masqueraders	Accidental mistakes Intentional misuse	System outage or service denial
Application	Penetrations of application integrity	Programming errors in application code	Application (e.g., DBMS) undermined within operating systems (OSs)
Operating system	Penetrations of OS by unauthorized users	Flawed OS software Trojan-horsed OS Tampering by privileged users	OS undermined from within hardware: faults exceeding fault tolerance; hardware sabotage
Hardware	Electromagnetic and other interference	Bad hardware design/ implementation Hardware Trojan horses Unrecoverable faults	Power failures

package such as RACF or CA-TopSecret, or through a superuser mechanism, and gaining apparently legitimate access to the underlying operating system may then be able to undermine both operating-system integrity (compromise from within) and database integrity (compromise from below if through the operating system), even though the original compromise is from above. Similarly, a software implementation of an encryption algorithm or of a cryptological check sum used as an integrity seal can be compromised by someone gaining access to the unencrypted information in memory or to the encryption mechanism itself, at a lower layer of abstraction. A user exploiting an Internet Protocol router vulnerability may initially be able to compromise a system from within the logical layer of its networking software, but subsequently may create further compromises from above or below.

From the table, we observe that a system may be inherently compromisible, in a variety of ways. The purpose of system design is generally not to make the system completely noncompromisible, but rather to provide some assurance that the most likely and most devastating compromises are properly addressed by the design, and — if compromises do occur — to be able to determine the causes and effects, to limit the negative consequences, and to take appropriate actions. Thus, it is desirable to provide underlying mechanisms that are inherently difficult to compromise, and to build consistently on those mechanism. On the other hand, in the presence of underlying mechanisms that are inherently compromisible, it may still be possible to use Byzantine-like strategies to make the higher-layer mechanisms less compromisible. However, flaws that permit compromise of the underlying layers are inherently risky unless the effects of such compromises can be strictly contained.

2.9 Designing for Multilevel Security

The foregoing sections suggest that when designing a system or network of systems to enforce multilevel security throughout, care should be taken to minimize the trustworthiness required at the highest layers, to simplify the trustworthiness that must be required at the lowest layers, and to ensure that no untrusted higher layer can compromise a lower layer. For example, in a local network of workstations with highly distributed administration and control, it is inherently dangerous for a system design to have to place trust in every workstation to enforce multilevel security and other properties. However, conventional multilevel-secure architectures are generally based on the assumption that all the end-user system components enforce multilevel security, that is, every user workstation must conform to the appropriate Orange-Book B-level criteria.

The work of Proctor and Neumann [192][4] and the subsequent generalization by Neumann and Gong [163] propose an alternative family of architectures, each member of which is consistent with the structural desiderata noted above; those architectures have in common the complete avoidance of multilevel-secure end-user systems. That work is reconsidered in Section 5.

Appropriate use of system structure can contribute in many ways to system development.

- Systems that can be composed out of carefully analyzed (e.g., proven) components can lead to improved security and reliability, and simpler analysis of the overall system. Reducing the portions of systems and networks that have to be trusted with respect to various measures of trustworthiness tends to enhance design integrity, reduce the extent of implementation flaws, facilitate system operation and maintenance, and simplify system evaluation.

The structural integrity of a system design must be preserved throughout implementation and operation, across distributed systems and networks and otherwise composed systems.

Similar comments apply to systems that must enforce related mandatory policies such as multilevel integrity (Section 2.3) and multilevel availability (Section 2.4), subject to the facility for certain trusted exceptions.

The notion that hierarchically designed systems must necessarily be inefficient is a myth, although poorly conceived structure can of course be detrimental. Overall, the constructive use of appropriate design structures is an undervalued technique in the quest for systems with increased security.

3 Three Representative Structured Architectures

Section 3 considers three historically interesting architectural designs: Multics, a pioneering operating system from the 1960s; PSOS, the design of a highly layered capability-based object-oriented operating system from the 1970s; and SeaView, a database management system (DBMS) environment implemented on a multilevel-secure kernel from the 1980s. Each of these systems in turn represents significant advances in the use of architectural structure to enhance security. These three systems are chosen here because each represents what was at the time a relatively pure self-consistent set of innovations, from which many other systems have subsequently borrowed hybridized subsets of concepts.

Distributed-system architectures are deferred until Section 4.

[4] That work began under a study of distributed-system security [164], performed for Rome Laboratory.

3.1 Multics

Of particular interest from various historical perspectives is the extent to which architectural structure played a role in the development of Multics, which was a joint effort among MIT, Bell Telephone Laboratories and Honeywell (initially General Electric). The Multics hardware design began around 1963, and the software effort began in earnest in 1965. The original Multics design represented several very significant innovations. The file system introduced multitiered tree-structured directories [49], with symbolic file names that were dynamically bound to run-time instances of real storage objects, symbolic file-name links that acted as aliases, and a dynamically specifiable search strategy that determined the order in which directories were searched for a particular file name. The use of local and tree-structured symbolic file names enforced strict information hiding of the corresponding dynamically linked hardware-recognizable segment numbers [48], where the specific objects were determined according to a dynamic search strategy. The processor hardware supported a two-dimensional segmented virtual memory and dynamic paging within each segment — with each virtual-memory segment being independently protectable.[5] The hardware-implemented ring mechanism [79, 171] permitted the implementation of nested protection domains with different privileges operating within a single process, and provided the ability to switch rapidly from one domain to another, with appropriate argument validation. The associated stack discipline also added a layer of structural domain isolation with carefully controlled interprocess communication.

The use of a higher-level programming language (Multics used the very first implementation of PL/I, done by Doug McIlroy and Bob Morris) added to the structural integrity of the development process, by enforcing system-wide data declarations, stack disciplines, and linkage conventions. It also facilitated the virtualization of the processors, whereby Multics was capable of shared-memory multiple-processor execution. Each of these hierarchical structuring concepts added something significant to the security attainable and to the ease of developing new subsystems. Relative to what was going on elsewhere in the mid-1960s, Multics represents a remarkably innovative approach.

In the early 1970s, a subsequent restructuring of the innermost Multics system rings permitted the retrofitting of kernel-enforced multilevel security [225] into the standard commercially available system.

3.2 PSOS: Layered, Capability-Based, Object-Oriented

The Provably Secure Operating System [62, 161][6] was designed by SRI over the period from 1973 to 1977, with a relatively minor revision of the 1977 report in 1980, and a subsequent implementation feasibility study that continued into the early 1980s. In retrospect, PSOS represents what to our knowledge is the first object-oriented operating-system design (although the concepts of inheritance and polymorphism were only in their infancy at the time and were not yet so named). Table 6 summarizes the layered hierarchical design of PSOS, in which the abstraction at each layer acts as the object manager for the type of objects that it creates or is otherwise responsible for; each object manager completely encapsulates its implementation. Some of the abstractions are present

[5] An additional structural separation was provided by the input-output coprocessor (the GIOC), which permitted interrupts to be fielded independently of the main processors and various processing to be done in parallel with the general-purpose CPUs. However, the early design did not adequately protect main memory from the GIOC, which had global permission and could perform reading and writing in terms of absolute addresses, bypassing the security provided by the operating system. This weakness was later rectified.

[6] An obsolete early paper [162] is of potential historically interest.

Table 6: PSOS abstraction hierarchy

Layer	PSOS Abstraction or Function
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 and windows (*)[c11]
7	paging [8]
6	system processes and 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 *

Notes	Interpretation
*	= functions visible at user interface
(*)	= partially visible at user interface
[i]	= module hidden by level i
[c11]	= creation/deletion hidden by level 11

at multiple layers in the hierarchy. For example, the system-process module at layer 6 is logically a part of the specifications at layers 7 through 11, but is replaced by a higher-layer user-process abstraction at layer 12. Similarly, the creation and deletion functions of layers 8, 9, and 10 are replaced at layer 11 by similar but more abstract user-oriented functionality. (The masking of lower-layer functionality by higher layers is indicated by square brackets in Table 6.)

A few functions appear as parts of all layers (indicated by asterisks in Table 6), as is the case with the capability mechanism — which is the essence of PSOS. Capabilities are nonforgeable hardware-created tagged primitive objects (that is, they are not handled by the extended-type mechanism), and are the basis of accessing all higher-layer objects, each of which has an associated typed capability. The objects of any particular type are managed solely by the corresponding object manager for that type. The table exhibits various important layers of objects such as pages, segments, type creation, hierarchically structured directories, processes, and named user objects. The design explicitly permits the creation and implementation of application-layer types, and the entire design is recursively extensible and open-ended. Thus, the Multics restriction of a small finite number of domains does not apply.

We observed at the time, but did not pursue in detail, that the basic architecture was amenable to distributed implementations, because the abstractions at each layer (for example, object-oriented

Table 7: PSOS generic hierarchy

Layer	PSOS generic hierarchy layer	Layers
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

Table 8: PSOS properties

Layer	Illustrative PSOS Properties (Not all are primary security properties)
17	Soundness of user type managers
15	Search path flaw avoidance
12	Process isolation, user input-output integrity, user authentication integrity
11	No lost objects (i.e., without capabilities)
9	Generic type safety
8	Correct segment access, no storage residues
6	Network and system input-output integrity, hardware interrupts properly masked
4	Correctness of hardware instructions
0	Capabilities nonforgeable, nonbypassable, nonalterable (MLS if desired at this layer)

type managers) could be distributed — either implicitly (invisibly) or explicitly (requiring program or user awareness of remote locations). The notion of distributed objects is considered in Section 4.

Table 7 gives a conceptual generic hierarchy that is an abstraction of the PSOS architecture. The basic PSOS architecture represented a single-level secure system. However, a multilayered design for enforcing multilevel security was also included. In particular, it was noted that MLS could be incorporated into the virtual resources layers, or directly incorporated into the existing layers, with hardware that supports multilevel capabilities directly at the lowest layer. The latter alternative was chosen in Honeywell's PSOS-inspired Secure Ada Target, which was the precursor of SCTC/SCC's LOgical Coprocessor Kernel, LOCK.

Table 8 illustrates typical properties to be enforced and verified at various layers. The integrity of the capability mechanism (intended to be implemented in hardware) is fundamental to PSOS, because that mechanism acts as a microkernel upon which all addressing is based. However, each layer contributes relevant additional security properties.

Table 9: SeaView design hierarchy

Layer	**SeaView TCBs and their functionality**
7	UNTRUSTED: User applications
6	TRUSTED FOR USER INTERFACE INTEGRITY: User interface, presentation view manager, conventional DBMS functions
5	TRUSTED FOR MSQL INTEGRITY: MSQL provides virtual multilevel relations and views
5	TRUSTED FOR DB DAC, DB INTEGRITY: DBMS Resource Manager, Extended DB DAC and DB consistency
3-4	TRUSTED SINGLE-LEVEL RELATIONS: DAC in operating-system TCB, DAC on relations
1-2 0	OPERATING SYSTEM MLS TCB: Mandatory MLS in operating-system TCB Mandatory MLS in operating-system kernel segments

3.3 SeaView: An MLS DBMS without MLS trustworthiness

The significance of the multilevel-secure kernel approach is illustrated by SRI's SeaView architecture [50, 125, 126, 128, 191] for a multilevel-secure database management system. SeaView explicitly uses multiple single-level databases to provide a multilevel-secure DBMS within which no trustworthiness for multilevel security is required. The hierarchy is summarized in Table 9. The abbreviation DAC is used for discretionary access controls.

SeaView illustrates the presence of different but mappable properties at each hierarchical layer.[7]

The SeaView DBMS enforces a variety of database-relevant integrity properties [125], including entity integrity, referential integrity, and polyinstantiation integrity,[8], as well as other consistency properties. These properties are modeled formally in the SeaView work [125, 191]. Additional notions of system and data integrity relate to integrity of atomic transactions, supposedly identical replicated versions, and multiple versions possibly differing in recency.

The SeaView design structure is summarized in Table 9. The SeaView architecture is based on an MLS kernel and an operating system TCB that enforces mandatory security on storage objects. The original prototype development used Gemini's GEMSOS OS-TCB, although GEMSOS was subsequently replaced by the Sun Compartmented Mode Workstation.

[7] Earlier recommendations for designing multilevel-secure DBMSs are found in [89, 24, 218]. TCB layering is considered in [217], and had previously been considered in a more limited form in the database context by [89, 81].

[8] Polyinstantiation is a solution to the problem of avoiding inference channels arising in multilevel secure databases. For example, suppose a low-level subject attempts to create a data item that the subject is not allowed to know already exists, because such an item has already been created by a high-level subject as a high-level object. The low-level subject cannot be given any negative response, because that would cause an adverse information flow; consequently, everything must appear as if the low-level operation has succeeded (which it has!), by including the creation of a polyinstantiated instance of the item.

Table 10: SeaView model hierarchy

Model	**SeaView Model Properties**
DB-TCB	Database TCB for views, multilevel (virtual) relations, discretionary access, labeling, data consistency, sanitization, reclassification, constraints on transactions, polyinstantiation integrity
OS-TCB	MLS security for single-level base relations

The OS-TCB is used to enforce mandatory security on single-level base relations for SeaView. An extended database TCB (DB-TCB) enforces various other properties such as database integrity and database discretionary access controls (DB DAC) on multilevel user relations (which are invisibly [virtually] mapped to single-level base relations), but is completely untrusted with respect to the enforcement of MLS. That is, the DB-TCB cannot compromise the MLS property, whose enforcement is completely controlled by the underlying OS-TCB. However, the DB-TCB is trusted to enforce the otherd database properties such as integrity and database discretionary access. The database management system is the Oracle RDBMS. The database language is called MSQL, for *multilevel-secure SQL*, an extension of single-level SQL. Integrity of the MSQL interface is also enforced for the sanity of the users, although it is not essential to security.

The MSQL processor is shown as a layer above the layer that is trusted for discretionary database confidentiality and discretionary database integrity, although the prototype implementation of both layers is in the same GEMSOS ring (ring 5). With the original ring assignments among the eight GEMSOS rings, a trusted user application would have to be implemented in GEMSOS ring 7, along with its users. Thus, even in a relatively simple system such as SeaView, there would be some advantages to a domain architecture that could provide further layering as well as separation within a layer such as in LOCK [23] or PSOS [62, 161], to support competing application environments. A ring or two could of course be freed up, for example, if rings 1 and 2 were collapsed into one ring, and rings 3 and 4 similarly, although the domain separation at the application layer might still be desirable. Overall, the table thus illustrates the different meanings of "trust" and "trustworthiness" at each hierarchical layer.

The properties at the operating system TCB layer and at the database TCB layer are summarized in Table 10. There is a distinct model for each of the TCBs. The OS-TCB enforces MLS on operating-system objects, while the DB-TCB enforces database security and integrity on database objects. Layering raises a question not of composability of components with respect to a given property (e.g., MLS), but rather of whether the higher layer can compromise the lower layer, because the upper layer is enforcing different properties on different types of objects. SeaView represents a system in which the kernel multilevel security properties and the database integrity properties are both largely noncompromisible from above.[9]

The SeaView model [50, 124, 125] does not explicitly address distributed data, and the near-term design focuses on a single database system [128]. A conceptual extension of the original design to a distributed implementation is considered in Section 4.1.

[9] Tables 9 and 10 and their discussion are adapted from [156].

4 Distributed-System Architectures

If each system in a collection of networked systems is itself structured fairly well — for example, in terms of object abstraction and compatible layering — then the structural concepts in the networking itself can add significantly to the security attainable. It is desirable to identify separately the security of the constituent operating systems, the security of the networking software and hardware, and the security of the networks themselves (for example, through the use of encryption). However, the integration of this trichotomy is fundamental to the ability to develop sound networks of systems.

4.1 Structural Concepts in Distributed Systems

From a security point of view, distributed systems must satisfy all the security requirements of a stand-alone centralized system. In addition, distributed systems present many additional vulnerabilities, such as weaknesses in the networks and other opportunities for attacks that are not usually considered in centralized systems. Consequently, much greater effort must be devoted to network media confidentiality and integrity, network software integrity, and pervasive authentication — of users, agents, and hosts. The partitioning of a distributed system into functionally distinct components can add greatly to the security of the overall system. In that way, servers — for example, network servers, file servers, and authentication servers — can be special-purpose systems that are better protected against conceivable types of misuse, whether malicious or accidental.

Many serious security flaws in networked systems have resulted from the lack of proper abstraction and encapsulation, and from the absence of both user and component authentication. In communications across organizational hierarchies, generals tend to speak only with generals, and CEOs tend to speak only with CEOs. A comparable protocol hierarchy is useful in distributed systems and networks, if it restricts interactions to those among peers. For example, it is usually risky from a security point of view to permit arbitrary unauthenticated (and in some cases unknown) computational entities to initiate or control highly trusted functions. Forcing communications to occur between comparable levels of trustworthiness and abstraction tends to diminish those risks considerably. Encapsulation of functionality within local boundaries is another important structural concept, permitting execution only within trusted and carefully controlled enclaves.

Two examples of the ease with which a well-structured conventional secure-system architecture can be transformed into a secure distributed architecture are given by considering PSOS and SeaView.

- With respect to the PSOS design, secure distribution could be implemented at any of several layers. For example, the capability mechanism could have been extended to support capabilities that were globally unique across multiple processors and multiple systems. The paging mechanism could have been extended to permit dynamic access to remotely stored segments. The directory hierarchy could have been extended to provide globally unique symbolic names across multiple file systems. The user-object layer could have been extended to support globally identifiable distributed objects. Each of these approaches has some advantages and some disadvantages, and the appropriate choices are dictated by the trustworthiness of the various system platforms.
- With respect to the SeaView design, it is conceptually simple to permit multilevel-secure TCB-to-TCB communications from one SeaView platform to another, thus retaining the in-

tegrity of the SeaView architecture with MLS-untrusted database management functionality. This could be achieved either explicitly (with the DBMS being aware of remote physical whereabouts of database objects) or implicitly (with the underlying TCBs keeping track of remote locations). However, in either case, the multilevel-security separation would be enforced solely by the underlying TCBs.

If the distribution is done implicitly (that is, virtualized), so that it is largely hidden by the relevant interface, then properties that are satisfied by the original nondistributed design can be transformed straightforwardly into equivalent properties in the distributed design.

Structural contributions to distributed systems are considered further in [164].

4.2 Security Requirements

We begin with a few intuitively motivated high-level structural security requirements, each of which implies lower-level functional requirements. The functional requirements are intended to be applicable to each of the subsystems — for example, file servers, authentication servers, and network servers. They in turn imply more detailed, finer-grained, requirements, which are considered in Section 7 — which also considers the implications of these requirements on applying formal methods to secure distributed systems.

The Overarching Security Requirement

- **Goal 1: Overall system security.** The entire system environment must have adequate security in the large, relating to the preservation of confidentiality, integrity, availability, authentication, accountability, and other security-related attributes. There must be no exploitable weak links in the overall system, the components, or the infrastructure.

Structurally Oriented Security Requirements

- **Goal 2: End-user-system security.** Each end-user system must have sufficient integrity to prevent its being compromised locally or remotely. Local data must be free from misuse by others. Any communications or interactions with the rest of the system environment must be free from loss of confidentiality and integrity. All users must be properly authenticated.
- **Goal 3: System-infrastructure and server security.** All servers (for example, network, file, and authentication servers) and other components of the infrastructure must protect the stored data, crypto and authentication keys, and transmitted audit data from browsing, alteration, deletion, and spoofing. All server operations must be properly authenticated.
- **Goal 4: Network-media security.** All network media must provide adequate protection against loss of confidentiality, loss of integrity, and denials of service. All network operations must be properly authenticated. (Whenever the media are untrustworthy, the infrastructure must compensate by means of encryption, fault tolerance, and other defensive techniques.)

Functional Requirements

- **Goal 5: System integrity.** The system and each of its component subsystems must be able to prevent or withstand intentional or accidental tampering. All subsystem components must be legitimate, and checked for authenticity, to hinder Trojan horses and other forms of tampering. Some subsystem confidentiality requirements are necessary to provide system integrity — for example, relating to sensitive algorithms and parameters used in configuration, authentication, and encryption (keys).
- **Goal 6: Data confidentiality.** Within systems and across networks, user data and internal data must be protected from loss of confidentiality (such as unauthorized or unintended disclosure).
- **Goal 7: Data integrity.** Within systems and across networks, user data and internal data must be protected from loss of integrity (such as unauthorized or unintended modification).
- **Goal 8: Nondenial of service.** System and user resources must be protected from intentional or accidental denials of service.
- **Goal 9: Authentication integrity.** Users, systems, subsystems, and other computational entities must have their identities unambiguously validated, with adequate certainty, based on sound associations of authentifying information with the entities in question (that is, based on their cryptobindings, discussed in Section 5.7. (There are occasional exceptions in which anonymity can be permitted.)
- **Goal 10: Nonrepudiation.** Authentication of (id)entities must be of sufficient credibility that a supposedly authenticated (id)entity cannot with any credibility later support a claim that the authentication had been erroneous or spoofed.
- **Goal 11: Accountability.** Monitoring, audit-trail analysis, and other accountability functions must be protected from misuse (including undesired bypassing, reading, modification, and denials of service). These functions must also not be able to interfere with the operations being monitored, via back-channel effects. All relevant operations must be monitored. All secondary subsystems that monitor behavior of the primary systems must protect the monitored data from misuse. The monitoring subsystems must not be able to interfere or otherwise adversely affect the operations of the primary monitoring systems. Similarly, passive subsystems whose purpose is to provide accountability for system operations must also be protected and isolated.

4.3 Preventing Misuse, Compromise, and Tampering

We next consider the protection of a system or subsystem environment — and how it can prevent compromises such as some of those illustrated in Table 5, whether the environment is a distributed system, a network, or a stand-alone encapsulated application.

The first line of defense involves ensuring that the security-critical system and network components are legitimate. If components have been tampered with, all other defenses may already have been compromised. Measures such as cryptographic checksums and tamperproof hardware encapsulation may be helpful, both statically and dynamically.

The second line of defense involves identification and authentication — of users, agents, subsystems, systems, network components, and other entities. Without authentication, access controls are almost meaningless, and monitoring is unable to gather enough evidence. Sensitive functionality that requires no authentication (such as a freely exported network file system) is inherently riskful. However, authentication may be compromisible unless the system components are tamper-resistant.

The next line of defense involves authorization, with selective access controls based on identities, or roles, or other attributes. Without authorization, system use and misuse cannot be controlled. Sensitive functionality that requires no authorization (such as a freely exported network file system) is inherently riskful. However, authorization is almost meaningless in the absence of nontrivial authentication.

Structural encapsulation (Section 2.6) throughout the system design provides an approach that can help to prevent misuse. It aids in preventing compromise from above, by sealing off the internals from the external accesses. In combination with authentication and authorization, it also aids in preventing compromises from within by blocking access to the internals. In the same way, it can aid in preventing compromises from below, again in combination with authentication and authorization, by blocking access to the underlying mechanisms. Because flaws in any structural encapsulation might enable compromises, it is essential that security-relevant functionality be properly encapsulated.

Thus, it is vital that authentication, authorization, and structural encapsulation be used consistently throughout a system or network environment. It is additionally essential that the protection mechanisms and encapsulation specified in a system design must be strictly enforced by the implementation. However, additional measures are also necessary, relating to the integrity of the implementation, particularly those that hinder would-be attackers and malicious insiders.

Tampering may result from penetrations that led to code patches, data modifications, or other subversions. It may be preceded by browsing or reverse engineering that unveils certain details that can be used in subsequent tampering attacks. Either case may be critical in any particular operating environment. The threats may arise from authorized users and from unauthorized outsiders.

Compromises that do not require modification must also be prevented. Once again, structural encapsulation is essential, together with information hiding, authentication, and authorization.

The following section provides some of the basic security requirements that characterize defenses against compromise.

5 Minimizing Trustedness Within Multilevel-Secure Systems

Section 5 considers the problems involved in developing and using systems and networks that must operate under the constraints of multilevel security, and how the RISSC concept can contribute. Section 5.1 considers the RISSC philosophy. Section 5.2 examines the advantages that can result. Section 5.3 discusses some RISSC architectures. Sections 5.4, 5.5, and 5.6 elaborate on various practical considerations. Section 5.7 considers the application of the RISSC approach to implementations of crypto.

The original PSOS architecture considered two possibilities: either providing trusted multilevel-secure object managers at higher layers, or alternatively embedding multilevel-security enforcement into the hardware capability mechanism at the lowest layer. Although the assurance of the higher-layer approach of such an implementation would thus depend upon the lower layers, the intent was that if the lower layers were largely in hardware and proven to be correct, then the multilevel security would have high assurance. The assurance of the lowest-layer approach would be more directly in line with the RISSC philosophy, because the MLS enforcement would be primitive to the entire system. However, if the lower layers are implemented in trustworthy hardware, the RISSC approach with respect to the software would be equally well served in both cases.

The SeaView architecture relies on the presence of a multilevel-security kernel and a low-level trusted computing base — that is, it corresponds to the Orange Book paradigm in which there is a trusted computing base that enforces multilevel security, but where the database management system is explicitly not a part of the trusted computing base for multilevel security.

Here, we consider an alternative approach to that represented by the monolithic kernel and TCB of the TCSEC Orange Book. In this approach there are no multilevel-secure end-user systems; multilevel security is assured by means of multilevel servers.

A basic need exists for commercially available computer systems that can be securely networked. Conventional (single-level) systems and networks have not been adequately fulfilling this need, because of weak operating systems and weak networking software. However, some improvements are occurring.

Primarily in classified environments, there are serious needs for systems and networks that pervasively enforce multilevel security. However, there have been serious problems in developing, evaluating, and continually modifying these systems. Relatively few high-end MLS systems exist today that are consistent with the established criteria, and those criteria are themselves incomplete with respect to networking and distributed systems. Evaluation and reevaluation have been enormous stumbling blocks to the availability of MLS systems.

We characterize here a class of multilevel-secure systems in which multilevel security can be achieved without having to rely on MLS end-user systems. Instead, we rely upon single-level end-user systems and a few trusted multilevel servers and network components.

Ideally, multilevel-secure application environments should be configurable as interconnections of readily available commercial hardware, software, and networks. Unfortunately, the ability to construct a multilevel-secure system from existing components and networks is hindered by several serious obstacles.

- There is a dearth of suitably secure commercial operating systems, applications, and network components, especially with respect to multilevel security. Even in single-level systems, commercial offerings have serious security vulnerabilities. In addition, most existing systems lack adequately secure interconnectability and interoperability. Each vendor has taken its own approach, relatively independent of the others — although a few common efforts (such as OSF) may ultimately emerge.

- There is very little real understanding as to how security can be attained by integrating a collection of components, and even less understanding as to what assurance that security might provide.

- Vendors are discouraged from offering secure systems because significant time and effort are required to develop a system capable of meeting the TCSEC evaluation criteria and to marshall it through the evaluation process. Moreover, because of evaluation delays, an evaluated product is typically no longer the current version of the system, which necessitates repeated reevaluation. For high-assurance systems, the difficulties of using formal methods add further complexity to both development and evaluation. Export controls, lack of awareness, and lack of customer demands tend to provide further discouragement.

Because of these obstacles, and the resulting lack of viable high-assurance multilevel systems, less secure systems are commonly used to store highly sensitive data — despite considerable risk. (A compendium of vulnerabilities, threats, risks, and illustrative cases related to computer and communication systems is given in [159].)

Many of the hopes that appeared attainable by the security community in the 1980s have not yet been adequately fulfilled. We believe this shortfall is due to a variety of causes. For example, customer demand for MLS systems has been limited to the DoD sector. The evaluation criteria have emphasized architectural approaches that rely on monolithic, nondistributed multilevel-security kernels and trusted computing bases, with much less emphasis on end-to-end security as in electronic mail, digital commerce, and other applications heavily dependent on secure networking. Few R&D efforts have dealt comprehensively with real systems, particularly where formal methods are concerned. Inherent difficulties in dealing with formal methods have also contributed. U.S. export controls on crypto and high-end operating systems have also provided a damper in the eyes of system vendors. All these factors seem to have considerably increased the complexity of system development and evaluation, disenchanted the system developers, and retarded progress.

We propose an approach that is aimed at minimizing these obstacles while still attaining multilevel security, with high assurance, and providing more functionality than is achievable with traditional approaches.

5.1 The RISSC Philosophy

In an earlier paper [192] (reproduced here in Appendix C), Proctor and Neumann characterized an architectural family of multilevel-secure networked systems in which the use of multilevel-secure end-user systems can be avoided altogether, with reliance instead on multilevel-secure network interface devices and possibly other multilevel servers. The primary consequence of this type of architecture is that the existing infrastructure of single-level end-user systems can immediately be incorporated. A secondary consequence of such an architecture is that user-exploitable covert channels can be eliminated altogether.[10]

This report extends the Proctor-Neumann concept. We advocate the construction of multilevel systems using only a few carefully controlled multilevel components. We explicitly address the issue of what portions of the overall system must be trusted with respect to multilevel security, and conclude that this approach greatly reduces the need for trust. The RISSC concept is also applicable to single-level systems, whereby the security-dependent functionality has been appropriately isolated and constrained. (See Section 5.6.)

The 1992 paper by Proctor and Neumann [192] refers to **DSM** — an acronym for Distributed, Single-level-user-processor, Multilevel-secure — systems. This report generalizes the DSM approach, and refers to the RISSC philosophy — where RISSC (as noted above) is an acronym for Reduced Interfaces for Secure System Components. The RISSC philosophy is similar in concept to

[10] A covert channel is an out-of-band means of communication through which it is possible to signal information — for example, based on the occurrence of exception conditions triggered by resource exhaustions. For example, see [61, 189, 94, 251, 80] for analysis of covert channels, and [106] for avoiding them.

the RISC philosophy (Reduced Instruction Set Computer), in seeking simplicity of mechanism. The RISSC philosophy strives to localize the security-critical functionality within components that are not directly accessible to users. The RISSC philosophy is conducive to multilevel-secure systems that can be configured quickly and economically out of conventional single-level end-user systems, without losing any essential functionality. It is primarily an architectural concept, but it also relies heavily on sensible design efforts, good software-engineering practice, good system implementation, and good analysis. It is particularly amenable to formal specification and formal analysis, as discussed in Section 6.

DSM architectures are one example of systems satisfying the RISSC approach, and are themselves a logical successor to an earlier effort by Rushby and Randell, the Newcastle Distributed Secure System (DSS) [210]. In the Newcastle DSS, trusted network interface units are added to standard operating systems that are untrusted with respect to multilevel security. Rushby's notion of a separation kernel (e.g., [203, 205]) is another instance of RISSC. The RISSC philosophy is compatible with client-server architectures in which the servers are suitably trusted but the clients are not, and therefore it can take advantage of existing work on Kerberos [14, 236] and the Digital Distributed System Security Architecture [68] — although even stronger network authentication will be desirable in the future.

5.2 Advantages of the RISSC Philosophy

Many potential advantages are offered by RISSC. Systems can be fully functional and multilevel secure. Because an end-user workstation can efficiently read information at its level and below from network servers, a user can integrate data from a variety of classifications. Advisory labels can be provided in single-level end-user systems, as is the case in existing low-assurance compartmented-mode workstations (CMWs). Because end-user systems are single-level, they can have full commercial functionality, rather than having to be ported to the environment of a Trusted Computing Base. This means that standard conventional, commercial-off-the-shelf (COTS) software and existing applications can be used without change for end-user systems.

System security and integrity can be increased because the reliance on the security of end-user operating systems is minimized. No ordinary users have direct access to the multilevel-secure components. Tampering with end-user systems cannot compromise multilevel security.

The evaluation effort for the entire system with respect to multilevel security is reduced to evaluating the multilevel components and their interactions with the rest of the system. It is largely independent of the internals of the single-level components. Thus, the evaluation effort can focus on the network interfaces and servers. It is easier to identify and monitor any residual covert channels that arise because of the multilevel networking.

The reevaluation effort is also vastly reduced. Once an overall system has been evaluated and certified, upgraded versions of component end-user operating systems can be installed without any reevaluation for multilevel security, just as there was no such evaluation in the first place.

Thus, the RISSC philosophy has the potential for greatly increasing the ease of developing, evaluating, and procuring multilevel-secure systems when contrasted with what is experienced with the present TCSEC B-level evaluation process today.

5.3 Alternative RISSC Architectures

In a precursor to this study, several architectural families were outlined that represent instances the RISSC concept. To provide a framework for considering a variety of architectural types, the families enumerated in [158] are summarized in Appendix A, with some modifications and extensions. (This report therefore supersedes that earlier categorization.)

The families of primary interest here are those that have explicit relevance to the RISSC concept. For example, the subfamilies [[1a]], [[2a]], and [[3a]][11] all have single-level single-end-user workstations and multilevel-secure network servers. Within those families, there are two basic alternatives — one ([[1a]] in Section A.1 and [[2a]] in Section A.2) with single-level file servers and the other ([[3a]] in Section A.3) with multilevel-secure file servers; the systems of family [[1]] have files stored unencrypted, while those of family [[2]] have files stored encrypted. In a very rough sense (considered in Appendix A), these various alternatives can be considered as equivalent implementations of the same conceptual design; trustworthiness is located somewhat differently, but the functionality that must be trusted is roughly comparable.

To a lesser extent, several architectures that support multiple coexisting but competitive end users ([[1b]], [[2c]], [[2d]], and [[3b]]) also have RISSC-like attributes, although the non-MLS security perimeters are larger in those cases. In particular, greater reliance must be placed on the local end-user operating systems, because of the presence of potentially competing users. However, in other respects the security considerations are similar to the noncompetitive-user situation.

Several of these subfamilies are considered in greater detail in Section 11.

5.4 Realities of Multilevel Security

We next consider some of the practical implications of system architecture, first with respect to conventional multilevel-secure system designs and then with respect to RISSC architectures.

Multilevel security has not been considered with uniform favor by the rest of the world outside of the U.S. DoD — and in some cases, not even within the DoD. Air Force Lt. Gen. Carl O'Berry (U.S. Air Force deputy chief of staff for command, control, communications, and computers) was quoted[12] as saying that multilevel security is "a brain-dead idea based on the assumption that you can take responsibility for information security off the shoulders of man and put it in a machine." Also, he asserted that DoD has spent "billions upon billions" of dollars in trying to get MLS systems accredited, but he doesn't believe the process has served DoD well. His view is reported to be that "InfoSec should focus on data transmission, user authentication, and risk assessment of transmitted data."

There are several reasons why efforts to make multilevel-secure systems more widely available have not been more successful.

- Architectures adhering rigidly to the TCSEC kernel-TCB philosophy are inherently inflexible.
- The MLS requirements are only a small part of what is desirable in commercial systems. The TCSEC MLS requirements have been overendowed — both in the development process and

[11] Double square brackets are used to identify classes of architecture family members. See Appendix A for the notation that is used.

[12] Article by Paul Constance, *Government Computer News,* 14 April 1995, p. 3.

as a focus of the research community — at the expense of real user needs.

- Customer demand has not been sufficient to justify the development effort and associated costs of MLS systems as mainline commercial offerings. There is considerable reluctance inertia acting against multilevel security.

- MLS systems are difficult to evaluate, particularly at B2 and above, and the evaluation process has great difficulty keeping up with incremental system changes.

The fact remains that multilevel security is important for DoD use, and possibly useful for non-DoD use as well (for example, see [123]). Thus, we need to consider alternative MLS architectures that can overcome the difficulties enumerated here.

5.5 Potentials of MLS RISSC Systems

If a RISSC system has no multilevel-secure end-user systems, the existing TCSEC criteria for B-level systems are not directly sufficient to constrain the multilevel security of the overall system — although the principles underlying the TCSEC [151] and the TNI [150] are valid when an entire network of systems is considered as a single system. Clearly, the C2 requirements must be extended to include some aspects of trusted system distribution and trusted paths (for example); however, those extended C2 requirements would then be appropriate for the user systems out of which such RISSC systems could be built. We believe that the TNI B-level criteria will be directly applicable to (but also incomplete for) the trusted multilevel-secure network software that is at the heart of the RISSC concept, although those criteria will need to be extended (just as they need to be extended to accommodate distributed systems anyway).

Each RISSC trusted server is essentially a trusted computing base. However, because it has no end users *per se,* it can be much simpler than that required for a multilevel-secure end-user operating system. For example, file servers can be isolated as systems whose only direct users are system agents. The trusted network interface is the primary component mediating between each end-user system and other systems, but its role is reduced to that of a filter, gateway, or firewall (in software and/or hardware), according to the functionality required. The TNI must ensure that imported and exported data entities are at appropriate levels commensurate with the source and destination, and that channel switching is done consistently with the multilevel constraints. Authentication and key management help assure the integrity of the networking.

In the RISSC concept, trust for enforcing multilevel security resides solely in the multilevel servers, and not in any end-user systems. Nevertheless, there are certain functionalities in the end-user systems that must be noncircumventable, such as the protection of any stored encryption keys and the integrity of the authentication process, in part through hardware or software encapsulation. A challenge of any specific architecture is to minimize that need for trust.

The RISSC philosophy encompasses various options, the choices among which involve consideration of the intended application environment. The strict DSM approach avoids end-user-created covert channels altogether at the cost of separate system components for each security level and compartment, as analyzed in [164, 192]. This approach may be appropriate for certain applications but not for others. On the other hand, hybrid strategies may be able to provide better performance in certain selected cases in which the exploitation of covert channels can be prevented or else monitored by other techniques. Other trade-offs are also possible within the RISSC philosophy — for example, whether there are multilevel-secure servers or multilevel media. There are also trade-offs between multiple single-level servers or multilevel servers — for example, for name servers.

If the strict-sense DSM architecture is used, or if compartmented-mode workstation end-user nodes can be avoided, the covert channels can be reduced. However, in certain RISSC environments, CMWs could be used without introducing covert channels. That could be the case in a SeaView environment [128] (noted in Section 3.3) in which a security officer needs to see both a Top Secret view and a Secret view of the same data query — for example, in attempting to ensure that less classified cover stories are sensible and consistent.

As an example of the effectiveness of the RISSC concept in the context of an application subsystem, consider SeaView. By placing an off-the-shelf database management system (Oracle) on top of a multilevel-secure trusted computing base, and completely encapsulating the TCB to prohibit all external accesses to the TCB interface, a multilevel-secure DBMS can be implemented in which there is no need to trust the database software with respect to multilevel security. Analysis shows that the assurance with which multilevel security can be provided by SeaView is the same as that provided by the TCB. (See [191].)

Furthermore, it is possible to develop an overall distributed-system/network concept in which certain security flaws in the component operating systems can be tolerated, while still achieving security in the large. This is somewhat analogous to the design of fault-tolerant and Byzantine subsystems. In addition, we believe it is possible to support various operating-system interfaces within the RISSC concept, such as Unix, POSIX, and Mach, as long as the networking interfaces can be suitably trustworthy.

5.6 Potentials of non-MLS RISSC Systems

The RISSC concept is of course also valuable in conventional single-level systems and in hybrid single-level systems that enforce compartmented separation within the single level.[13] In the present context, we consider single-level compartmented systems to be a special case of single-level systems rather than a degenerate case of multilevel-secure systems, because the absence of multilevel requirements for files and network interfaces can significantly simplify the architecture — particularly when ultra-high assurance (e.g., A1) is not required. (This special-case association also makes sense linguistically, because those hybrid systems are, strictly speaking, not *multilevel* secure.) However, in practice, if high assurance is necessary, it is clearly better to dumb down a multilevel-secure system to a single level than it is to add compartments to a conventional system — primarily because many of the additional requirements would not be met.

In single-level systems, the RISSC concept is an architectural concept as well as an embodiment of good software-engineering practice. It requires strict modularization and encapsulation of security-critical functionality in such a way that the interfaces to that functionality can be completely controlled. It also requires that the interconnections among different modules cannot compromise security, through replays, spoofing, or other attacks. The RISSC concept is advantageous whether or not the systems are distributed. However, it is particularly relevant to distributed systems, in which the notion of isolating functionality within trusted servers (see Section 5.3) can contribute

[13] As an example of such a hybrid system, the Internal Revenue Service is contemplating the use of single-level compartmented servers throughout its Tax Systems Modernization program, with fine-grain access controls, in an effort to minimize undesirable browsing and insider misuse.

significantly to single-level systems as well as multilevel-secure systems. Indeed, most of the architectural issues in Section 5.3 apply directly to single-level systems, compartmented or otherwise; the trusted servers still need to be trustworthy in some respects, even if they are not trusted for maintaining multilevel-security separation.

We note in passing that Jagannathan's GLU ([97, 96]) could be used to help partition processing that has natural parallelism. GLU consists of a methodology and preprogramming language. It facilitates dynamic virtual multiprocessing, flexible system configuration that can adapt to whatever hardware is available, dynamically alterable fault tolerance, and distributed computation, while placing very little burden on the programmer. GLU statements are preprocessed into C, which after compilation can result in virtualized parallel tasks being dynamically allocated to whatever real resources are available. Appropriate choice of GLU statements could be used to enhance the RISSC nature of the implementation by partitioning functionality according to security constraints.

5.7 RISSC Applied to Crypto Implementations

One of the trickiest aspects of computer system architectures involves the encapsulation of crypto in such a way that the security provided by the crypto cannot be compromised — despite the presence of not-entirely-trustworthy system components. The potential risks are quite numerous.

In the following text, a careful distinction is made between the process of creating an object that can later be authenticated and the subsequent process of validating that the object is indeed authentic. The former process is generically referred to here as a *cryptobinding*, and the latter process is referred to as *authentication*. In other contexts in which this distinction is not important, "authentication" more conventionally includes both the cryptobinding and the validation of the cryptobinding.

In general, the cryptobinding function needs to have a sufficiently strong cryptographic basis that it can be nonsubvertible under malicious attack. (Although in principle the binding could be noncryptographic, as in the case of a cyclic redundancy checksum, we nevertheless refer to the generic function of creating the binding as cryptobinding.)

- Loss of confidentiality — for example, due to key compromise that makes the crypto ineffective, cryptanalysis that breaks the crypto, or bypasses of the crypto that undermine the crypto without requiring key compromise or cryptanalysis (as in penetrations of underlying operating systems)
- Loss of integrity — for example, due to changes in cryptotext or in the cryptobound information that is subsequently expected to be decrypted or authenticated
- Denials of service — for example, due to intentional or accidental losses of integrity
- Loss of authenticity — for example, due to spoofing of the cryptobinding process (for example, creating a bogus digital signature or integrity seal), disabling the process of checking authenticity, or conducting a replay attack
- Repudiation — that is, a claim that seemingly legitimate cryptobinding had actually been erroneous, spoofed, or otherwise compromised
- Loss of key-escrow confidentiality — for example, due to breaches of the key-escrow process

- Loss of key-escrow integrity — for example, avoiding key escrow altogether, breaking the *escrow binding* between keys escrowed and keys actually used, using crypto implementations for which the keys do not need to be escrowed, law-enforcement access-field attacks
- Loss of audit confidentiality — for example, undesired accesses to an encrypted audit trail
- Loss of audit integrity — for example, accidental or intentional modification of an encrypted audit trail

The modularity implied by the RISSC concept can aid considerably in avoiding these risks — if the modules are sufficiently noncompromisible.

A particular additional challenge is provided by the desire for high-assurance implementations that can use modular crypto. At present, United States export-control restrictions seriously discourage the use of modular crypto, sometimes misleadingly known as crypto-with-a-hole (although the hole is in the surrounding system, not in the crypto).

The RISSC concept can provide considerable assistance in ensuring that modular crypto cannot be subverted, bypassed, or removed for other purposes, if it is possible to encapsulate the crypto within trustworthy subsystems. This can be achieved by isolating the crypto into separate modules (whether they are implemented in hardware or in software). Some of the opposition to modular crypto implicit in the export controls may be diminished if assurances can be provided that the modular crypto cannot be removed, replaced, tampered with, bypassed, or otherwise compromised. One way of accomplishing that goal is to provide a trustworthy cryptographic/cryptologic means of ensuring that a given module is precisely in every respect one of a legitimate set of previously authorized modules — for example, by applying serious cryptographic integrity checks within trustworthy components. In this way, it would be possible for a vendor to provide multiple versions of a computer operating system or application software, with different crypto implementations that could be adapted to the nature of the application and that could conform with any external restrictions such as export controls.

At present, system developers are reluctant to provide multiple versions of software systems, because of perceived difficulties in production and maintenance. Such an approach would sharply diminish the difficulties involved.[14]

Similarly, encapsulation within the RISSC concept could also enhance the assurance with which certain usage restrictions could be accommodated — for example, requiring the presence of non-subvertible key escrow on a crypto implementation.

In all crypto implementations and, more generally, in all system implementations, the risks must be rigorously analyzed from a total systems point of view — including hardware platforms, operating systems, application software, networking software, network media, supposedly untrusted components that in fact are able to acquire unanticipated privileges, and supposedly trustworthy components that are not trustworthy, as well as external attacks such as masquerading and spoofing. Crypto algorithms, protocols, and implementations are particularly fertile topics for formal analysis of security. This concept is considered further in Section 11.3.

[14] For example, two versions of RC-4 are used in the first commercial version of Netscape, one with long keys (128 bits) and the other with shorter keys (40 bits). As an interesting implementation quirk, the 40-bit version is achieved by passing 88 bits of the 128-bit key along with the message, but without changing the implementation. However, 40-bit crypto is becoming unacceptable in commercial applications, as evidenced by recent cracking of 40-bit RC-4 within Netscape.

Part Two: FORMAL METHODS

6 Formal Methods Applicable to Secure-System Architectures

Section 6 considers the applicability of formal methods to architectures for secure systems.

Much of the work to date in applications of formal methods has emphasized the specification of relatively low-level functionality. A useful cross-section of the early history of formal methods applied to computer systems can be found in the proceedings of the first three verification workshops (*VERkshops*), the first two of which [165, 166] were coorganized by Peter Neumann and Steve Walker (April 1980 and April 1981), and the third [117] was coorganized by Karl Levitt, Steve Crocker, and Dan Craigen (February 1985). Because the proceedings of these VERkshops are not widely available, and because of their historical importance to the evolution of formal methods applied to security, we include a list of the VERKshop contributions in Appendix D.

Although there is some work on properties of systems in the large, and on the specification of complex systems (as in HDM/EHDM and the CLI work), much of the effort in the past has been devoted to properties of algorithms and components in the small.

The present study is much more concerned with specifications and properties of systems in the large, with the fundamental belief that there is enormous potential to be obtained from the composition of systems and networks out of system building blocks — especially if that functionality can be related to its components and their properties in a formal manner. Thus, we stress hierarchical structures, vertical abstractions, horizontal compositions of distributed components, and configuration management.

6.1 Goals of Formal Methods in System and Network Architecture

The need to apply formal methods to the architectures of supposedly secure systems and networks suggests the following set of desirable goals. These goals are generally commonly perceived goals that have been either implicit or explicit for many years, although many of them are still not easily achieved. Thus, we believe that it is useful to articulate these goals coherently within the global distributed-system scope of this report. The recommendations in Section 12 consider what is needed to satisy these goals more fully.

- Formalisms should be readily understandable by appropriate people (designers, system implementers, interface users who are developing higher layers, system maintainers, and documenters) at a level of detail appropriate to their individual needs. That is, designers should be able to write specifications that are sufficiently detailed and unambiguous to permit sound implementation by programmers and hardware fabricators who have not been a part of the design process; those specifications should also be viewable at varying layers of abstraction — for example, by someone explicitly wishing to suppress certain types of detail, or alternatively, someone wishing to have particular upper-layer constructs explicitly expanded in terms of lower-layer functionality.

- Formalisms should be realistically applicable to complex and practical systems, including operating systems, special-purpose systems such as database management systems, firewalls and other types of gateways, network controllers, file servers, encryption control units, cryptologic protocols, and critical application environments. Furthermore, such formalisms should also be applicable to the specification and analysis of distributed systems and networks formed from compositions of components, whereby the analysis of the overall system can be based on properties derived from the analysis of the components. Formalisms should provide the ability to handle complexity in its totality — for example, reflecting all possible exceptions and other abnormal conditions. The formalisms should scale well, from simple systems to highly interconnected complex systems. Difficult concepts as well as simple ones should be capable of being clearly explicated.

- Formalisms should contribute significantly to the entire system life cycle — for example, enhancing the requirement formulation, analyses of the requirements, the design process, analyses of the design, the implementation, analyses of the implementation, and the management of the entire development. The benefits of these contributions might include tighter management of the development process, sharply reduced production costs, fewer scheduling delays, and improved satisfaction of the stated requirements — with greater reliability and greater assurance of proper system behavior.

- The formalisms should be particularly helpful at the initial stages of the development cycle. In general, many system flaws occur during the design phase rather than during implementation; the cost of repairing serious design flaws after a system has been fielded is often orders of magnitude greater than the comparable cost for repairing code flaws. As a consequence, the greatest payoffs from formal analysis are typically found up-front: formalizing requirements, and analyzing the adequacy, completeness, and self-consistency of those requirements; attempting to demonstrate the consistency of specifications with requirements, and identifying design flaws; carrying out explorations of incremental design alternatives, and being able to iterate on the successive analysis of improved designs. All these attributes are highly desirable.

- Formalisms should be applicable at different layers of abstraction and different types of abstraction, from hardware to operating systems to database management systems to application software, and throughout distributed systems. Formalisms should facilitate vertical (hierarchical) abstraction, relating each layer to its neighboring layers (either less abstract or more abstract). They should also facilitate horizontal abstraction, allowing for consistent coexistence among different interoperating distributed-system and networked components, and for the transversal of any gateways or other network subsystems.

- Formalisms should be able to facilitate rigorous analyses, interrelating with one another whenever different formalisms are used or must be used in different contexts. For example, a particular system analysis might employ a requirements language, a module specification language for operating-system and application functionalities, mathematical techniques for analyzing the strength of cryptographic algorithms, and BAN logic 6.3 for reasoning about cryptologic protocols. It should be possible for different analyses to be logically linked, interrelating requirements analyses, design proofs and other specification analyses, and implementation proof techniques for determining the consistency of code (or hardware) with its specifications. The formalisms used should be substantively interoperable — in the sense that reasoning about properties at one layer or component must be able to draw on assumptions, properties, and inferences about other formalisms of depended-on functionality. Thus, expressions in one formalism should be transformable or otherwise representable in other formalisms, wherever appropriate. In addition, properties of fault tolerance and real-time performance should be able to accommodated, as relevant to security.

- Formalisms should be able to facilitate analytic and synthetic transformations, involving both vertical and horizontal abstraction. For example, analysis of a composition of the specifications of multiple modules into a single module specification should be formally straightforward. The same is true for decomposition of a subsystem into separate modules, for example, so that a component can be used separately elsewhere. Similarly, the resulting behavioral properties resulting from composition or decomposition should be formally derivable, whether the properties in question are invariant or transformed into other properties.

- Formalisms at certain higher levels of abstraction should be relatively independent of specific programming languages and language constructs, but at lower levels should be capable of explicitly and completely representing programming-language constructs, hardware instructions, and microcode.

- Formalisms should be broader and richer than just those relating to a few specific security properties at a particular layer of abstraction. As appropriate to a particular application, other relevant requirements should be addressed, such as application-layer properties relating to interface completeness, ease of use, overall system reliability and fault tolerance, and system safety.

Some of the past work is applicable to computer systems in general, although distributed systems have been slighted until recently. Other work is specific to particular types of formalisms, such as the BAN logic [29] for crypto protocols. Overall, it is clear that no single approach is sufficient for specifying and reasoning about distributed systems in the large, and that multiple formalisms must be used. An important open question is how to choose the different formalisms that are necessary for a particular application and to integrate them effectively.

6.2 SRI's Computer Science Laboratory: HDM, EHDM, and PVS

The SRI Hierarchical Development Methodology (HDM) (see [161, 199, 201, 231], plus an earlier reference [200] for historical interest) was an early effort to unify formal methods within a single framework that would be applicable to real systems, in hardware and software. HDM embodied a first-order SPECIfication and Assertion Language (SPECIAL) [233] for specifying modules at each hierarchical layer, and an abstract Intermediate-Level Programming Language (ILPL) [161] whose pseudoprograms represented explicit mappings between hierarchical layers — acting as abstract implementations for each layer written in terms of lower-layer functionality. In addition to hierarchically layered abstractions, HDM encompassed staged development and module refinements, extending from requirements to abstract design to abstract implementation to actual implementation. In concept, HDM also supported analysis techniques associated with each development stage — namely, requirements analysis, design proofs, abstract code proofs, and real code verification. The HDM effort began in 1973 under the PSOS project, in which it was used extensively. HDM is no longer supported, but it pioneered several important concepts.

HDM tools included a syntactic specification checker and the Boyer-Moore Theorem Prover [26, 27][15] — which was used for both design proofs that specifications are consistent with their formally stated requirements and for implementation proofs that code is consistent with its specification. For example, for any given module specification, the Feiertag flow-analysis tool [61] identified all potential information flows through that module and then formulated putative theorems whose satisfaction would imply no adverse multilevel-security flow; the Boyer-Moore theorem prover was then invoked to determine whether or not the putative theorems were indeed valid. If they were valid, the flows satisfied the multilevel-security properties; if they were not valid, flaws in the specification or the design were identified, including covert storage channels. (Actually, only a fraction of the logical abilities of the Boyer-Moore theorem prover were required, and a simpler tool could have sufficed.) As with most verification work, the benefits arise largely in the identification of flaws rather in the demonstration that everything is perfect. This approach was applied to several multilevel-security projects, including the kernelized Unix system KSOS [133, 16]. Out of 34 kernel functions in the initial KSOS kernel design, 16 were shown to have some sort of security problem, many of which could be and were fixed; however, some of the covert storage channels that were identified were not removed.

One of the important advances of HDM was its ability to take a group of modules specified in SPECIAL, and arrange them either hierarchically or coexisting at a given layer, and then relate those modules to one another using the abstract programming language ILPL [199]. It was also possible to constrain precisely which higher layers could have access to which functions at lower layers. In this way a particular functionality could be accessed only up to a particular layer, and its functionality could then be replaced by a more abstract equivalent. This type of accessibility constraint is demonstrated by the square-bracketed level numbers in Table 6 — an example of which is given by the system processes at level 6 and user processes at level 12.

SRI's EHDM [204] and its higher-order specification language [234, 235, 245, 246, 247] provides a facility for hierarchical layering comparable to that in HDM. EHDM provides a stronger subtype mechanism than HDM, and also introduced module parameterization. It also had an improved approach to the axiomatization of theories and subtheories, and their interrelations. The EHDM effort began in 1981, under a project that was primarily concerned with investigating the feasibility of implementing PSOS.[16]

SRI's Prototype Verification System, PVS, is the intended successor to the verification tools used in EHDM. Its specification language has a still richer type and subtype facility than that of EHDM (including dependent types). Model checking (see Section 9) is compatibly being included in version 2. At present, PVS (in version 1 and in the soon-to-be-released version 2) does not yet provide support for hierarchical design validation and program verification that are found in EHDM, although a new approach to that capability is planned for PVS version 3, along with an extension of the EHDM facility for interpretation of models and theories. On the other hand, the PVS verification environment does support interactive proofs — which EHDM's prover does not. Furthermore, the specifications, proofs, and proof process of PVS appear to be significantly more comprehensible and easy to develop than is the case for EHDM.

All three of these environments (HDM, EHDM, and PVS) provide tight coupling between the

[15] See Donald MacKenzie's 1995 Annals article [129] for an excellent review of the history and sociological role of theorem proving, with 185 references.

[16] EHDM was originally so named because it began as an "Enhanced Hierarchical Development Methodology." That acronym expansion is no longer relevant, and it is now named simply EHDM; its specification language began life as Revised SPECIAL [233].

specifications and the verification systems, and permit early formal analysis of specifications as well as formal code proofs. Since 1973, this sequence of SRI efforts has been explicitly aimed at spanning the entire system-development cycle, with particular emphasis on effective up-front analysis, applicability to large and complex systems, and long-term evolvability. A broad spectrum of techniques is incorporated, including syntactic specification checking, strong type checking, some ability to provide local executability of specifications, model checking, and various other forms of semantic analysis such as forward incremental closure (reachability with respect to descendents) and backward incremental closure (reachability with respect to ancestors).

A quote from Owre et al. [173] is appropriate to put these approaches in perspective:

> It is not easy to directly compare EHDM and PVS with other approaches to formal methods, such as those embodied in Z and VDM notations, or the Boyer-Moore theorem prover, since they are based on very different foundations. The HOL system is based on similar foundations to EHDM and PVS, but its language, proof-checker, and environment are much more austere than those of our systems. Over several years of experimentation, we have found that our specification languages have permitted concise and perspicuous treatments of all the examples we have tried, and that the PVS theorem prover, in particular, is a more productive instrument than others we have used.

In the short term, some of the considerable advantages of PVS exist as a result of PVS's temporarily eschewing the abilities of HDM/EHDM with respect to hierarchical layering; the comparative appropriateness of PVS and EHDM must be considered specifically with respect to system-wide architectural applications in the large. In the long term, PVS is expected to be more appropriate in almost all cases.

The PSOS specification [161] gives an extensive example of how HDM can be used to provide detailed specifications for a fairly complex hierarchically layered centralized operating system, in hardware and software. Various examples of the use of EHDM and PVS are given in [173]. Although these examples are relatively small, EHDM is at least as appropriate for large and complex systems as HDM was. EHDM satisfies all the goals enumerated in Section 6.1 to a considerable extent.

The reader interested in pursuing PVS further should consider an elementary tutorial by Butler [30] and a less elementary tutorial by Rushby and Stringer-Calvert [212].

Another SRI Computer Science Laboratory effort, due to Moriconi, Qian, and Riemenschneider [145, 146], is also relevant to specifying the relationships among hierarchical abstractions. The MQR approach provides a method for the stepwise refinement of an abstract architecture into a (relatively) correct lower-layer architecture. Once a particular refinement pattern has been proved correct, instances of that pattern can then be used directly without further proof. Refinements can themselves be syntactically composed. At present, this approach is not integrated with EHDM or PVS, although it appears that such a union could be very useful — in proving a priori that refinements are correct and in then relating the refined architectures to their implementations. In this way, the MQR approach could be used to reason about the security of an architecture in the large, and PVS could be used to demonstrate the required properties of the components.

The SRI Computer Science Laboratory WorldWideWeb pages (http://www.csl.sri.com/) provide further background and access to on-line reports.

6.3 BAN Logic and Related Reasoning about Crypto Protocols

Efforts to formalize reasoning about cryptologic protocols include the Burrows, Abadi and Needham's BAN logic [29] and its augmentations (see Wenbo Mao [132]) — with some important intermediate work by Gong, Needham and Yahalom [75], Abadi, and van Oostrup, among others. All the work to date is lacking in one respect or another, for example, being based on unprovable assumptions of key secrecy, or suffering from incompletenesses in the logic. (For example, see Gligor et al. [70, 237].) In addition, prudent practice for cryptographic protocols such as that suggested by Abadi and Needham [3] must be enforced. In any event, some sort of logic is essential for reasoning about cryptographic protocols and how they interrelate with the systems that encapsulate them. The use of formal techniques for cryptographic protocols is revisited in Section 8.6.

6.4 Other Approaches Relating to Security

We do not attempt here to provide comprehensive coverage of all of the various methodologies and supporting tools. However, it is worth noting that there are many different approaches, each of which has its own merits. We regret that we have not been able to track all recent developments in other approaches.

Computational Logic Inc. has been pursuing its specification-programming languages Gypsy and Rose, with formal methods applied to security and other application areas for almost as long as SRI. The Boyer-Moore theorem-proving tools are now an integral part of the CLI environment. Relevant references include documents on Gypsy [76, 77, 230], its flow analyzer [138], a code generator [253], and applications [18, 78, 88].

The CLI WorldWideWeb pages (http://www.cli.com/) provide further background and copies of on-line reports.

Odyssey Research Corporation's Romulus and Penelope provide similar capabilities. For example, formal methods have been applied to the Theta security kernel [168, 169, 170, 226].

Odyssey's WorldWideWeb pages (http://www.oracorp.com/) were still under construction when this report was completed.

A very significant recent effort to formalize security requirements is given by Johnson et al. [101], in which detailed formal specifications are provided for the MISSI security policy. This effort embodies many of the recommendations made here, including a layering of policy elements and an architecturally comprehensive approach that encompasses authentication and other infrastructure funcationality as well as conventional multilevel security.

7 Formal Methods Applied to Secure Distributed Systems

Section 4.2 summarizes high-level requirements for distributed-system security. These requirements are now reinterpreted in Section 7.1. The implications of attempting to apply formal methods to the elaboration of those requirements are then considered in Section 10.

7.1 General Properties of Distributed Systems

All the requirements outlined in Section 4.2 are directly applicable to distributed systems and to RISSC systems in particular. We reinterpret them here with greater specificity, and consider how to approach them formally. Our intent is to characterize the desired properties within the scope of this report. As is the case in Section 6.1 for satisfying the stated goals, the recommendations in Section 12 consider what is needed to reason about these properties more comprehensively.

1. **Overall system security.** To specify and reason about the security of a distributed system as a whole, all the necessary requirements must be stated explicitly and related to the behavior of the functions of corresponding visible interfaces. Those interfaces may be at the application layer for a totally dedicated system, at the database layer for a database application, at the operating-system layer for an open-system network of systems, or at a combination of layers in those cases in which interfaces at multiple layers can be simultaneously visible. It must be possible to relate the visible-layer interface requirement specifications to the lower-layer specifications and corresponding lower-layer properties (whether visible or hidden), such as those properties discussed in the subsequent bulleted items. Thus, for example, the formalisms must facilitate abstraction refinement and mappings among different layers of abstraction. Different formalisms may be appropriate for different functions and different properties at different layers; these formalisms must be capable of interrelating, and should satisfy the bulleted items in Section 6.1.

2. **End-user-system security.** The TCSEC Orange Book B-level criteria are heavily concerned with multilevel security; it is important to be able to demonstrate that the kernel and TCB do not violate the multilevel-security requirements even in highly distributed implementations. Ideally, to demonstrate that, it should not be necessary to demonstrate the satisfaction of all sorts of properties apparently unrelated to multilevel security. However, in a conventional TCB architecture, there are many other properties beyond MLS with respect to which each component system must be trusted. On the other hand, in a RISSC architecture, it may be easier to demonstrate multilevel security without having to specify completely each individual end-user system — on the grounds that the end-user system cannot compromise multilevel security. Nevertheless, in the conventional architecture or in a RISSC architecture, it is much too simplistic to be interested only in the MLS property when vulnerabilities can exist that permit MLS compromise without a flaw in the MLS mechanisms themselves. Thus, the belief that demonstrating MLS-ness of a mechanism in isolation is sufficient is both simplistic and dangerous. For example, suppose the cryptobinding provided by an end-user system can be compromised so that one user can masquerade as another user who is authorized at a higher security level, or that the authentication itself can be subverted. It follows immediately that security of the overall system may depend upon the integrity of the end-user systems not being compromised, even in a RISSC architecture. Alternatively, in the presence of a trusted authentication server, the design may permit the end-user system to be untrusted with respect to authentication. In either case, the security of the overall system must be characterized and demonstrated, and in both cases that security depends on strong authentication in addition to enforcement of MLS.

3. **System-infrastructure and server security.** Traditional formal methods apply to servers with respect to demonstrating multilevel security properties; they are also suitable for other

properties essential to overall system security, such as the security of file servers, authentication servers, and network servers. Thus, confidentiality, integrity, and authentication properties must be characterized and ensured. Of particular concern is the management of the networking and distributed-system intercommunications, which are particularly vulnerable. The confidentiality and integrity of all crypto key management and networking crypto must also be ensured, with respect to the networking — irrespective of whether the software and hardware are located in network servers or (in non-RISSC architectures) in end-user systems. As discussed in Section 6.3, a collection of well-integrated formalisms for analyzing network protocols is very much needed; the BAN logic and subsequent extensions provide a prototypical basis for such formalisms.

4. **Network media security.** Network security itself (that is, media protection to ensure against loss of confidentiality, loss of integrity, and to some extent denials of service) would presumably rely on the use of excellent cryptographic techniques. If that is the case, then the network media need not be trusted to maintain confidentiality and integrity of communications, and presumably some alternative routing could minimize denials of service. Thus, the extent to which formal representations might be applicable to the analysis of network communications is related to the quality of the crypto and its key management. In a sensible RISSC implementation, the media themselves would be largely untrusted, and no analysis would be needed — except possibly for reliability and denials of service.

5. **System integrity.** Subsystem integrity relies on authentication, access controls, and many other subsystem attributes. Tamperproofing is particularly important. (See Section 4.3.) Formal requirements must embody and constrain structural encapsulation, nonbypassability, and nonalterability (for example, via some sort of cryptographic integrity check). They must also address responses to any detected alterations. For example, the use of cryptographically based integrity seals to ensure the presence of legitimate software modules would require formalisms to represent the security of the underlying operating-system mechanisms that might embody the sealing and seal checking, and something like the BAN logic to reason about the cryptographic processes.

6. **Data confidentiality.** Multilevel-secure systems can draw on the previous efforts to model MLS properties, to specify functionality that must satisfy those properties, and to verify that those properties are satisfied systemically. Additionally, formal methods can be applied to access controls (such as in [1]), changes to permissions, and transfers of authority.

7. **Data integrity.** Cryptographic or other forms of integrity checks can be modeled. If data integrity is handled similarly to system integrity (item 5, above), comparable methods would apply. However, whenever data is significantly more ephemeral in nature in comparison with the system software and hardware (which tend to remain unchanged for long periods of time), then different techniques might have to be used.

8. **Nondenial of service.** Traditionally, the prevention or mitigation of service denials has been less amenable to formal methods, although some existing research is applicable [69, 254].

9. **Authentication integrity.** The BAN logic [29] and its successors can be used to model and analyze authentication protocols (such as are used in Kerberos [14, 155, 236] and the Digital Distributed Secure System Architecture [68], and other strategies for distributed authentication — for example, [122, 252]).

10. **Nonrepudiation.** Nonrepudiation requires strong cryptobindings and nontamperable authentication, plus system security that would prevent or at least detect spoofing activities within end-user operating systems and servers (including authentication servers, file servers, and network servers). Ensuring nonrepudiatability requires more than just local reasoning about authentication protocols.

11. **Accountability.** Accountability as an end in itself seems to have been widely ignored by formal methodologists. It is a derived property rather than a primitive property. For example, the confidentiality of the audit-trail data depends on the security of the infrastructure; similarly, the nonbypassability of audit-trail data collection process depends on the integrity of the systems being monitored and on the integrity of the monitoring subsystems. The integrity of the audit-trail analysis depends on the nontamperability of the analysis subsystem. The integrity of the systems being monitored depends on interfaces between the target systems and the analysis subsystems.

A collection of typical properties that must be enforced is considered further in Appendix B, in the context of a monitoring system.

7.2 Finer-Grained Properties of Distributed Systems

The eleven properties of distributed systems given in Section 7.1 are stated as rather broad characteristic requirements. Table 11 illustrates some of the primary ways in which these properties depend on other properties of distributed system architectures. The properties are numbered in the table as enumerated above — although a finer-grain distinction is made among the system infrastructure components (i), the end-user systems (e), and the network media (n). For simplicity, all the infrastructure components (for example, file servers, authentication servers, and network subsystems such as network servers, trusted interface units, gateways, and firewalls) are lumped together in the table, although there are clearly differences among the different types of infrastructure components. However, we make a distinction between the networking subsystems (in software and hardware) and the network media. The network media accommodate the transmission of information, which is presumably encrypted whenever the use of an untrustworthy medium is deemed too risky; the networking subsystems ensure that encryption and decryption are performed in a trustworthy and nonsubvertible fashion. (As seen from the table, the network media and the networking subsystems depend on very different properties.)

Table 11 exhibits a collection of property dependencies that is considerably more complex than the simplistic view of an application `depending upon` an MLS TCB to enforce application-layer multilevel security (as in SeaView, in which the DBMS need not be trusted to enforce multilevel security). The table's depiction of dependencies is more realistic than the usual simplistic view (but by no means complete), and suggests that a major challenge of formal methods is to represent what really happens rather than to give an approximate and highly abstract oversimplification. Nevertheless, abstract representations are very useful, as long as they do not diverge from reality.

Table 11: Property dependence

Goal	Security Properties	Depended-upon subgoal
	OVERALL	
1	Overall system security: MLS, access controls, etc.	2,3,4, 5,6,7,8,9,10,11 (i,e,n in each case)
	STRUCTURAL	
2	End-user system security	3,4, 5e,6e,7e,8e,9e,10e,11e, 5i,6i,7i,8i,9i,10i,11i, 5n,6n,7n,8n,9n,10n,11n
3	Infrastructure security	4, 5i,6i,7i,8i,9i,10i,11i, 5n,6n,7n,8n,9n,10n,11n
4	Network media security	5n,6n,7n,8n,9n,10n,11n
	FUNCTIONAL	(within each infrastructure component)
5i	System integrity	6i,7i,9i, 5n,6n,7n, reliability, fault-tolerance
6i	Data confidentiality	9i,6n
7i	Data integrity	9i,7n
8i	Nondenial of service	5i,7i,9i,5n,7n,9n, hardware availability
9i	Authentication integrity	Authentication mechanism, keys
10i	Nonrepudiation	9i, algorithm strength, key protection
11i	Accountability	9i
	FUNCTIONAL	(within each end-user system)
5e	System integrity	5i,6i,7i,8i,9i,10i,11i,6e,7e,8e,9e, reliability, fault-tolerance
6e	Data confidentiality	6i,9i,9e, local encryption
7e	Data integrity	7i,9i,9e, local encryption
8e	Nondenial of service	5i,6i,7i,9i,5e,6e,7e,9e, hardware availability
9e	Authentication integrity	9i (authentication server)
10e	Nonrepudiation	9i, and perhaps 9e (but ideally not!)
11e	Accountability	9i,9e
	FUNCTIONAL	(over network media)
5n	Media integrity	No dependence if traffic well encrypted; otherwise, dependent on media behavior, malfeasors, and the environment
6n	Data confidentiality	No dependence if traffic well encrypted; otherwise, similar to media integrity, plus strength of encryption and key management,
7n	Data integrity	Strength of encryption and key management, error-detecting and -correcting codes
8n	Nondenial of service	Alternate routing, redundancy, retries; depends on media being within tolerances
9n	Authentication integrity	[Nonspoofing depends on infrastructure]
10n	Nonrepudiation	[Also depends on the infrastructure]
11n	Accountability	Network traffic monitoring

8 Property Transformations Under Composition and Layering

Section 8 considers some fundamental types of system and subsystem properties, and how those properties are affected by subsystem composition and hierarchical layering. In general, subsystem properties may undergo transformations as a result of composition or layering of subsystems, and properties of the resulting systems may be formally derived from the underlying properties. Our intent here is to make explicit the properties, the transformations they undergo, and the process of deriving composite properties — and to address all the assumptions on which formal reasoning must depend.

8.1 Properties

In the past, much emphasis has been placed on properties that remain invariant under simple no-feedback subsystem composition [2, 131, 134, 135, 136, 137, 139, 202, 206, 238, 255], in which the outputs of one subsystem become the inputs to the next subsystem.

Here we take a more general view that properties may be transformed under less simple compositions. Under the notion of *generalized composition,* we include aggregation of collections of modules within particular layers, hierarchical layering of modules in the presence of privilege mechanisms, and interactions among components — in both centralized and distributed systems. We also include effects of call-and-return semantics, feedback, and bilateral module structures in distributed systems. In certain cases, a transformation may represent an identity — namely, when particular properties are preserved by the composition. With this generalized approach, the compositions that preserve properties become an important special case of the approach in which properties are transformed, and to which case the existing body of research on composition invariants applies.

Basically, this notion of generalized composition is a tricky business. For example, suppose that we have two A1 systems that are connected via an unprotected local network. If passwords are able to flow unencrypted over the network, and are able to be read en route by would-be misusers, then the fact that two A1 systems are involved becomes almost irrelevant with respect to the assurance relating to the security of the composition. Similarly, if we layer a supposedly secure operating system on top of a fault-tolerant hardware platform, we might hope that we have attained a secure, reliable system. However, certain fault modes that exceed the given fault tolerance may result in a serious loss of security.

Lamport considers two types of properties, safety and liveness. A *liveness* property implies that a system will eventually do something; a *safety* property implies that if the system eventually does something, its behavior will be as specified. Adequate behavior typically implies that certain Lamport-style safety properties and certain liveness properties be satisfied. (In this report, we use "safety" to imply human safety rather than Lamport-style safety, and explicitly qualify the latter term.)

Several concepts that are normally undifferentiated in their relation to composition have been treated individually in a recent doctoral thesis by Heather Hinton [90]. Hinton distinguishes among five types of Lamport-style safety properties: *faithfulness, provenance, authenticity, integrity,* and *confidentiality.* She also identifies four types of progress properties (a form of liveness): *accessibility, eventual progress, termination,* and *eventual output.* Faithfulness implies consistent repeatability in the behavior of any particular component. Provenance implies that the identity of an originating system, program, or agent is known and well defined. Authenticity implies the absence of masquerading and the genuineness of a resource or action — for example, that the provenance is really genuine. Integrity and confidentiality essentially are as defined here. Accessibility implies that inputs are eventually accepted for processing. Eventual progress implies that some forward progress will eventually be made. Termination implies that the progress will ultimately conclude. Eventual output implies that the results of the progress will eventually be released.

This classification is useful in understanding the effects of subsystem composition. Some types of properties are domain independent (namely, faithfulness, provenance, authenticity, and the progress properties), whereas the others are domain dependent (integrity and confidentiality). With respect to Hinton's definition of composability (which is somewhat different from that of Sutherland [238]), the domain-independent properties are composable, whereas the domain-dependent properties are not always composable — depending on whether or not the composition permits feedback. Without feedback, the domain-independent properties are composable; with feedback, those properties are *contingently composable,* that is, they remain true for a particular component locally, but may not be true for interconnections of that component with other components having the same property.

Of particular interest here is the concept of an *emergent property* — namely, a property that is not meaningful with respect to lower layers or to individual components at a particular layer, but that is meaningful with respect to a higher layer or with respect to the composition of components at the given layer.[17] Hinton shows that the composition of domain-independent composable properties does not result in emergent properties, whereas the composition of domain-dependent composable properties may result in emergent properties. This concept is important because it reinforces the necessity of addressing domain-dependent properties.

To understand the intrinsic nature of emergent properties, consider the class of application-layer safety properties. Ultimately, user safety depends not only on the implementation of the application layer, but also on lower-layer functionality. For example, it depends on the reliability of hardware, operating systems, and database management systems (if any). It also depends on the ability of those lower layers to withstand malicious or accidental misuse, because any subversion could potentially result in a loss of safety. Whereas it is theoretically possible to expand the highest-layer properties in terms of all the subtended properties, it is in practice absurd to do so. It would violate the tenets of software engineering (such as abstraction) as well as common sense. Thus, we stress the importance of a specification approach in which abstractions at any particular layer are specified in terms of their own relevant constructs, and where the mappings between layers are used to bridge the gaps in abstraction between layers — in terms of both specifications and properties to be satisfied, in the style of Robinson and Levitt [199]. As a consequence, many of the higher-layer properties are either emergent or partially emergent. On the other hand, a module specified at one layer may also appear identically at higher layers, as illustrated by the asterisks and square brackets in the PSOS hierarchy shown in Table 6. In cases where identical functionality appears at different layers, the corresponding higher-layer properties are not emergent.

In dealing with hierarchical layering and compositions, abstraction is really the fundamental concept. However, Einstein's statement noted in Section 2.1 must be observed. Abstractions must be sufficiently complete, consistent, and truthful to describe everything that is essential to permit the

[17] Emergent properties are an old concept — in cellular biology and other disciplines. More recently, Leveson [116] considers emergent properties in the context of software safety, making the case that safety is itself an emergent property that is not meaningful at lower layers. Jim Horning privately suggested that resonance is an emergent property in a circuit composed of resistors, capacitors, and inductors.

gaps among different abstractions to be bridged and to enable reasoning about properties to be based on realistic assumptions. Parnas makes the distinction between abstractions and falsehoods. Abstractions are incomplete descriptions that intentionally simplify reality. Ideally, they must be truthful (as far as they go). On the other hand, descriptions that permit incorrect behavior to occur are dangerous, and must be avoided. There is a serious risk of errors of omission resulting from incomplete descriptions that introduce falsehoods.

8.2 Transformations in MLS systems

Viewed somewhat simplistically, the canonical kernel-TCB decomposition espoused by the TCSEC leads to a kernel that enforces the strict MLS property but that also must permit certain privileged operations that may (and can) actually violate the strict-sense MLS properties (for example, no adverse flow with respect to security levels and no bypasses of the kernel interface); the TCB then attempts to ensure that the privileged operations cannot be misused from the TCB interface, which must itself be nonbypassable and must properly encapsulate the kernel. An example of such a violation of the kernel MLS property occurs when it is necessary to perform trusted downgrading, sanitization, or abstracting. Thus, the set of kernel properties that must be satisfied is not identical to the set of TCB properties, and the TCB objects are not necessarily identical to the kernel objects. Whether or not the abstract objects are the same at each layer, the trusted exceptions that potentially violate strict MLS must be completely modeled and analyzed, and the relationship between the TCB and the rest of the system must be suitably constrained.

An early effort to model privileged TCB operations was reported in 1985 by Benzel and Tavilla [15] for the Honeywell Secure Communications Processor (SCOMP), using Gypsy to specify and analyze the TCB. The kernel enforced multilevel security [229], but permitted some privileged overrides. The trusted mechanisms were represented by three TCB policies, as follows:

- Privilege Policy, relating to the invocation of privileged kernel functions, the privileged modification of certain kernel attributes, and the explicit overriding of the kernel properties (in particularly, enabling writing down in violation of Bell and LaPadula).
- Integrity Policy, in particular the Biba policy of preventing low-integrity entities from contaminating higher-integrity entities, with respect to objects perceived by the TCB. Note that in SCOMP, MLI was enforced by the TCB and not by the kernel.
- Functional Correctness Policy, with respect to certain highly critical but unprivileged TCB functions that must operate correctly. Critical functions included correct (noncompromisible) labeling of security levels and Trojan-horseless system editors.

Each of these policies represents an emergent property that is not relevant to the underlying security kernel. The strict MLS property of a security kernel (even including any privileged-exception mechanisms) is typically considered as the primary kernel property of interest. However, the trusted computing base must also be analyzed thoroughly, and that requires the representation and mappings of emergent properties above and beyond the kernel MLS property, as noted in Section 8.1.

As described in [15], the formal design proofs carried out with Gypsy found one multilevel-security flow violation and one covert channel. This example clearly illustrates that there is much more than strict MLS that must be considered, and that the TCB and kernel each has its own set of properties to be enforced and maintained. This example, along with SeaView and PSOS, show how the properties of one layer can be analyzed based on the specifications and properties of the lower layers.

Also of interest in this context is an effort undertaken for the LOCK system, to show the relationship between properties at one layer and properties at the next layer [215].

8.3 Transformations of SeaView Properties

The transformational approach is further illustrated by considering the SeaView properties noted in Table 10. The OS-TCB layer enforces multilevel security on operating-system virtual memory objects, while the DB-TCB layer enforces a variety of database properties on database objects. Although the multilevel-security property is in a vague sense preserved by ascending from the OS-TCB to the DB-TCB, a formal representation of the properties at each layer must reflect the fact that the objects at each layer are actually quite different, and therefore that the property representations are different. Indeed, the objects at the different layers must be different, in the interest of type consistency. Furthermore, there are numerous other database properties at the upper layer that can be derived from the properties of the lower layer and the specification of the upper-layer functionality. Besides, the exception conditions are different at each layer, and at each layer they are expressed in terms of their own abstract objects. Whereas it is possible in an abstract sense to polymorphically have the same property hold at both layers, the complete representation must include the different interface semantics as well as the relevant different exception conditions. Thus, it makes sense to consider how properties of the upper-layer specifications can be considered as transformations of the lower-layer properties.

Referring to Table 10, the relevant DB-TCB property relating to MLS requires that, for any given granularity of database access (e.g., queries or updates on multirelation joins, single-relation views, relations, or elements), the security level of the requester must be commensurate with the security level applicable to the requested data. In addition, many of the upper-layer properties are emergent, and dependent on lower-layer properties that are not noted in the table, for example, relating to the correctness or fault-masking abilities of the lower layer. Thus, to reason about the upper-layer properties, a more complete set of lower-layer properties must be present. Perhaps this added difficulty is why so much emphasis has historically been placed on simple properties such as strict MLS and on simple compositions, rather than on more complex properties and on less simple compositions and layerings.

8.4 Transformations Within a Byzantine Clock Subsystem

Byzantine protocols also present an opportunity for a transformational view. Consider a Byzantine clock subsystem [113] that is built out of $3f+1$ ordinary clocks, for which the Byzantine clock subsystem provides the correct time despite the arbitrary misbehavior of any f constituent clocks (each of which, for example, might maliciously or accidentally provide substantively different readings to its neighboring clocks at any particular time). Here the Byzantine clock subsystem `depends on` the constituent clocks, but does not `depend upon` the correctness of all the constituent clocks; furthermore, the properties of the individual clocks can be used to derive (that is, to prove) the properties of the resulting Byzantine clock, based on the details of the particular Byzantine clock algorithm and its implementation, and on the details of the logic for masking misbehavior. The nature of the transformations from components to subsystem is similar to a simple hierarchical layering, although the resulting subsystem properties can be derived despite a total lack of assumptions about the behavior of at most f misbehaving components. This is somewhat different from the more conventional layering, in which explicit formalization of relevant properties of lower-layer functionality is required to derive the desired properties of the upper layer.

Further discussion of a three-layered model consisting of (1) clock synchronization [207], (2) Byzantine agreement [118, 119], and (3) diagnosis and removal of faulty components is considered by Lincoln et al. [120] — who also provide formal verifications for a variety of hybrid algorithms [119] that can greatly increase the coverage of misbehaving components. This three-layered integration of separate models and proofs is of considerable practical interest, as well as representing innovative uses of formal methods.

8.5 Transformations Under Gateway Interposition

A different kind of example is provided by a gateway that controls all traffic entering or leaving a given site (that is, the inputs and outputs of potentially all functionality logically inside the gateway). For example, a gateway might act as

- a filter that permits certain communications subject to specific authorizations (inbound, or outbound, or both)
- a firewall that blocks all communications (inbound and outbound)
- a firedoor that operates only in emergency situations (inbound, or outbound, or both)

Each of these cases might involve differing constraints on inbound and outbound traffic, and on different types of communications.

Gateways to the Internet represent a popular example of the need for a trustworthy subsystem.

Three types of desirable properties are suggested (in addition to the properties normally expected by the systems on whose behalf the gateway is operating):

- Maintenance of the confidentiality of information within the given site, with respect to information traversing the gateway from the given site to the rest of the Internet. Thus, there must be no possibilities of information confidentiality being compromised through the gateway.
- Maintenance of the system integrity of the given site, with respect to traffic traversing the gateway from the rest of the Internet to the given site. Thus, modifications of internal site systems (including the insertion of Trojan horses and other nasty software) must be prevented by the gateway.
- Prevention of denials of service within the given site, caused from the rest of the Internet.

In each of these three cases, the gateway has been described as protecting or governing what is logically inside the gateway. Alternatively, by reflection, a gateway could conceptually act on behalf of both the inside and the outside, for any of the three types of properties — for example, in the case of a bidirectional filter. (In that case, it might seem superficially that a bidirectional filter could be split into two unidirectional filters; however, it may be desirable for the filter to have a state memory that can enable it to relate what has gone out to what has come in and vice versa — in which case the filter may more easily be considered as either a single bidirectional filter, or as two unidirectional filters with controlled communications.)

The first two of these gateway properties have the appearance of properties that must be preserved by the gateway, with respect to outward information flow and inward information flow, respectively. However, the gateway must also be able to enforce transformed versions of those properties, and in some cases stronger versions — for example, relating to additional authentication, monitoring, and traceability that may not be present in the internal systems that the gateway is protecting. Furthermore, the third property relating to the prevention of denials of service within the gateway and within the internal systems could be absent or only vestigially present in the internal systems. Thus, the third property is emergent (in the sense of Section 8), whereas the first two properties also have some emergent aspects. Thus, the relationships among properties to be satisfied by the gateway and the properties to be satisfied by the internal systems can beneficially be addressed by considering the transformations that each property undergoes.

8.6 Transformations Within a Cryptographic Protocol

Situations in which information and associated properties are transformed from step to step within a cryptographic protocol can be modeled by a logic such as the BAN logic [29] (Section 6.3), which assists in reasoning about trustedness and trustworthiness in cryptographic protocols.

As information undergoes transformations, so do properties pertaining to the information.

- Crypto for confidentiality implies a transformation from an unencrypted form to an encrypted form and a transformation from an encrypted form to an unencrypted form. Properties pertaining to the use of such crypto include nonbypassability of the encryption process, nonexposure of the keys, and algorithmic strength. For example, what happens to these properties overall? Furthermore, there may be vulnerabilities if intermediate stages of a particular implementation are accessible, as in the case of a software-implemented triple DES.
- Crypto for user or peer authentication implies a transformation from identity information to cryptobinding information, and a transformation on the cryptobinding information that authenticates the original identity. Properties that may themselves be transformed under these information transformations involve (for example) authenticity, nonspoofability, and nonrepudiatability. For example, these attributes typically depend on the integrity of the underlying operating system and on the protection of crypto keys.
- Crypto for entity integrity implies a transformation from (for example) a message to its cryptobinding information and another transformation from the message and its cryptobinding information to a binary decision that verifies entity integrity. Properties that may be assessed involve (for example) integrity, nonspoofability, and nonrepudiatability.

Confidentiality, integrity, authenticity, nonspoofability, and nonrepudiatability are all dynamic properties that can be compromised during system use.

8.7 Commonalities Among Different Types of Transformations

The foregoing sections exhibit various forms of composition and layering, above and beyond the traditional simple serial connection in which outputs from one component become inputs to another (with no reverse path in the automata sense) or invocation (with some sort of call-and-return semantics). These forms are summarized as follows.

- 8.2. Multilevel security: Layering of a trustworthy mechanism on top of a privilege mechanism
- 8.3. SeaView: Layering of an untrusted mechanism on top of a trusted mechanism
- 8.4. Byzantium: Layering of a trustworthy mechanism on top of untrusted mechanisms
- 8.5. Trustworthy gateways such as firewalls: Interposition of a trustworthy mechanism between two mechanisms of indeterminate trustworthiness, to control the flow of traffic (in one or both directions)
- 8.6a. Crypto for confidentiality: Interposition of a trustworthy mechanism between two mechanisms of indeterminate trustworthiness, to protect the intermediate medium and one or both of the two end mechanisms from being compromised while information is in an untrustworthy transmission or storage medium
- 8.6b. Crypto for entity authentication: Inclusion of a trustworthy mechanism for cryptobinding to determine an authentication signature, and a possibly somewhat less trustworthy mechanism for checking the authenticity of the cryptobinding
- 8.6c. Crypto for entity integrity: Inclusion of a trustworthy mechanism for applying an integrity signature, and a possibly somewhat less trustworthy mechanism for checking its integrity

9 Formal and Semiformal Methods Useful in Other Disciplines

Section 9 summarizes only briefly a few relevant efforts outside of the security area that should be studied further for their potential applicability to security.

A session organized and chaired by P.G. Neumann for the 1995 National Information System Security Conference considers recent progress relating to reliability, fault tolerance, safety, and security applications, and how progress in other disciplines might be applied to secure systems and communications. Position statements from Neumann [160], R.W. Butler [31], R. Kurshan [112], and W. Legato [115] are included in the 1995 NISSC proceedings.

Butler has sponsored extensive work at the NASA Langley Research Center, aimed at applying formal methods to life-critical subsystems in aerospace applications — much of which is summarized in [32]. In various relevant R&D and technology-transfer projects, formal methods have been applied to architectural-level fault tolerance, clock synchronization, interactive consistency and Byzantine agreement, design of special-purpose hardware devices and units, asynchronous communication protocols, design and verification of ASICs (application-specific integrated circuits — custom-designed hardware for specific problem domains such as signal processing), decision tables, railroad-signalling systems, and analysis of both the Space Shuttle software and aircraft navigation software. Some of this work uses approaches and tools developed by organizations known for past security work, notably SRI Computer Science Laboratory (SRI's PVS for specifying the instruction set and microarchitecture of the AAMP5 microprocessor, EHDM for the Reliable Computing Platform and for clock synchronization — see [173]), Odyssey Research Associates (modeling railway switching control for Union Switch and Signal, using TBell for decision tables, and the development of Romulus, and using Penelope for a Boeing 777 component), to cite just a few recent efforts. See [32, 33] for further references, and [92] for the proceedings of the most recent Langley Formal Methods Workshop. Also, see [37, 38] for two NASA guidebooks on the use of formal methods in NASA applications.

John Rushby provides a comprehensive general view of the ways in which formal methods are applicable to airborne systems [209], and to safety more generally [208]. Many of his conclusions relative to safety are also applicable here. (See Section 12.1.)

David Parnas and Nancy Leveson have each used extensive formal analytic techniques to model control-system properties, for example, related to the Ontario Hydro nuclear-power shutdown system. Parnas's techniques for formalizing documentation and applying them to state tables appears to be quite applicable to security applications. For the FAA's aircraft collision-avoidance systems, Leveson has also provided formal requirements that have been adopted as the official requirements. Her techniques for fault-tree analysis and hazard analysis are somewhat specialized, but representative of a kind of analysis that might be useful in security applications. Leveson's recent book on Safeware [116] provides extensive background on these techniques. Some of Parnas's recent work is summarized in [179].

Model checking is becoming recognized as a useful middle ground between two extremes: higher-order logics that require complicated verification, and informal description languages that permit little or no formal analysis. Model checking uses finite-state representations (such as binary decision diagrams) and enables a decidable analysis (explicit or implicit) of the state space to determine the satisfaction of certain properties. It is applicable to hardware and software. Typical approaches involve exhaustive or heuristically motivated searches through the entire state space.

Edmund Clarke and Kenneth McMillan (SMV [43, 140]) at Carnegie Mellon University, Robert Kurshan (COSPAN [87]) and G.J. Holzmann (SPIN [93]) at AT&T Bell Laboratories, David Dill at Stanford (Murϕ [57]), and others have used model checking in various applications, involving communications protocols, cache-coherence algorithms, hardware designs, and asynchronous systems. For additional references, see [9, 11, 25, 28]. Model checking is easily automated, and is adept at finding design flaws — which are explicitly represented in the analysis rather than merely suggested. Model checking does not seem to scale well for very large systems, but is ideal for analyzing relatively small state spaces. It would appear to be most useful if it could be used in combination with hierarchical and compositional formal methods, and with theorem proving and/or proof checking. However, this integration of different approaches has not yet been accomplished other than partially and experimentally.

The European community is very fond of formal description languages (FDLs) such as Z (Zed), VDM, ESTELLE (Extended Finite State Machine Language), LOTOS (Language Of Temporal Ordering Specification), and SDL (Specification and Description Language). See Spivey [232] and Potter et al. [190] for definitions of Z and [243] for a detailed overviews of ESTELLE, LOTOS, and SDL. Each of these languages is logically based on finite-state machines. ESTELLE uses Pascal data types; LOTOS and SDL use algebraic specifications of abstract data types. Of these languages, ES-

TELLE appears best able to accommodate hierarchical layering of specifications. One of the main uses of the last three of these languages is for protocol specification and verification; for example, see [100] for an application of ESTELLE. The establishment of each language was heavily driven by desires for language standardization, but each is amenable to some formal analysis. Each of these approaches has its own strengths and areas of applicability.

Z has been used in a variety of applications, including specification of a safety-critical control system [95] and a specification of part of IBM's CICS operating system component. Various examples of the use of the other three FDLs are given in [243].

Bicarregui and Ritchie give a comparison between the specifications in VDM and B of a communications protocol previously specified in CCS [20].

Numerous algebraic (as opposed to state-based) specification languages also can be considered, such as OBJ [71] using equational logic, and CSP [91] using Communicating Sequential Processes. Historically, algebraic techniques have generally been less applicable to representing and reasoning about large and complex systems, and have not been used much for secure systems.

Various temporal logics exist that are suitable for specifying real-time algorithms. However, they have usually not been applied to security problems or to specifying large systems.

A useful summary of the use of formal methods in a variety of real-system industrial applications is given by Craigen at al. [47], based on an earlier report [46].

Part Three: FORMALIZING RISSC ARCHITECTURES

10 Implications of the RISSC Philosophy on Formal Methods

Section 10 considers the effects that RISSC architectures have on system specification and analysis. We examine whether the use of formal methods in the design and implementation of RISSC systems can be simpler than the corresponding uses in more conventional distributed-system and centralized architectures. In particular, we consider the implications of subsystem dependence and property dependence on the corresponding statements and proofs of properties.

In principle, the reduced-interface notion of the RISSC philosophy should simplify the task of applying formal methods to complex distributed systems. In general, in a given design (or worse yet, in its implementation), it is difficult to disentangle interrelationships among properties unless the corresponding design and implementation entities to which those properties refer are themselves suitably decoupled. Thus, the primary challenge is one of decoupling a design into relatively separable components, according to the RISSC philosophy; from that decoupling, a simplification of the specification and analysis can follow.

The dependencies of Table 11 are seemingly more elaborate than those made explicit in published formal analyses. However, this apparent complexity must not be construed as an indication that things are any more complicated than they have always been. Reality is often more complex than we would like it to be, and the table merely illustrates that (1) the satisfaction of a particular abstract property typically depends on the satisfaction of other properties, in a way that is usually not made explicit, and (2) many detailed properties and the ways in which they interrelate have tended to be sublimated in the past.

10.1 Implications on the System Development Process

Conventional distributed systems are typically plagued by a variety of shortcomings that affect system development and analysis, such as the following:

- Overly powerful global mechanisms (such as the Unix superuser facility) violate the principle of least privilege. (There are various efforts to separate the superuser functionality into privileged subdomains — such as Badger et al. [10], who describe a partitioning of the Unix superuser functionality into 27 different privilege classes.)
- Unprotected external and internal interfaces such as `.rhosts`, `rsh`, and `rlogin` mechanisms may be easily penetrated. Pervasive interconnectivities that are largely unchecked are also risky (such as the use of `root` within mechanisms that in actuality do not require privileges but are given them primarily to simplify implementation).
- Unforeseen interactions may exist among seemingly isolated or single-purpose subsystems (such as global dependence on a single nonredundant authentication server, `depended upon` for a variety of functions above and beyond just authentication).
- Unvalidated executables may be hidden within apparently benign information — for example, PostScript files or Microsoft icons containing Trojan horses, or E-mail letter bombs that get

interpreted by mail-reader programs.

Vulnerabilities such as these present an open invitation to Trojan horses and other forms of misuse.

The RISSC approach seeks to minimize these harmful effects, by decoupling the subsystems and localizing effects to within subsystems, wherever possible — especially with respect to security, but also with respect to system reliability and fault tolerance. In a narrower sense, that is sort of what the TCSEC TCB-kernel approach attempted to do in the small. The significance of the RISSC approach is that it attempts to extend the intent of the TCSEC, in the large — to highly distributed systems and networks, to reliability issues, and to a broader sense of security. Abstraction, modularization, structural encapsulation, separation of concerns, allocation of least privilege, minimization of dependencies, and other similar techniques are completely consistent with the RISSC philosophy, and can significantly enhance security, as noted in Sections 2, 4.1, and 5. These concepts also lend themselves naturally to easier application of formal methods.

10.2 Implications on System Analysis

RISSC techniques can contribute notably to the security inherent in a system design, perhaps most significantly by forcing a structure on the design that in turn can simplify specification and implementation, and can help to avoid many of the more common system vulnerabilities that typically result from poor design (such as those enumerated in [159]). Assuming that a design has adhered to the RISSC philosophy, the analysis of that design and of its implementation can also be greatly simplified.

Interactions among seemingly isolated subsystems must be made explicit, and potentially harmful interactions must either be demonstrably avoided, or else masked by higher layers. In addition, the effects of would-be Trojan horses and design flaws must be prevented, masked, or at least identified and isolated. A pitfall with all system specification techniques is that a component may do exactly what it is expected to do, but may additionally do something else that is not explicitly prohibited — such as occurs in surreptitious Trojan-horse activities. Thus, it is advisable to have both positive specifications as to what *must* occur, and negative constraints as to what must *not* occur — in whatever form is suitable, for example, requirements or specifications. However, because it is very difficult to enumerate exhaustively all the bad things that should not happen, explicit requirements that constrain bad effects are important, particularly if those bad effects can arise from weaknesses resulting from the structure of the overall architecture.

11 Appropriate RISSC Architectures

Section 11 selects several representative subfamilies from three basic families of multilevel-secure architectures enumerated in Appendix A. These subfamilies are more or less adherents of the RISSC concept, and have varying ease of amenability to formal methods.

The first family [[1]] of system architectures in Section A.1 contains two subfamilies [[1a]] and [[1b]], denoted as follows:

- [[1a]] MLS,User-S1,AT,ZN-ZT-MT,XU,FP-FS-MU-CT
- [[1b]] MLS,User-Sn,AT,ZN-ZT-MT,XU,FP-FS-MU-CT

The second family [[2]] includes one subfamily that is of particular interest, namely, [[2a]] — which is a slight variant of [[1a]]:

- [[2a]] MLS,User-S1,AT,ZN-ZT-MT,XU,FNc-FS-MU-CT

The architectures of [[1a]], [[1b]], and [[2a]] have only single-level file servers, and are useful examples of the RISSC concept. Another subfamily [[3a]] allows the presence of multilevel-secure file servers, and is therefore somewhat less RISSC oriented, but still appropriate for discussion:

- [[3a]] MLS,User-S1,AT,ZN-ZT-MT,XU,FP-FT-FM-MT

The notation used to represent these cases is that used in Appendix A. For each of the selected RISSC-oriented architecture subfamilies, multilevel-secure systems (MLS) are composed of single-level end-user systems (User-S), trustworthy authentication servers (AT), trustworthy networking that enforces multilevel security (ZN-ZT-MT), untrusted network communication media (XU), and semitrusted file servers. Neither [[1a]] nor [[1b]] nor [[3a]] adds any encryption for file storage (FP), while [[2a]] is identical to [[1a]] except that files are encrypted by the end-user system and stored by the file servers in the resulting encrypted form. Each of [[1a], [[1b], and [[2a]] has file servers that are trusted to support discretionary access controls for shared file access, but that do not need to enforce multilevel security (FS-MU-CT); that is, each file server is a single-level server (FS). Family [[3a]] has multilevel-secure servers. Variants [[1a]], [[2a]], and [[3a]] have single-user end-user systems (User-S1); variant [[1b]] has end-user systems any of which may have competitive users (User-Sn).

Because these architectural subfamilies are closely related to each other, they are considered together in Section 11.1.

This report has devoted some effort to multilevel-secure system complexes. It suggests that the RISSC concept is beneficial in simplifying the problems of getting vendors to provide multilevel security without having to do massive redesign of their primary product lines, while at the same time simplifying some of the uses of formal methods as applied to those systems.

On the other hand, Section 5.6 considers the usefulness of the RISSC concept even in the context of conventional single-level applications in which no multilevel security is required. The architectural subfamily [[7a]] also represents a RISSC-oriented architecture that has most of the benefits of [[2a]] — except that it does not support multilevel security:

- [[7a]] User-S1,AT,ZN-ZT,XU,FNc-FU

Much of the discussion of [[2a]] applies to [[7a]] as well.

The choice of architecture ultimately depends on what components can sensibly be trusted. If end-user systems are not trustworthy, placing more security in the servers is important. If file servers and network connections are particularly trustworthy in the environments in which they operate, encryption may be unnecessary for storing files on the file servers. If file servers cannot be secure, then end-user encryption is desirable, along with multiple storage sites to avoid denial-of-service attacks. If end-user systems are really trustworthy, then multiuser end-user systems are feasible. If it is absolutely essential that each end-user system be multilevel secure, then trustworthiness must be much greater. Whether or not the resulting analysis is simplified depends on which architecture is chosen.

11.1 Properties of RISSC Architectures

We characterize here the security-relevant properties that must be satisfied in each case. These properties provide an instantiation of the generalized properties of Section 7.1. Properties are treated as a group wherever they are identical for multiple cases, and differences are explicitly noted (namely, those relating to end-user-systems, file encryption, and file servers). The term *trustworthy* is used in the general sense, not just in the narrow sense of MLS; we attempt to be very explicit as to what functionality must be trustworthy.

- **Overall system security (MLS)**. Every system complex of each subfamily must enforce multilevel security with respect to all end users and all accessible objects (irrespective of the MLS-trustworthiness of the end-user systems). In addition, all other relevant security criteria (including but not limited to the applicable elements of the TCSEC and its successors) must be satisfied, as required.

- **Overall network security (MLS)**. Any networking of each subfamily must enforce multilevel security with respect to all end users and all accessible objects (irrespective of the MLS-trustworthiness of the end-user systems), as well as other relevant security criteria.

- **End-user-system security for [[1a]], [[2a]], and [[3a]] (User-S1) and [[1b]] (User-Sn)**. The common requirements among the subfamilies are as follows. Each end-user workstation is a single-level system. Consequently, there are no local multilevel-security requirements. The end-user systems must be trustworthy with respect to user authentication (nonspoofable), whether there is only one user or a collection of potentially competitive users. The end-user systems must be free of maliciously implanted Trojan horses that could compromise user-system integrity. Their network interface software must not permit unintended leakage of (single-level) outbound information or insertion of extraneous inbound information (data and executables). The audit-trail collection must be nonbypassable.
  - **Noncompetitive end-user systems [[1a]], [[2a]], and [[3a]] (User-S1)**. Single-user end-user systems do not require individualized access controls, although they still can benefit from controls that isolate users from the operating system and other utility software.
  - **Competitive end-user systems [[1b]] (User-Sn)**. Competitive-user end-user systems must additionally provide local separation of users, in terms of discretionary access control and user process isolation.

- **Authentication-server security (AT)**. Normal user-to-system authentication (that is, authenticating the user to the system) and the reverse-direction system-to-user authentication (as in the TCSEC trusted path that authenticates the system to the user) could in principle be distributed and replicated into each of the end-user systems, whereas system-by-system (e.g., server-by-server) authentication cannot. However, such distribution of responsibility would imply a level of authentication-trustworthiness that would often not be commensurate with the trustworthiness of the available end-user systems — which themselves might be too easily compromisible. Furthermore, the bilateral authentication is an example of mutual suspicion, where the lack of integrity on both sides is a potential vulnerability. Thus, it is sensible to remove from the end-user systems as much trustworthiness as possible for end-user authentication.

The use of trustworthy authentication servers can avoid the need to trust the end-user systems for user-to-system authentication, although it is still desirable to have end-user systems be able to trust the authentication servers and other servers via server-to-user authentication. The avoidance of trust in the end-user systems is perhaps more important in [[1b]] than in [[1a]], [[2a]], and [[3a]], but can also be important in [[1a]] and [[2a]] environments whenever physical security cannot be trusted. The authentication servers must be secure against both remote and internal attacks, and their authentication process must be nonspoofable (including protection against replay attacks). Although the authentication servers do not need to be trusted to enforce multilevel separation with respect to user objects, they may still need some knowledge about the association of maximum clearance levels with users and with systems. They must also be concerned with inference channels and possibilities for traffic analysis that may exist because of the presence of the authentication servers. If this is perceived as a problem, it is possible to have separate authentication servers for the most sensitive levels. Alternatively, distributed authentication can be employed, using multikey encryption schemes such as Micali [52, 142], whereby dependence on a single authority can be avoided.

- **Network-server security (ZN-ZT-MT)**. In each of these RISSC subfamilies, network interfacing cannot be distributed and replicated into each of the end-user systems — because the network servers must ultimately be enforcers of multilevel security, and the end-user systems cannot be trusted for multilevel security. The usual MLS properties must be satisfied by all traffic passing through the network servers, and those servers must not be bypassable or compromisible. The network servers are also responsible for encryption and decryption of any sensitive communications, especially whenever those functions are not provided end-to-end by the end-user systems. In addition, cryptographic checksums and error-detecting and error-correcting coding would presumably be provided by the network servers to increase integrity, reliability, and availability. File access is mediated by the combination of the authentication servers and the network servers, which prevent violation of multilevel security. Encryption and decryption of files for transmission can be provided either by the end-user systems (particularly in [[1a]] and [[2a]]) or — whenever the paths between end-user systems and network servers are suitably trustworthy — by the network servers).

  In architecture family [[1]], the single-level file servers are not required to provide any file encryption, although they could decrypt encrypted transmissions of files from the network servers. In [[2a]], the network servers may not need to provide additional crypto for files, because the end-user systems already do so. In [[3a]], encryption by the network servers is desirable.

- **Communication security (XU)**. The communication media can be untrusted for confidentiality (CU), and also for integrity (IU) if the network servers provide integrity checks and error-correcting coding. Some additional protection may be desired for defense against denials of service (DT), such as alternative routing among en-route network controllers.

- **File-server security (FP-FS-MU-CT in [[1a]] and [[1b]], FNc-FS-MU-CT in [[2a]])**. In each of these subfamilies, the single-level file servers store information only at a single level (and, if desired, compartment). The common requirement is that the file servers must isolate the files for one user from the files for another user. That is, the file servers must deliver the unimpaired files to the proper end-user system. In [[1b]], it is then the responsibility of the end-user system to ensure appropriate separation among competing users on each end-user system. Because there is no multilevel security *per se* (and labels, if they exist, would presumably be for advisory purposes only), only discretionary access controls are required in the file servers, with read, write, execute, and any other appropriate privileges, as desired. Whenever files may be shared among different users, the access-control permissions must reside in the file servers.

File encryption and decryption are not required in [[1]]. File encryption is provided by the end-user systems in [[2a]], which architecture (as noted above) is identical to [[1a]] except that the user client provides the encryption before files are transmitted to the file server. In this case, discretionary access controls include the typical read-write permissions, but access also requires possession of the appropriate file-storage keys. Consequently, controlled sharing of stored-encrypted files also requires controlled key distribution.[18]

- **File-server security (FP-FM-MT-FT in [[3a]]).** In this subfamily, the multilevel-secure file servers must enforce multilevel-security separation, with sufficiently high assurance commensurate with the overall security requirements. This adds considerably greater trustworthiness requirements and necessitates that the file servers must be strongly noncompromisible. With respect to enforcing multilevel security, there are various trade-offs between having to trust primarily the network servers (as in [[1]] and [[2]]) and also having to trust the file servers as well (as in [[3]]).

Other architectures can be subjected to similar analyses. In addition, hybrid architectures can be considered, with combinations of components and interconnections from several families or subfamilies. The main purpose of this section is to demonstrate the types of properties that must realistically be made explicit.

Of the four cases shown, [[1a]] and [[2a]] most strongly adhere to the RISSC concept, with [[1b]] less strictly compliant, and [[3a]] also less — but for different reasons. Nevertheless, all four cases are of considerable interest from the point of view of incorporation into readily configurable commercial systems, and have great potential in terms of their implications on the application of formal methods, as noted in Sections 10.1 and 10.2. Similar comments apply to the single-level systems of subfamily [[7a]].

11.2 RISSC Relevance to Security Criteria

Table 12 exhibits the various TCSEC criteria elements [151] and indicates how relevant they are for each of these different system components of each of the RISSC-oriented architecture subfamilies, [[1a]], [[1b]], and [[2a]]. The TCSEC criteria elements are used here for illustrative purposes, because they are familiar. One extra item is added, augmenting covert-channel analysis with a new requirement for defense against non-MLS traffic analysis, such as provided by OPSEC (Operational Security) techniques. Other, more recent, criteria could be used, but would have very similar manifestations in the table.

In the table, a solid bullet (•) indicates that the particular component contributes significantly to the satisfaction of the given criteria element. An open bullet (∘) indicates a secondary contribution.

[18] Although it is conceptually possible that access controls could be replaced altogether by using different keys for encrypting the same objects with different access permissions, this is not a good idea, for a variety of reasons.

Table 12: Relevance of RISSC families [[1]] and [[2]] to security criteria

Criteria elements	Architectural Components						
	[[1a]] User-S1	[[1b]] User-Sn	[[2a]] User-S1	AT	ZN-ZT-MT	XU	FS-MU-CT
TCSEC Security Policy:							
Discretionary access control	•	•	•				•
Object reuse		•					•[[1]] only
Labels:							
Label integrity	◦	◦	◦		•		
Exportation (3 criteria)					•		
Labeling human-read output	◦	◦	◦				
Mandatory access controls					•		
Subject sensitivity labels	◦	◦	◦				
Device labels					•		
TCSEC Accountability:							
Identification/authentication				•			
Audit	•	•	•	•	•		•
Trusted path	•	•	•	•	•		•
TCSEC Assurance:							
System architecture		◦		•	•		◦
System integrity	◦	◦	◦	•	•		•
Security testing	◦	◦	◦	•	•		◦
Design spec/verification				•	•		◦
MLS covert-channel analysis					◦		
Non-MLS traffic analysis						◦	
Trusted facility management		◦		•	•		◦
Configuration management		◦		•	•		◦
Trusted recovery	◦	•	◦	•	•		◦
Trusted distribution	◦	•	◦	•	•		◦

Legend:
• = primary contribution to criterion fulfillment
◦ = secondary contribution to criterion fulfillment

Table 13: Relevance of RISSC family [[3]] to security criteria

	Architectural Components					
Criteria elements	[[3a]] User-S1	[[3b]] User-S1	AT	ZN-ZT-MT	XU	FP-FM-MT-FT
TCSEC Security Policy:						
Discretionary access control	•	•				•
Object reuse		•				•
Labels:						
Label integrity	◦	◦		•		•
Exportation (3 criteria)				•		•
Labeling human-read output	◦	◦				•
Mandatory access controls				•		•
Subject sensitivity labels	◦	◦				•
Device labels				•		•
TCSEC Accountability:						
Identification/authentication			•			
Audit	•	•	•	•		•
Trusted path	•	•	•	•		•
TCSEC Assurance:						
System architecture		◦	•	•		◦
System integrity	◦	◦	•	•		•
Security testing	◦	◦	•	•		◦
Design spec/verification			•	•		◦
MLS covert-channel analysis				◦		
Non-MLS traffic analysis					◦	
Trusted facility management		◦	•	•		◦
Configuration management		◦	•	•		◦
Trusted recovery	◦	•	•	•		◦
Trusted distribution	◦	•	•	•		◦

Legend:
• = primary contribution to criterion fulfillment
◦ = secondary contribution to criterion fulfillment

Family [[3]] is considered in Table 13. Because of the requirement that potentially all the file servers may need to enforce multilevel security, the architectures of family [[3]] are somewhat less in tune with the RISSC philosophy, although still vastly moreso than the more TCSEC-oriented systems in which all end-user systems may be multilevel secure. The analysis for [[3a]] is similar to those of [[1a]] and [[2a]]. Table 13 clearly demonstrates the criteria-related critical dependence on multilevel separation for the file servers. However, this is not an obstacle if such file servers are routinely available. For comparison, Table 13 also includes subfamily [[3b]], which bears the same resemblance to [[3a]] as [[1b]] bears to [[1a]], namely it allows competitive users on the end-user systems:

- [[3b]] MLS,User-Sn,AT,ZN-ZT-MT,XU,FP-FT-FM-MT

11.3 Illustrative RISSC Properties: Modular Crypto

As an illustration of desirable properties relating to the RISSC architectures of Section 11.1, consider the design and implementation of systems with modular crypto discussed in Section 5.7, and the risks that must be avoided. In these RISSC architectures, crypto contributes to the satisfaction of several security criteria elements of Table 12, notably, discretionary access controls in [[1b]], object reuse in [[1a]] and [[1b]] (it is not a problem in [[2a]] or [[7a]]), cryptobinding and authentication, and trusted paths from end-user systems to servers and vice versa. It also can help to reduce covert channels and non-MLS traffic analysis of network media transmissions.

The risks to be avoided include, for example (among those noted in Section 5.7), key capture, replay of encrypted commands, key tampering to cause denials of service, and unauthorized replacement of an implementation of escrowed-key crypto with an implementation that undermines the escrow process.

The following list enumerates a collection of illustrative properties that may be desirable for certain crypto implementations, or on which the strength of such implementations depends. Satisfaction of some of these properties is desirable in facilitating countermeasures to the various risks, to whatever extent is desired. However, none of the given properties is absolute, and each must be tailored somewhat to various relevant assumptions and implementation circumstances (such as hardware versus software implementation, and whether the operating environment is an open one or a closed one). Some properties are system properties relating to how the crypto is encapsulated or used, while others pertain specifically to the crypto algorithms or protocols, or to their implementation.

The properties are grouped functionally. The first property is a generic catch-all that relates to risk reduction; it depends on many of the succeeding properties. The next three groups of properties relate to crypto implementations, cryptobinding and authentication, and key escrow. Pervasive auditing is lumped together as a single property, even though it is distributed throughout and associated with most security-relevant functions. The last property involves the security of the underlying infrastructure, which is itself a broad category that encompasses operating systems, database management, networking, with any or all the usual attributes such as confidentiality, integrity, and availability (and in some cases invoking certain crypto mechanisms). This ordering is followed in Table 14, which summarizes these properties; Table 14 also indicates for each property the other properties on which the given property depends. In the case of infrastructure security, specific aspects are noted as applicable.

- **Risk reduction.** All the potential risks summarized in Section 5.7 must be addressed — for example, must be minimized or determined to be acceptable risks.

- **Crypto strength.** Any cryptographic algorithm (or hashing function, in the case of authentication) must be sufficiently strong to resist cryptanalytic attacks and brute-force exhaustion (commensurate with the perceived risks of compromise).

- **Crypto encapsulation.** Any cryptologic implementation must be resistant to subversion, tampering, or other form of compromise, including attacks on its encapsulating environment (for example, operating system, application software, network server, or hardware) and its key management. This property relies critically on the security of the underlying infrastructure.

- **Source protection.** A proprietary or classified crypto module must have its source code protected or isolated, so that it cannot be reimplemented easily in similar or different operating environments. This property is primarily for closed-world applications, such as SKIPJACK.

- **Object protection.** A proprietary or classified crypto module must have its object code or hardware implementation protected, so that it cannot easily be reverse engineered. Implementation could require stealthy encapsulation or tamper-resistant hardware enchipment. This property is primarily for closed-world applications, of which Clipper and Capstone are examples.

- **Crypto integrity.** A crypto module must be difficult to replace, tamper, bypass, spoof, or otherwise compromise. (However, replacement may be possible under certain carefully controlled circumstances, with proper authorization. See the next item.) In addition, each crypto-relevant module must be authenticated as being the desired module — for example, to prevent substitution of a trapdoored module or a Trojan horse that makes surreptitious uses of the unencrypted information or the keys.

- **Operational flexibility.** Whether embedded in a larger subsystem or modularly pluggable, a crypto module must be amenable to installation, initialization, and operation by authorized persons or agents. It must be replaceable or configurable only by authorized agents, in terms of its source code, object code, or hardware components, under specifically designated circumstances (for example, at installation or system recompilation), as permitted.

- **Crypto usability.** Crypto implementations and the very existence of crypto should be essentially invisible within their encapsulation, primarily for ease of use. In particular, key management should be of low visibility.

- **Secure key management.** Crypto keying information must be protected with great care. Establishing keys through key exchange or key agreement must be largely hidden, irrespective of its complexity, subjected to stringent controls, and at the same time very difficult to intercept, spoof, or jam.

- **Authentication nonspoofability.** Crypto used for cryptobinding, authentication, and integrity checking must be nonspoofable (for example, it should hinder reverse engineering of a hash algorithm or a digital signature, masquerading, forging, using replay attacks, or simulating a positive acknowledgment from an authentication server).

- **Authentication nonrepudiatability.** Crypto used for cryptobinding, authentication, and integrity checking must be nonrepudiatable, with whatever degree of certainty is required.

- **Escrowing process integrity.** If key escrow is required, all established keys must be properly escrowed (under a suitable definition of "properly"). This is a special case of crypto integrity.
- **Escrowed-key confidentiality.** Escrowed keys must be strongly protected against loss of confidentiality (for example, through system failures or human misbehavior, whether accidental or intentional, or misuse of the court-authorized escrowed-key decryption process).
- **Escrowed-key nonbypassability.** The validity of an escrowed key must not be superseded or overridden by the use of a key that is not properly escrowed. For example, the keys that are escrowed must be the keys that are actually used. (Note that this property does not preclude superencryption.)
- **Escrowed-key decryption nonsubvertibility.** Alterations in transmitted or stored information (as in the attacks on the Clipper law-enforcement access fields described by Matt Blaze [21]) must not be able to subvert the validity of the decryption process with respect to the properly escrowed keys.
- **Escrowed decryption confidentiality.** The authorized decryption process performed using properly escrowed keys must be strongly protected against loss of confidentiality (for example, through system failures or human misbehavior relating to the decryption processes, whether accidental or intentional, or through other misuse).
- **Pervasive auditing.** All security-relevant crypto operations (for example, crypto module alterations, key changes, configuration changes, escrow retrievals, and emergency overrides) must be controlled, restricted to properly authenticated agents, and thoroughly audited. The audit trails must be nontamperable. If any compromises occur, the audit trails must indicate exactly what has happened.
- **Infrastructure security.** Infrastructure security refers to any underlying mechanisms and operational procedures that are not necessarily directly crypto related, but whose misbehavior could undermine some of the crypto properties. Infrastructure security implies the prevention of such misbehavior. It may include operating system security, database security, network security physical security, and appropriate human behavior. It may apply only to selected components, such as network servers or authentication servers, some of which may themselves require crypto (although typically not recursively). Auditing must also apply to security-relevant operations in the infrastructure.

Of these properties, some are very difficult to formalize precisely, partly because they are (intentionally) stated imprecisely, but more often because they are extrinsic properties — that is, intrinsically unformalizable within the scope of other system properties or concepts. Examples of extrinsic properties are statements about cryptanalytic complexity, such as the difficulty of factoring a product of large primes, or whether any shortcuts exist to an exhaustive attack. Another example is escrowed-key decrypt integrity, which relies in part on the honesty and integrity of escrow officials. Overall, the security of crypto-based systems may depend critically on the validity of somewhat intangible properties as well as explicit technological properties. Some properties are indeed formalizable, such as those relating to BAN-logic statements about who (actually or anthropomorphically) knows what at what time in an authentication protocol, and who can do what to

Table 14: Dependencies among crypto-related properties

Property	Acronym	Depends on (among others)
Risk reduction	RR	CS, CE, CI, SKM, ANS, ISS, etc.
Crypto strength	CS	Algorithms, factoring
Crypto encapsulation	CE	ISS (encapsulating)
Source protection	SP	ANS, ISS (source sites)
Object protection	OP	ANS, ISS (object sites)
Crypto integrity	CI	ANS, ISS (physical/logical security)
Operational flexibility	OF	Operational practice
Crypto usability	CU	System/network architecture
Secure key management	SKM	ANS, CS, CE, CI, ISS (key handling)
Authentication nonspoofability	ANS	CS, CE, CI, ISS
Authentication nonrepudiatability	ANR	CS, CE, AUD, ISS
Escrowing process integrity	EPI	ANS, CI, Escrow schema, ISS
Escrowed-key confidentiality	EKC	ANS, ISS (especially procedures)
Escrowed-key nonbypassability	EKN	ANS, CI, Escrow schema, ISS
Escrowed decryption nonsubvertibility	EDN	ANS, Escrow schema
Escrowed decryption confidentiality	EDC	ANS, CS, CE, EKC, AUD, ISS, people
Pervasive auditing	AUD	ANS, ISS, Escrow schema
Infrastructure security	ISS	Some crypto, some people!

whom. But again, the satisfaction of those properties may actually depend on extrinsic properties that are not usually included in the formalization. Thus, we need to address a middle ground in which certain properties are formalizable, but only in terms of explicitly stated assumptions about other properties that may be either intrinsic or extrinsic. Examples of such underlying assumptions relate to whether an underlying operating system has a flaw that would permit crypto compromise, whether a public key can be factored within a particular time period, whether a key compromise much later in time can result in a serious retroactive compromise, or whether a particular human being is trustworthy. Furthermore, there may be additional factors relating only to the intended use of a particular mechanism, such as the presence of superencryption whose intent is to defeat the intent of the key-escrow process. It is very difficult to formalize intent and other socially motivated factors. However, certain assumptions about them can be stated and used in subsequent analyses.

For ease of understanding, Table 14 lumps together properties that may actually arise at different layers of abstraction in any particular implementation. This is an oversimplification that appears to cause loops in the dependence ordering, although those loops do not actually exist in reality. For example, in the large, the infrastructure may depend on crypto for its implementation, while the integrity of the crypto depends on the infrastructure. More explicitly, crypto encapsulation depends on certain aspects of the security of the infrastructure, whereas particular functions of the infrastructure may depend on crypto encapsulation. However, a careful ordering of hierarchical layers in terms of specific abstractions at particular layers avoids the existence of circular dependencies. What is certainly represented in the table is the fact that most crypto-related functionality depends on the security of its infrastructure.

There is always a temptation in formal analyses to concentrate on those attributes that are easily formalizable, and to ignore those that are not. This situation resembles the story about the person whose keys were lost in a field, but who is looking for them under the streetlight — because that is where the light is. The challenge, of course, is to make all the relevant properties and assumptions explicit, and to reason accordingly.

11.4 RISSC Crypto-Based Authentication Properties

Section 11.3 includes two basic properties relating to crypto-based authentication, namely, nonspoofability (ANS) and nonrepudiatability (ANR). Table 14 summarizes some of the other properties on which those properties depend. Those relationships are now explored in greater detail.

Nonspoofability (ANS) `depends on`

- Crypto strength of the cryptobinding, including nonreversibility of hashing algorithms (such as MD4 [197]) and cryptographic seals such as are used in RSA (Rivest-Shamir-Adleman) authentication or DSS (the Digital Signature Standard) (CS)
- Crypto encapsulation, including the protection of the cryptobinding keys and protection of any internal hashing and crypto implementations from spurious use (CE)
- Crypto integrity, particularly the ability to tamper with the authenticating code (or hardware) that could give the appearance of authentication when in fact the cryptobinding information is actually bogus (CI)
- Infrastructure security, including prevention of subversion of the cryptobinding and authenticating processes through operating-system penetrations, and techniques such as nonces for preventing replay attacks (ISS). This includes preventing forgeries, simulations of positive acknowledgments from an authentication server, and users masquerading as other users. Infrastructure security is particularly important when cryptobinding is done on a shared end-user system (which is avoided in the RISSC subfamilies [[1a]], [[1b]], and [[2a]]) or on a shared authentication server.

Nonrepudiatability (ANR) `depends on`

- Crypto strength of the cryptobinding (CS), although this is somewhat less important for nonrepudiatability than for nonspoofability
- Crypto encapsulation, including high-assurance demonstrations of the protection of the cryptobinding keys and protection of any internal hashing and crypto implementations from spurious use (CE)
- Avoidance of tampering with the audit-data collection and analysis that could mask a nonauthentication or to create the false impression of an authentication, and generally high assurance that audit trails are noncompromisible (AUD)
- Infrastructure security, particularly in protecting against internal misuse that might result in the appearance of spoofing when in fact no spoofing had occurred (ISS)

See also Kailar, Gligor, and Gong [102], for issues of dependence in crypto protocols.

11.5 RISSC Key-Escrowed Crypto Properties

Section 11.3 includes five basic properties relating to escrowed-key encryption and decryption, namely, escrowing process integrity (EPI), escrowed-key confidentiality (EKC), escrowed-key non-bypassability (EKN), escrowed decryption nonsubvertibility (EDN), and escrowed decryption confidentiality (EDC). Table 14 summarizes some of the other properties on which those properties depend.

All five of these escrow-related properties rely on adequate infrastructure, including the the security of the entire escrow process and nonspoofability of authentication throughout the escrowing process, key storage, the supposedly authorized use of the escrowed keys, and subsequent to authorized decryption using the escrowed keys. People involved in the escrow process must also be trustworthy, including algorithm and protocol designers, chip designers and fabricators, escrow operators, and authorized decrypting agents. In the case of the escrow process associated with Clipper, the split key implies that multiple individuals or processes would have to be compromised during the process, although the decrypted results might be vulnerable to single-party misuse. Modeling of the entire process must make assumptions of trustworthiness explicit, including those relating to people. The `depends on` relation rather than the `depends upon` relation is relevant whenever a multikey cryptosystem is involved that can mask the unreliability or untrustworthiness of some number of components (human or otherwise).

11.6 RISSC Crypto Auditing Properties

Section 11.3 includes a catch-all property relating to pervasive auditing. A detailed discussion of the requirements for securing an auditing environment is given in Appendix B, with particular attention to making that environment tamperproof. Much of that discussion is relevant here.

11.7 RISSC Crypto Infrastructure Properties

The crypto-related properties enumerated in Section 11.3 end with the security of the infrastructure, which ultimately depends on the satisfaction of the requirements of Sections 4.2 and 11.1, among others.

11.8 Analyzing Crypto Implementations in the Large

It remains an enormous challenge to completely model all of these dependencies involving crypto implementations, within the computer-communication infrastructure and in the context of an entire complex of systems and networks. In this report, we have merely sketched an outline of how that might be carried out, in hopes that future efforts may pursue this challenge further. We believe this challenge to be a very important one.

Part Four: CONCLUSIONS

12 Conclusions and Recommendations

This report suggests that the attainment of significant security in realistic computer systems and networks demands an approach that thoroughly integrates two elements:

1. *Architecture.* An extremely disciplined approach to architecture and system development is desirable, making constructive use of existing system and network components wherever possible, encouraging the development of certain trusted subsystems that contribute to secure infrastructures, and combining all components with constructively beneficial interconnections, so that the resulting systems can be readily configured out of commercially available products and readily evaluated as combinations of subsystems rather than having to evaluate entire systems. It is desirable to avoid hodge-podge or *ad-hoc* interconnections of nonsecure components within nonsecure infrastructures.

2. *Formal methods.* Formal methods should be applied judiciously to the most critical system aspects, where the benefits are greatest in terms of dependability, assurance, and performance. However, they must be applied so that no seemingly unimportant aspects can compromise the more critical aspects. The formal methods must be able to span the full range of mechanisms, properties, and analyses necessary to evaluate the security of supposedly secure distributed systems — including their operating systems, their networking software, and their network media. Properties on which critical functionality depends should be represented in the formalizations, even if they are extrinsic and cannot be proven; at least, those properties can then become explicit, and the manner in which they are depended on can be represented. Dependencies on seemingly less critical functionality and properties should also be made explicit, so that all assumptions can be checked for validity — to ensure that their dependencies are indeed noncritical (for example, in the sense of multilevel integrity where there must be no dependence on less trustworthy entities).

We believe that this two-pronged approach can dramatically improve the security attainable. Many of the constituent steps in this process already exist in the small. What is most needed is the unification of all relevant techniques within an overall framework that permits reasoning about entire systems and networks of systems in the large, in terms of their component subsystems, and that also permits the process to be carried on iteratively into lower layers of abstraction.

12.1 General Conclusions

Certain government organizations (particularly the National Computer Security Center in the United States and CESG in the United Kingdom) have long hoped that the research and development communities would be able to employ formal methods as a cost-effective and practical approach to the development of realistically secure computer systems. Formal methods have considerable potential throughout the development cycle, with respect to requirements, specifications, and implementations, in software, hardware, and microcode, for a wide range of systems and networks. However, this potential has been very slow to reach fruition.

- For the optimistic reader, this report concludes that there is still an enormous untapped potential for formal methods in the development of systems and networks capable of satisfying stringent security requirements. Whereas much of the prior work has been concerned with properties of components and subsystems in the small, there are enormous benefits that can result from being able to express and reason about system properties on a more global basis, for example, reasoning about integrated systems based on properties of their constituent subsystems. Furthermore, significant successes of formal methods reported in other disciplines are encouraging as harbingers of what should be possible in applications of formal methods to security.

- A pessimistic reader might counter with a comment that formal methods have thus far been too difficult and too expensive to apply, even to components and small subsystems; why then should we expect such techniques to be applied successfully to systems in the large?

One response to the pessimistic view is that the focus in the past has centered on low-layer properties such as multilevel security of kernels, without sufficiently addressing all of the relevant properties of trusted computing bases using those kernels. The TCSEC criteria have not encouraged representation and evaluation of application-specific properties, being concerned primarily with lower-layer trusted computing bases. Another response is that formal methods must be applicable to higher-layer properties as well as low-layer properties and their interrelationships, if those methods are to be really useful. A further response is that the overemphasis on preservation of properties (such as restrictiveness and strict-sense composability) has led to the neglect of work on the transformations of properties under generalized compositions. For example, what happens when different components satisfying nonidentical properties are grouped in the same subsystem or networked together?

The practical long-term viability of the use of formal methods for security will be determined in part on these methods being applied successfully to real systems, in the near future. As noted in Section 9, there are some genuine successes applied to real systems with stringent requirements for reliability, fault tolerance, and safety. There are also a few rather more limited successes relating to secure computer and communications systems, and some recently renewed interest in further pursuing such applications. Some successes are urgently needed in which formal methods are applied to real security-critical systems.

It is a conclusion of this report that there has been overemphasis in the past on Orange-book-like kernel properties and on property-preserving transformations, and that new paradigms are essential for much more pronounced successes in the future. In addition, the modeling of systems derived from combinations of subsystems (simple composition, bilateral associations, hierarchical layering, and networking) desperately need to be represented more explicitly in realistic security criteria. An approach that reflects the intent of the appendix to the original unreleased draft version of Trusted Network Interface document [150] (the Red Book) is needed, representing the real complexities of developing distributed systems and networks, and providing a basis for applying formal methods thereto.

Some of the conclusions reached in this report with respect to security echo those reached by Rushby [209] in the context of reliability and safety in airborne systems. In particular, with respect to both this report and Rushby's, it appears that the following conclusions are in order:

- *Avoiding vulnerabilities.* Formal methods are best applied when they dramatically improve the process of detecting and eliminating significant system flaws and vulnerabilities. To this end, it is appropriate to validate that certain critical requirements soundly represent their intended purpose, and to demonstrate by formal (or even semiformal) reasoning that critical properties are satisfied by particular components or subsystems with respect to their formal specifications. It is also appropriate to examine critical aspects of the resulting system and of the implementation process itself, seeking to derive or prove properties of the system in terms of properties of its subsystems. The essence of this process is to provide cumulatively increasing confidence in the system design and its implementation, by identifying inadequate requirements and flawed designs, and overcoming them. Steps that do not add substantively to this process are less important.

- *Correctness.* It is unwise to overemphasize the process of trying to prove code correctness. Code proofs are premature unless the requirements and the design are demonstrably sound. Thus, efforts to increase the assurance with which system dependability can be attained by the implementation should be deferred, at least until no further flaws can be found in the requirements and the design. However, at that point, code proofs and other forms of analyzing the implementation (see the second bulleted item in Section 12.4) can be very valuable — especially if they are able to show the absence of nonspecified effects such as surreptitious Trojan horses. On the other hand, there is an enormous potential for the use of formal methods in hardware implementations — for example, in specifications, mask layouts, and fabrication.

- *Top-to-bottom analysis.* Past efforts on proving a continuous refinement thread from top-level requirements through the detailed design to the software and finally down to the hardware are of considerable theoretical interest, particularly when applied to large and complex systems. However, those efforts must be considered as overkill unless it can be demonstrated that all the relevant critical paths can thus be encompassed and that no serious vulnerabilities can exist in other threads. Then, the top-to-bottom analysis can be very compelling.

- *Transformations.* Greater emphasis is needed on the property transformations that result from nontrivial compositions, hierarchical layerings, and interpositions of mediators such as firewall systems and trusted guards. The transformational approach should also permit parametric architectural representations, for example, what simplifications or complications might result when a particular RISSC architecture undergoes a particular change in design, such as making the file server multilevel secure instead of having multiple single-level file servers. The effects should be formally derivable.

- *Methodology.* The choices of methodologies, specification languages, and programming languages are important to the success of a development effort and to the effective application of formal methods. However, those choices are less critical if the architecture is poorly chosen and if the properties to be satisfied are not appropriate — for example, seriously incomplete, or too abstract, or too low-level. Thus, considerable effort must be devoted to establishing an architecture that is amenable to the use of formal methods. However, there is also a danger that premature choices of methodology and approach will lock the development into a nonconstructive path. Thus, it is essential that all these factors be considered early in the system development process.

- *Research and development.* Consistent with the ability to apply formal methods to critical aspects of complex and critical systems, research efforts must continue to explore the frontiers of the technology, and development efforts must be carried out that employ formal methods constructively as suggested in these conclusions. Some specific suggestions for future R&D are given in Sections 12.2, 12.3, and 12.4.

There have been many advances in formal methods in the past twenty years. However, major successes are still awaited in the fruitful application of these methods. We conclude that considerable potential remains untapped for formal methods applied to security, and that we are now actually much closer to realizing that potential. Many of the pieces of the puzzle — theory, methods, and tools — are now in place. The combination of approaches suggested here could help, if the desired paradigm shifts are taken and if the following recommendations are considered.

12.2 Recommendations for Applying Formal Methods to Architectures

Section 12.2 provides a collection of recommendations for the future, relating to the use of formal methods applicable to system and network architectures intended to satisfy critical security requirements.

- *RISSC concepts.* More emphasis should be placed on RISSC-like systems, for both multilevel and single-level security applications. The attempt to get developers of secure computer systems to produce commercially viable B2, B3, and A1 systems has been rather dismal. Adoption of the RISSC approach would permit the rapid development of multilevel-secure system complexes using off-the-shelf single-level end-user systems, and would greatly increase the ability to configure MLS systems to suit particular application needs, to evaluate those systems, and to evolve them over time. This approach can also be very useful in developing single-level systems with dependence on demonstrably trustworthy network servers, file servers, and authentication servers, somewhat independent of the specifics of the particular end-user systems, vendors, operating systems, and application software. Overall, the RISSC approach would increase the ease with which heterogeneous system complexes can be produced. Whether or not such an approach can simplify the formal analysis remains to be demonstrated. We believe that, by removing much of the trustworthiness from end-user systems, the analysis will indeed be less intricate when all security-relevant factors are considered — for single-level security as well as multilevel security. However, perhaps the biggest benefit would be the ability to obtain commercial MLS operating environments as configurations of off-the-shelf end-user systems.

- *Properties, interrelationships, and combining effects.* More emphasis should be devoted rather pervasively to important properties (including, but by no means limited to, MLS) and to complete systems (including distributed and networked systems), although efforts relating to properties of their component modules are also important. Although more difficult to handle, interrelations between security and other types of properties (such as fault tolerance, real-time performance, and emergency overrides) should also be represented. A system may not be adequately secure overall if its reliability is in doubt, or if its security properties can be (or indeed *must* be) compromised under real-time stringencies or other emergencies such as lost passwords or crypto keys. Furthermore, formal methods should be interoperable —

for example, in the sense that techniques for module specification, network protocols, and crypto should be able to be mixed together compatibly. Greatly improved understanding of generalized composition of subsystems is required. Increased understanding is also necessary with respect to the ways in which subsystem properties can be preserved or transformed under generalized composition.

- *Assumptions.* More emphasis should be placed on making explicit all of the otherwise unstated assumptions, functional dependencies, and property dependencies underlying any system to be formally specified and analyzed. Assumptions of proper human behavior are essential, and should be factored into the analysis of any critical system so that the dependencies on that assumed behavior can show up as part of the reasoning process. Similarly, environmental assumptions should be represented. Just as Byzantine techniques can be used to withstand the misbehavior of arbitrarily faulty components, so is it desirable to use such techniques advantageously to overcome undesired human and system behaviors. For example, multikey crypto [52, 142] and error-correcting coding could be useful in this regard.

- *Criteria.* The TCSEC/ITSEC/CTCPEC efforts are all very useful iterations in a long process of attempting to create useful security criteria. We recognize that any set of criteria is likely to be incomplete, and also on one hand difficult to interpret or on the other hand overly explicit and restrictive. Nevertheless, a successor to the Orange Book is urgently needed, possibly including a major upgrading of the Red Book to address distributed systems and networks more explicitly.

- *Tools.* More emphasis is needed on the realistic usability of formal methods and their supporting tools. There are many potentially useful requirement languages and specification languages, and supporting tools. However, further effort is needed to incorporate them into better human-engineered development environments into which the mechanizations of formal methods have been carefully integrated, and in which the human interfaces to the tools have been driven by ease-of-use requirements. Different formalisms must interrelate, so that it is possible to reason in the large about system and network properties such as operating system security, crypto encapsulation, crypto strength, and network reliability. For real-time systems, incorporation of temporal logics or other approaches to representing real-time issues would also be desirable. Readable and easily understood tutorial documents such as [30, 212] are essential for those who are able to use formal-methods environments. Carefully worked and well-documented examples of formal methods applied to real systems are absolutely essential.

- *Education, experience, and training.* More emphasis should be placed on the education of students and industry employees to help them appreciate the potential practical utility of formal methods applied to real systems, as well as the theoretical beauty (for example, see [63]). For this to be successful, teachers must have a better understanding of the fundamental issues of discrete mathematics and logic, and those issues must permeate the instruction and training.

  The unfortunate lack of an appropriate systems perspective represents a fundamental problem at the present time in education and training efforts. Computer-science curricula are for the most part sorely out of touch with the needs of developers of critical systems and complex applications. Programming and formal methods are generally taught in the small, and the students develop very little system sense. Good software engineering (as opposed to overly simplistic panaceas) is rarely emphasized, and seems to be considered more or less irrelevant in favor of a predilection toward programming in the small. Unfortunately, the practical needs of system developers seem to be the tail trying to wag the dog.[19] Our *uni*versities must embody more *di*versities (or even *multi*versities), teaching much more than just C, Unix, Windows, and HTML. The situation in industry is generally not much better than in universities, the result being that complex systems and networks are often poorly conceived and poorly developed by people with narrowing rather than broadening experiences. A greater appreciation of the need for system perspectives should permeate education at all levels.

12.3 Near-Term Recommendations

The conclusions of Sections 12.1 and 12.2 include some recommendations for immediate action, and others that are more appropriate in the long term. The former are considered here, the latter in Section 12.4.

- *Illustrative worked examples.* It is very important to have more examples of how formal methods can be applied to real systems, during the system development rather than *post-hoc* retrofits. These examples should be carefully documented, thoroughly worked, and — at least initially — applied up-front in the development process to real systems (commercial or custom) — that is, applied primarily to requirements and specifications and to their analyses. Some of these examples will naturally be able to extend to hierarchical mappings among different layers and explicit representations of properties at different layers and explicit representations of the dependencies on lower layers. Suggested examples might include the following:

  - A Rushby-style separation kernel together with some applications that depend on it, complete with all privileged exceptions, demonstrating clearly how the technology can be applied, how the higher-layer properties depend on the kernel properties, and how formal analysis can be effectively carried out for more than just the kernel
  - Complete modeling and analysis of a crypto encapsulation (whether escrowed or not) and pursuit of the concepts outlined in Section 11.3 — including making explicit all of the assumptions on which the confidentiality and integrity of the crypto-based security depend, and reasoning based on those assumptions
  - The effort begun in [101] should be continued, in several directions — (1) refining the existing representation of the MISSI security policy, (2) examining the consistency of the detailed design with those requirements, and (3) possibly attempting some reasoning about the implementation, but not necessarily code-consistency proofs (see the second bulleted item in Section 12.4). Specification of the security requirements is clearly a valuable activity. However, there appears to be good potential for getting further mileage out of the effort already begun. An effort to track ongoing changes in the requirements, design, and implementation could also be valuable.

  In these efforts, we recommend that research people with intimate knowledge of the formal methods and the tools be heavily involved, together with systems experts.

[19] There are of course some exceptions, such as efforts by Leveson at the University of Washington and by Knight, Prey, and Wulf [111] at the University of Virginia. Although the Virginia undergraduate program devotes relatively little attention to formal methods, it nevertheless attempts to provide a more rigorous treatment of software engineering, completely integrated into the curriculum.

- *Model checking.* Model checking has considerable potential. Efforts specifically oriented to applying model checking to security are desirable. Careful examination of the relative benefits and areas of applicability is needed.

- *Integration of model checking.* A somewhat longer-term but still near-term goal involves incorporating model-checking tools compatibly into existing analysis tools, such as is being pursued at SRI and the University of British Columbia (among others).

- *Integration of various approaches.* The integration of methodologies, specification languages, formal methods, and their corresponding toolsets within a common framework would be of considerable value to system developers desiring to use formal methods pervasively. For developers of secure real-time systems, some sort of temporal logic should be included.

- *Modularization and interoperability of different tools.* At present, SRI, ORA, CLI, and many others have tools for formal methods that operate in their own environments. It would be very useful to have modularized components of these toolsets that could interoperate across institutional boundaries.

12.4 Long-Term Recommendations

The conclusions of Sections 12.1 and 12.2 also suggest various actions with a longer-term perspective.

We suggest in this report that considerable gains can be achieved by taking a fresh view of secure-system architectures and applying formal methods selectively, where the payoffs are greatest. We also suggest that the choices of methodologies, formal methods, and languages are important, but somewhat less so than the architectures and the emphasis on up-front uses of formal methods. However, there is still much worthwhile research to be done, particularly where it can reduce the risks of system developments, increase the chances of success, and reduce the cost and time required for system development.

- *Reasoning about generalized compositions.* Significant effort should be devoted to a theory of generalized compositions and the property transformations they induce. This work should encompass system and network configurations generally, the interconnections involved in RISSC architectures, the interposition of trusted gateways, networking, and the necessary criteria to facilitate evaluation of modular systems.

  A particularly fruitful approach in this direction that seems to have considerable potential is noted in Section 6.2, namely the merging of the work of Moriconi, Qian, and Riemenschneider [145, 146] with PVS, to greatly expand the ability to reason about complex system designs.

- *Reasoning about implementations.* We are by no means opposed to proofs that an implementation is consistent with its specifications, despite earlier comments about the relatively bigger apparent payoffs resulting from up-front uses of formal methods. Some emphasis should also be placed on carrying out a formal-methods approach that extends into the code or microcode, especially if those code proofs can be formally related to the specifications and shown to cover the critical requirements under carefully specified assumptions. However, research should also be carried out to explore other approaches to reasoning about implementations that fall short of full code proofs. For example, it should be possible to reason about program changes and configuration control over implementations without having to reason about the programs themselves. (A precedent for that exists with respect to reasoning about designs in earlier work of Moriconi [144], which provides a framework for reasoning about design changes.) As in the corresponding near-term efforts, we recommend that research people with intimate knowledge of the formal methods and the tools be heavily involved together with systems experts and programmers.

- *More elaborate real examples.* Further worked examples of formal methods applied to real systems are needed, above and beyond the up-front examples suggested in Section 12.3. An ambitious system might involve a stem-to-stern specification and analysis of a distributed system, encompassing all of the necessary assumptions on the infrastructure and end-user systems, including all relevant properties of the operating systems, servers, crypto encapsulations, and people involved in operations (including people in the key-management loop and key-escrow retrievals). However, such an effort should not be attempted all at once; rather, it should use an incremental approach whereby the pieces can emerge separately and then be combined. Again, these examples should be realistic, thoroughly documented, and well motivated.

- *Comparative studies.* Comparative studies are needed to explore the explicit benefits that can result from consistent use of formal methods throughout a system development. Similar studies are also desirable to determine whether *post-hoc* uses of formal methods have any real value — that is, where modeling and analysis are carried out after a system development is well underway or even completed. These studies probably cannot be statistically meaningful, because of the many variables involved and the continual production pressures during development, but nevertheless some experiential learning is needed relating to the efficacy of using formal methods in large and realistic systems.

12.5 Final Remarks

The potential benefits of formal methods remain undiminished. The need for formal methods in the specification and analysis of critical systems and system components remains enormous. In the light of past events — system flaws and detected vulnerabilities, system failures, experienced penetrations, and flagrant system misuses — formal methods remain an essential part of the system development and assurance process. Their systematic use at appropriate places throughout the system life cycle can be far more productive than it has been in the past.

EPILOGUE

Epilogue — A Quote from Edsger W. Dijkstra

Just as I was completing this report, I received the latest in the long-running series of informal memos from Edsger W. Dijkstra, a long-time advocate of the importance of design structure [55] and the value of designing correctly in the first place, *before* programming [56]. The following excerpts serendipitously echo many of the main conclusions presented here, and provide a fitting end to the main text of this report.

> Being a better programmer means being able to design more effective and trustworthy programs and knowing how to do that efficiently.
>
> I see no meaningful difference between programming methodology and mathematical methodology in general.
>
> In a cruel twist of history, however, American society has chosen precisely the 20th Century to become more a-mathematical... As a result, Program Design is prevented from becoming a subdiscipline of Computing Science. There is considerable concern for correctness, but almost all of it has been directed towards *a posteriori* program verification because, again, that more readily appeals to the dream of complete automation. But, of course, many — I included — regard *a posteriori* verification as putting the cart before the horse because the whole procedure of programming first and verifying later raises the burning question where the verifiable program comes from. If the latter has been derived, verification is no more than checking the derivation. And in the mean time, programming methodology — renamed "software engineering" — has become the happy hunting-ground for the gurus and the quacks.
>
> Excerpted from *Edsger W. Dijkstra,* Why American Computing Science seems incurable, EWD 1209-0, Austin, Texas, 26 August 1995.

APPENDICES

A Summary of Architectural Families

Appendix A is based on [158], with some changes and extensions. It summarizes a range of architecture families. and provides a broad comparative view of what is really a multidimensional architectural spectrum.

Of particular interest from the RISSC point of view are the cases [[1a]] and [[2a]], as well as [[1b]] (all three of which are considered in Section 11 and in Table 12 in Section 11.2). These cases are the most assiduous in their adherence to the RISSC concept.

Because of their dependence on multilevel-secure file servers, cases [[3a] and [[3b] are somewhat less RISSC-oriented, but are still important — particularly whenever high-assurance MLS file servers are available. Family [[3]] is also considered in Section 11) and in Table 13 in Section 11.2.

For ease of description, we define an abstract notation for expressing configuration attributes. The architectures considered here include end-user systems (User), file servers (F), network servers (Z), network transmission media (X), and authentication functionality (A).

In a simple implementation, the various servers of each type could be collocated with the end-user systems; however, to minimize the extent to which the end-user systems must be trusted, the servers may preferably be on systems distinct from (most of) the end-user systems.

The notation used here is starkly simple, incomplete, and subject to wide ranges of differing requirements within each attribute. There are typically many shades of gray within a decidedly non-black-and-white multidimensional spectrum. However, although no attempt is made to capture notationally the relative degrees of trustworthiness within any give attribute, the notation enables us to identify and contrast some of the basic options without having to delve into many details that are less important to the challenge of attaining robust crypto in software.

An entire system or network of systems may support multilevel security (MLS) or only a single level. However, there are many different architectural types that enforce MLS system-wide or network-wide, as discussed here.

An end-user system may be a single-level single-user system (User-S1), a single-level multiuser system (User-Sn), a multilevel-secure single-user system (User-M1), or a multilevel-secure multiuser system (User-Mn).

Realistically, a **single-user system** as defined here could actually accommodate multiple simultaneous users, as long as those users are all mutually trustworthy and mutually trusting. That is, they are equivalent users from a protection point of view, although they would presumably have individual user identities and separate accountability. The essence of the single-user notion here is simply that security does not rely critically on interuser isolation. For simplicity, we refer to such systems as single-user systems, although the more cumbersome term "single-user-group systems" might be more accurate in cases in which protection-equivalent users are allowed to coexist.

In a networked environment, authentication functionality may be centralized into a collection of authentication servers or distributed among end-user operating systems. A commonality of all the nondegenerate approaches is that some component of authentication must to some extent be trustworthy (AT) rather than totally untrustworthy (AU). However, there are various authentication attributes each of which may or may not be trustworthy, including, respectively, the following:

CT or CU for key confidentiality
IT or IU for key integrity
ST or SU for antispoofing
RT or RU for nonrepudiation
DT or DU for nondenial of service

The attribute CT is essential for most but not all the crypto uses discussed here; the attribute IT is often necessary as well. When no untrustworthy authentication attributes are explicitly specified, AT is assumed to include at least some measures to address the authentication attributes CT, IT, and ST. The other attributes of trustworthiness may or may not be important, according to the nature of the application.

A file server may enforce multilevel security (FM) or may be single-level (FS). In either case, the file server may store files in either an encrypted form (FN, for eNcrypted) or an unencrypted form (FP, for Plaintext). In those cases in which storage encryption is used, the *cryption* (that is, encryption and decryption) may be provided by the client (FNc), by network servers (FNn), or by a file server itself (FNs). Multiple forms of cryption may be used together, such as communication cryption (ZT-FNn) on top of client-provided file cryption (FT-FNc).[20] For simplicity, encrypted system-to-system file transfers (for example, using the IP FTP protocol) are also represented by FNc or FNn even though a file server is not involved.

Carrying out the cryption would typically be largely invisible to all components other than the one that actually performs those functions.

Each file server may be trustworthy (FT) or untrustworthy (FU) with respect to each of several requirements:

MT or MU for multilevel security
CT or CU for single-level confidentiality
IT or IU for file integrity
DT or DU for preventing denials of service

If no untrustworthy file server attributes are specified, FT is assumed to include some measures to address each of these attributes, except for MT, which is always identified explicitly. Thus, the configuration FP-FT-FM-MT-CT-IT-DU or simply FP-FT-FM-MT-DU would denote a multilevel-secure file server storing unencrypted information, trustworthy with respect to enforcing multilevel security, discretionary confidentiality, and integrity, but untrusted for preventing denials of service. Similarly, FNc-FU-FS-CU-IU-DU or equivalently FNc-FU-FS would denote a single-level file server storing encrypted information provided by the client, and untrusted for single-level confidentiality, integrity, and nondenial of service. In such a case, the encryption provided by the client might ensure single-level confidentiality and integrity with respect to the file server, irrespective of what actions the file server might take.

[20] Multiple encryption causes a slight terminology problem that is easily resolved contextually. For example, consider the combination of client-provided file encryption and network-server-provided network encryption. To the end-user system, the client-provided encrypted form is encrypted text; to the network server, that encrypted form is plaintext.

The distinction between the network transmission media (X) and the network servers (Z) is significant. The servers may be trustworthy with respect to providing multilevel-security separation (ZT-MT), while the communication media may be untrustworthy (XU-MU-CU-IU) precisely because of the use of crypto by the network servers. Cryption provided by trustworthy network servers is denoted by ZN-ZT.

Network transmission media may be untrustworthy (XT) or untrustworthy (XU) with respect to each of several requirements:

MT or MU for multilevel security
CT or CU for single-level confidentiality
IT or IU for message integrity
DT or DU for preventing denials of service

If no untrustworthy network attributes are specified, XT is assumed to include some measures to address each of these transmission attributes except for MT, which is always identified explicitly. However, a transmission configuration may be basically untrustworthy (XU) if the network servers are suitably trustworthy and the network traffic is encrypted. In addition, some trustworthiness for nondenial of service (DT) may be obtained by error detection, multiple routing, and retry, in the absence of any other trustworthiness of the network media. A completely enclosed and protected wireway is an example of a trustworthy medium.

Network servers may be untrustworthy (ZT) or untrustworthy (ZU) with respect to each of the same requirements:

MT or MU for multilevel security
CT or CU for single-level confidentiality
IT or IU for message integrity
DT or DU for preventing denials of service

Whenever files and messages are not encrypted (and possibly even if they are encrypted), network servers may provide their own cryption, invisible to all components except each other. Thus, we also differentiate between network-server-provided cryption (ZN) and networking that adds no further encryption (ZP). If no untrustworthy network-server attributes are specified, ZT is assumed to include some measures to address each of these attributes except for MT, which is always identified explicitly.

A distinction is made between network-provided file storage cryption (FNn) and network-provided message cryption (ZN). In the former case, cryption is performed only at the client side of the network, and files are stored in that encrypted form; in the latter case, cryption is performed at each end of the network, and storage would not be encrypted unless otherwise provided by (FNc) or (FNs) functionality.

A distinction is normally made between local networks and nonlocal networks. However, in the present context that distinction may or may not be meaningful. Clearly, wide-area network traffic would normally be encrypted in any sensible networking environment. However, if local-net traffic is within a physically secure (and, if desired, properly shielded) environment and all the local-net taps are occupied by trustworthy systems and individuals, then protection of the local network may be avoided.

Once again, we must understand that there are many shades of trustworthiness. Each of the above 'T' attributes can be thought of as either an ideal to be striven for, or else as whatever can be pragmatically attained.

The art of secure systems and secure networking is essentially one of placing appropriate controls and protections in those facilities where they are needed. There is a sense in which an equivalence exists among various seemingly different architectural views. For example, whether the main class of RISSC systems has single-level file servers or multilevel-secure file servers is perhaps an implementation detail; a formal transformation can be constructed under which the two families of architectures are essentially equivalent. Furthermore, a TCSEC kernel–TCB–application–user layering represents one way of achieving security domains; a layered capability architecture such as the original PSOS design [62, 161] is another; a networking of unsecure systems using trusted gateways is still another. In the last case, whether the gateways are internal or external is more or less incidental. Each of these approaches attempts to provide suitable domain isolation.

The configuration AU,ZP-ZU,XP-XU is more or less today's sorry network state of the art. What is needed for trustworthy distributed systems is a configuration such as AT,ZN-ZT, where XT-CT-IT-DT is effectively a by-product of the network-provided cryption, or possibly AT,ZP-ZT,XP-XU-DT, where client-provided file cryption (FNc) can facilitate trustworthy XT-CT-IT networking without requiring trustworthiness for confidentiality and integrity on the part of the network servers.

The architectural families of systems that might contain crypto implemented in software fall into two basic types, multilevel and single-level secure. Each family has its own potential risks and hardware–software trade-offs. The architectural families enumerated here should be considered as representative points in a multidimensional spectrum of architectures, with end-user systems, authentication, networking, and file servers as different dimensions, and with security levels (multilevel vs. single-level) and the number of cohabiting users as parameters. Those dimensions and parameters may seem superficially independent (which would result in a large number of families); however, only certain combinations are sensible, which helps to reduce the discussion to somewhat sensible proportions.

- Multilevel security. In the first group of families (Sections A.1 through A.6), the overall systems or networks of systems are capable of enforcing multilevel security. Among the six basic families and variants, the first three baseline architectures considered below (Sections A.1 through A.3) have end-user systems that are all single-level rather than multilevel, but in which the overall system still enforces multilevel security. The other three families use multilevel-secure end-user systems. The fourth family (Section A.4) has end-user compartmented-mode workstations, while the others (Sections A.5 through Section A.6) have multilevel-secure multiple-end-user systems, with or without crypto storage, respectively. Among those six families, the first two use only single-level file servers, while the remaining four use multilevel-secure file servers. The second and fifth have client-provided file cryption, while the others do not.

- Conventional systems. In the second group of families (Sections A.7 through A.10), the overall systems do not enforce multilevel security, and everything (e.g., files and networking) operates at a single level throughout. The local operating systems may be for personal computers or for high-end single-level minicomputer or mainframe systems lashed together with single-level secure servers. The systems may be single-user (Section A.7) or multiple-user (Section A.8). The file servers may or may not use encrypted storage, as indicated. To provide a *reductio ad* *absurdum,* Section A.9 includes the low-end systems that one might obtain from off-the-shelf software with minimal concern for security, and Section A.10 considers stand-alone systems with no networking whatsoever.

The first column of Table 15 summarizes the architectural families discussed in the following sections. Section A.11 contrasts the different families and discusses the remaining columns of the table.

Although arbitrary crypto schemes can be used, we assume for simplicity that the illustrative architecture families include software implementations of a symmetric algorithm such as the Digital Encryption Standard (DES) [167] or Rivest's RSA-Data-Security-proprietary RC-4 for communication security, as well as software implementations of public-key crypto used for authentication and key distribution. The public-key schemes might involve RSA [198] or Diffie-Hellman [53], and might in addition employ one of the fair-crypto or multikey algorithms [22, 52, 142, 227] discussed in the appendix to [158], as appropriate for any particular application.

In our discussion, we generally ignore arguments relating to the strength of the crypto itself, because the strength can always be improved — for example, by using different algorithms, different implementations, longer keys, or multiple passes. We consider here primarily the extent to which the crypto and — as needed — its key escrowing can be securely embedded into the architectures.

The emphasis in Sections A.1 through A.9 is on the basic architectural configurations and the relative crypto trustworthiness that can be attained with each configuration. (As noted above, the first six sections consider environments that enforce multilevel security; the remainder do not.)

A.1 Multilevel-Secure Network Interfacing

- MLS,User-S1,AT,ZN-ZT-MT,XU,FP-FS-MU-CT [[1a]]

The first baseline architecture family includes single-level single-end-user systems with a multiplicity of file servers, each of which is a single-level file server for some particular level (and compartment). In [[1a]] the network traffic is encrypted by the trustworthy network servers, and thus the transmission media need not be trustworthy with respect to confidentiality (or for integrity, if encryption is used appropriately).

The network interfaces must be trusted not to violate the multilevel security constraints, that is, they must not allow information to be accessible from a lower security level or incompatible compartment. However, the file servers and the network media need not be trusted to maintain multilevel separation or to protect keying information.

The analysis of Proctor and Neumann [192] applies in this case. That is, multilevel security can be implemented efficiently with off-the-shelf single-level end-user systems. The network servers can ensure multilevel-security separation, and the interuser covert-channel bandwidth can be essentially zero. Read-down can be implemented efficiently, while read-up and write-down are prevented. Some defense against denial-of-service attacks can be provided by the presence of multiple file servers and multiple communication paths, especially for particularly critical levels and categories.

[[1b]] is a variant of [[1a]] that also has single-level end-user systems. However, in [[1b]] each end-user system may have multiple users, that is, potentially competitive users. Network-provided cryption and multilevel separation enforced by the network servers avoid end-user-system concerns for file security.

Table 15: Architectural families

Multilevel-secure overall systems	**Weak links**	**Security**
MLS network interfacing:		
1a. MLS,User-S1,AT,ZN-ZT-MT,XU,FP-FS-MU-CT	AT,S1,ZT	Very good potential
1b. MLS,User-Sn,AT,ZN-ZT-MT,XU,FP-FS-MU-CT	AT,Sn,FT	Sn-C2 too weak
MLS network interfacing with storage crypto:		
2a. MLS,User-S1,AT,ZN-ZT-MT,XU,FNc-FS-MU-CT	AT,S1,ZT	Excellent potential
2b. MLS,User-S1,AT,ZP-ZT-MT,XU,FNc-FS-MU-CT	AT,S1,ZT	Very good potential
2c. MLS,User-Sn,AT,ZN-ZT-MT,XU,FNc-FS-MU-CT	AT,Sn,ZT	Sn-C2 inadequate
2d. MLS,User-Sn,AT,ZP-ZT-MT,XU,FNc-FS-MU-CT	AT,Sn,ZT	Sn-C2 inadequate
MLS network interfacing with MLS file servers:		
3a. MLS,User-S1,AT,ZN-ZT-MT,XU,FP-FT-FM-MT	AT,ZT,FT	Needs FT-B3+
3b. MLS,User-Sn,AT,ZN-ZT-MT,XU,FP-FT-FM-MT	AT,ZT,FT	Needs FT-B3+, C2++ Sn
Compartmented-mode end-user systems:		
4a. MLS,User-M1-B1,AT,ZN-ZT-MT,XU,FP-FT-FM-MT	AT,ZT,FT	B1 too weak
4b. MLS,User-Mn-B1,AT,ZN-ZT-MT,XU,FP-FT-FM-MT	AT,Mn,FT	B1 too weak
MLS end-user systems with storage crypto:		
5a. MLS,User-Mn-B3,AT,ZN-ZT-MT,XU,FNc-FT-FM-MU	AT,Mn,FT	Needs Mn-B3+
5b. MLS,User-Mn-B3,AT,ZP-ZT-MT,XU,FNc-FT-FM-MU	AT,Mn,ZT	Needs Mn-B3+
MLS end-user systems without storage crypto:		
6. MLS,User-Mn-B3,AT,ZN-ZT-MT,XU,FP-FT-FM-MT	AT,Mn,FT	Needs FT-B3+
Single-level overall systems	**Weak links**	**Security**
Conventional single-user, single-level systems:		
7a. User-S1,AT,ZN-ZT,XU,FNc-FU	AT,S1	Pretty good
7b. User-S1,AT,ZN-ZT,XU,FP-FT-CT-IT-DU	AT,S1,FT	FT-C2 too weak
Conventional multiple-user, single-level systems:		
8a. User-Sn,AT,ZN-ZT,XU,FN-FT-CT-IT-DU	AT,Sn,ZT	Sn-C2 too weak
8b. User-Sn,AT,ZN-ZT,XU,FP-FT-CT-IT-DU	AT,Sn,ZT	Sn-C2 too weak
Low-end conventional systems:		
9a. User-S1,AU,ZP-ZU,XU,FP	Everywhere	Forget it!
9b. User-Sn,AU,ZP-ZU,XU,FP	Everywhere	Forget it!
Stand-alone end-user systems:		
10a. User-S1 stand-alone, with or without FNc	S1	Physical
10b. User-Sn stand-alone, with or without FNc	Sn	Physical

MLS systems [[1a]] and [[2a,b]] are those that are most RISSC oriented.

A well-known instance of [[1b]] is provided by the Newcastle Distributed Secure System [211] in which Trusted Network Interface Units (TNIUs) are the primary enforcers of the multilevel-security property by restricting file access accordingly. (A TNIU is also expected to be a Trust*worthy* Network Interface Unit!)

In [[1b]], each local shared end-user system does not need to enforce multilevel separation, because everything on each system is at the same level. A weakness of such architectures is that because the local systems may not even be locally secure, there is a risk of information leakage or of integrity violations between users. However, the risk of key leakage or tampering can be reduced whenever the TNIUs can completely and securely encapsulate the network crypto.

In both [[1a]] and [[1b]], the network servers are essentially trusted gateways that mediate all inbound and outbound traffic. They enable secure communication paths, but may also securely enable other services such as FTP and TELNET. (For a detailed presentation of firewall gateways, see [41] — which is in part relevant throughout this section whenever trusted network servers are included.) Network crypto can be completely encapsulated within the network servers, which must be trustworthy.

A.2 Multilevel-Secure Network Interfacing with Storage Crypto

- MLS,User-S1,AT,ZN-ZT-MT,XU,FNc-FU-FS [[2a]]
- MLS,User-S1,AT,ZP-ZT-MT,XU,FNc-FU-FS [[2b]]

Baseline architecture [[2]] includes single-level single-end-user systems and a multiplicity of file servers, each of which is a single-level server for some particular level (and compartment) — as in configurations [[1a]] and [[1b]]. However, here information is stored in an encrypted form, where cryption is provided by the client (FNc). (Cryption could alternatively be provided by the network server.)

Two basic variants within this family are considered here, with and without encrypted networking ([[2a]] and [[2b]], respectively). On one hand, the network encryption (ZN) in [[2a]] might be considered superfluous in the presence of the client-provided file cryption (FNc); on the other hand, there are many other reasons for encrypting network traffic — such as protecting keying and control information, and minimizing traffic analysis. In particular, [[2b]] is not desirable whenever the network traffic is not purely file based, although remote sessions (for example, using the Internet Protocol `telnet`) would presumably be encrypted (ZN-Nn), in which case [[2b]] would be partially transformed into [[2a]]. Configuration [[2b] is similar to [[1a]] except that it has storage crypto instead of network crypto; configuration [[2a]] has both, and may be preferable.

The network interfaces must be trusted not to violate the multilevel security constraints, that is, they must not allow information to be accessible from a lower security level or incompatible compartment. However, because information is encrypted and decrypted only by the client (for both files and authentication), the file servers and the network media need not be trusted to maintain multilevel separation or to protect keying information. Nevertheless, sharing of a key between two users is a critical operation, which implies that the network software and file system software on each end-user system must not leak keying information. Diffie-Hellman key exchange can be used to provide virtual key sharing without actually having to transmit the shared key, while avoiding having to trust the network servers to manage the shared keys.

The analysis of [192] also applies to variants [[2a]] and [[2b]]. The absence of end-user covert channels is particularly significant in the context of software crypto implementations, because leakage of keying information through a covert channel could be very serious, even with very low covert-channel bandwidth. Some defense against denial-of-service attacks can be provided by the presence of multiple file servers and multiple communication paths, especially for particularly critical levels and categories.

In both of these two variants, compromises of the local single-level end-user operating systems must not be permitted to compromise the local cryptographic keys, the authority-granting capabilities, or the authenticity-ensuring certificates. The presence of single-end-user systems helps to minimize the user-competitive aspects of local operating-system usage.

Multiple-end-user systems may be considered as another alternative within this family, although they tend to entail higher risks:

- MLS,User-Sn,AT,ZN-ZT-MT,XU,FNc-FU-FS [[2c]]
- MLS,User-Sn,AT,ZP-ZT-MT,XU,FNc-FU-FS [[2d]]

Configurations [[2c]] and [[2d]] exist with and without network encryption, respectively, similar to the corresponding pair of single-user systems [[2a]] and [[2b]]. However, in [[2c]] and]]2d]], the trustworthiness of the operating systems that are not required to enforce multilevel security then becomes a vital issue with respect to key storage, as do the competitive aspects of multiuser network interfaces, processes, and user files. These multiuser alternatives are not generally recommended with any of today's C2 systems, and are likely not to be particularly attractive in the future — unless multilevel-security separation systemwide is by itself sufficient in the absence of assured interuser (discretionary) security on each end-user system. On the other hand, because discretionary access controls are typically vulnerable to misuse by insiders and penetrators, such a situation may indeed be adequate.

A.3 Multilevel-Secure Network Interfacing with MLS File Servers

- MLS,User-S1,AT,ZN-ZT-MT,XU,FP-FT-FM-MT [[3]]

Baseline architecture family [[3]] has single-user single-level end-user systems, as in alternatives [[2a]] and [[2b]], but does not use storage crypto. However, in this case at least some of the file servers may store multilevel-secure information, and any that do so must be trustworthy in enforcing MLS separation (in contrast to the first two architecture families). As a consequence, even though the end-user systems operate as single-level systems with respect to their users, the end-user system authentication must be sufficiently trustworthy to prevent those end-user systems from masquerading as systems authorized at higher security levels. All network communications would be encrypted invisibly by the network servers.

A.4 Compartmented-Mode End-User Systems

- MLS,User-M1-B1,AT,ZN-ZT-MT,XU,FP-FT-FM-MT [[4a]]
- MLS,User-Mn-B1,AT,ZN-ZT-MT,XU,FP-FT-FM-MT [[4b]]

Today's compartmented-mode workstations minimally satisfy the TCSEC B1 criteria, but are not likely to be adequate for environments with a user community that is not completely trustworthy.

(Full MLS requirements as in B3 or A1 systems are found in [[6]].)

A.5 Multilevel-Secure End-User Systems with Storage Crypto

- MLS,User-Mn-B3,AT,ZN-ZT-MT,XU,FNc-FT-FM-MT [[5a]]
- MLS,User-Mn-B3,AT,ZP-ZT-MT,XU,FNc-FT-FM-MT [[5b]]

Family [[5]] has multilevel-secure end-user systems and client-provided storage crypto. Two cases are considered, a more conservatively designed version with network encryption [[5a]] and another without network encryption [[5b]]. Although it might seem that network cryption is superfluous in [[5a]], the situation is similar to that in [[2a]] — in which additional benefits result from the combination of FNc and ZN.

The presence of unencrypted data on a shared end-user system requires an advanced system architecture to prevent local loss of confidentiality and integrity. In particular, leakage of key information could result in one user's being able to decrypt another user's files.

A.6 Multilevel-Secure End-User Systems without Storage Crypto

- MLS,User-Mn-B3,AT,ZN-ZT-MT,XU,FP-FT-FM-MT [[6]]

Family [[6]] has network cryption but not client-provided storage cryption. (File servers could choose to store the network-encrypted form rather than decrypting it, when the destination is a file server as opposed to another end-user system.)

Again, the presence of unencrypted data on a shared end-user system requires an advanced system architecture to prevent local loss of confidentiality and integrity by direct access.

Family [[6]] is a paradigmatic architecture according to the conceptual view of multilevel-secure networked systems based on the B3 TCSEC criteria embodied in the Orange Book [151] and the Red Book [150]. However, it suffers from the wide dispersion of the need for trustworthy components. In addition, the B3/A1 criteria are incomplete in many respects, and need to be augmented. (For example, see [157] for a summary of some of the missing criteria elements.) Nevertheless, with TCSEC B3/A1 systems, secure networking, trustworthy file servers, and a trustworthy user community, this family could in principle soundly support crypto in software.

A.7 Conventional Single-User Single-Level Systems

- User-S1,AT,ZN-ZT,XU,FNc-FU [[7a]] with storage crypto
- User-S1,AT,ZN-ZT,XU,FP-FT-CT-IT-DU [[7b]] without storage crypto

The first family of conventional systems includes operating systems that permit only a single user (or group of equivalent users, as defined above) of each end-user system and that provide nontrivial networked interuser separation and file system security.

If there is never more than one authorized user (or equivalent group) for the lifetime (or less time, under certain specific circumstances) of a particular end-user system, and if the user authentication and physical security are strong enough to shut out all would-be external intruders, then the revealing of shared keys for communications and authentication can become much less of a concern, as does the revealing of nonshared keys for storage crypto. Ironically, in such a case, despite the considerable lack of security of the local end-user system, key hiding becomes less critical — although we must still recognize the problems presented by hardware maintenance.

A.8 Conventional Multiple-User Single-Level Systems

- User-Sn,AT,ZN-ZT,XU,FN-FT-CT-IT-DU [[8a]] with storage crypto (FN = FNc or FNn or FNs)
- User-Sn,AT,ZN-ZT,XU,FP-FT-CT-IT-DU [[8b]] without storage crypto

Conventional systems of family [[8]] include operating systems similar to those in [[7]], but which permit multiple users on the same end-user system and also take some care to provide nontrivial interuser separation and file system security. Multiple use may be either simultaneous (as in the case of the Sun-OS and other multiprogrammed Unix workstations) or only one at a time, as in the case of Novell's NetWare 4.0 running on top of an MS-DOS personal computer. The basic question here is whether crypto applications (storage, communications, and authentication) can be securely implemented even when the operating systems are vulnerable.

User-specific crypto keys must be protected against attack by other authorized users. Encrypting those keys with master keys is in general iteratively unsatisfactory, as noted in [158]. System-specific keys may be vulnerable to spoofing attacks by other users.

A.9 Low-End Conventional Systems

- User-S1,AU,ZP-ZU,XP-XU,FP-FU [[9a]]
- User-Sn,AU,ZP-ZU,XP-XU,FP-FU [[9b]]

To contrast further the systems already discussed, we conclude this discussion with two cases of what might be termed comic relief, the run-of-the-mill state of the art typifying (somewhat exaggeratedly) today's low-end systems. The subfamily [[9a]] includes a collection of rather primitive systems that may or may not allow single-user multiprogramming but that do not permit multiplexing of any end-user system among even nonsimultaneous users. Very little discussion is necessary, except to note that this is essentially what one gets if one's head is in the sand.

A.10 Stand-alone End-User Systems

- User-S1 stand-alone, with or without FNc [[10a]]
- User-Sn stand-alone, with or without FNc [[10b]]

No enumeration of system configurations would be complete without including a stand-alone system with no networking whatsoever (except possibly for the system's own private file server, which logically can be considered to be part of the stand-alone system). Although there is consequently no need for communication crypto, crypto is meaningful here for both storage and authentication.

A.11 Comparison of the Architecture Families

Table 15 (highlighting the various architectural families considered in Sections A.1 through A.10) summarizes what functionality must be trustworthy and gives a rough basis for comparing these families. The second column of the table summarizes the weak links for each of the families, while the third column provides some comments applicable to implementations of the given architectural family.

In each case, the authentication server must be sufficiently trustworthy, whether it is redundantly distributed or in part collocated with the end-user systems. Confidentiality and integrity are both vital. However, if the authentication server enforces the use of nonreusable tokens with encryption of suitably embedded identifiers and timestamps or nonces, many of the conventional vulnerabilities (for example, capture of passwords, or replay attacks with deficient certificates) can be avoided. (For example, see [68, 73, 74].) Typically, some of the authentication functionality would be centralized in several authentication servers, while complementary functionality would be distributed in each of the end-user systems.

In several of the family members (for example, [[1a,2a,2b,3,4a]]), each end-user system is restricted to be a single-user system, as defined above.

In three families ([[1,2,5]]), the file servers do not need to be trusted for multilevel-security separation, although in [[5]] a particular file server may be capable of handling files at multiple security levels (which is not the case in [[1]] and [[2]]). In several cases ([[1a,2,5,7a]]), the file servers do not need to be trusted for single-level file confidentiality or integrity, if the client-provided cryption is used appropriately; in other cases, the file servers must be trustworthy — with respect to single-level interuser separation in [[1b]] and multilevel-security separation in [[3,4,6]].

Client-provided cryption may be appropriate whenever the network servers and file servers are potentially less trustworthy than the end-user systems. Similarly, network encryption may be suitable whenever the network servers are sufficiently trustworthy. File-server-provided file encryption is useful mainly when the storage media themselves may be vulnerable to attack.

A primary motivation for storage crypto is to provide greater assurance that media capture or software subversions do not compromise storage security. Decisions on whether to use storage crypto are more or less independent of other architectural concerns — unless for other reasons network crypto is explicitly to be avoided. On the other hand, there are problems involved in trying to use the network-encrypted format for storage media as well, because typically some decryption must take place for headers and identifiers, or else some of the network traffic must be in the clear. In either case, traffic analysis becomes a problem, and thus it may be advantageous to have both storage crypto and network crypto.

B Tamperproofing NIDES

Appendix B considers a specific distributed-system networked environment in which user behavior is monitored by a separate analytic system that seeks to detect and analyze potentially anomalous behavior. A set of tamperproofing principles is considered, and a particular illustration of the adherence to these tamperproofing principles is given, in terms of SRI's Next-Generation IDES (NIDES) [6, 7, 98], the successor to SRI's Intrusion-Detection Expert System (IDES) [51, 99, 127].[21] The IDES/NIDES effort began in the mid 1980s.

B.1 Tamperproofing via Subsystem Encapsulation

Various requirements are considered here. To make them somewhat specific, we apply them directly to a NIDES-like situation in which an analysis system monitors the behavior of the target system(s). In this case, the target systems must not `depend on` any of the functionality in the analysis system, although the analysis system clearly `depends on` the target system(s) for its data.

- **Goal 1: Target-system integrity.** The environment of the analysis system must not be permitted to have any adverse effects on the integrity of the target systems. The normal flow of information is from the target systems to the analysis system. Any reverse information flow (including control flow, exception conditions, and indirect interactions) from the analytic system to the target system must be authenticated and validated. The analysis system should continually validate the status of all target systems to detect involuntary shutdown.

- **Goal 2: Target-system confidentiality of transmitted data.** The environment must protect any sensitive audit data on the target systems, before and during its transmission to the analysis system. This suggests that encryption should be used if the transmission path cannot otherwise be secured.

- **Goal 3: Analysis-system data security.** The target systems and the analysis system must protect the transmitted audit data from browsing, alteration, deletion, and spoofing.

- **Goal 4: Analysis-system integrity.** The analysis system must be protected from modification of its procedures and system data.

- **Goal 5: Analysis-system availability.** The analysis system must be protected from malicious denials of service.

- **Goal 6: Rulebase protection.** The analysis-system rulebase must be protected from undesired reading, unauthorized modification, and reverse engineering.

- **Goal 7: Analysis-system user access.** Access to the analysis system and its data must be restricted through user authentication within a tightly controlled set of authorized personnel.

- **Goal 8: Analysis-system user accountability.** User activities must be logged and analyzed in sufficient detail for the detection, by security officers and administrators, of malicious misuses of the analysis system and its data, and of intrusions by unauthorized personnel.

[21] IDES was perhaps an overly narrowly chosen name, because from the beginning IDES included included a statistical component as well as an expert system.

B.2 Protection Against Reverse Engineering

It may be desirable to protect sensitive knowledge about what is being monitored (for example, potential exploitation of little-known system vulnerabilities) from unauthorized reading or modification.[22] This desire for antitampering suggests the identification of more specific goals relating to those noted in Section B.1. (The relationships among those goals are indicated.)

- **Goal 9: Encapsulation and hiding of sensitive information — for example, the rulebase and the statistical measures that are used in the analysis.** Goal 9 requires the satisfaction of Goals 2 and 3.
- **Goal 10: Separation of roles among users and processes of the analysis system.** Omnipotent superuser roles must be avoided in the analysis system, just as elsewhere.
- **Goal 11: Interprocess and server authentication.** The internal components of the analysis system must be authenticated with respect to one another. (Goal 7 relates to users, whereas Goal 11 relates to subsystem components.)
- **Goal 12: Self-monitoring of accesses to the analysis system itself.** Goal 12 requires the satisfaction of Goal 8, but also requires the logging of certain actions taken by the subsystem components themselves.

B.3 Illustration of the Realization of These Goals

To illustrate these tamperproofing goals, we consider here an encapsulated environment that demonstrates the needs for additional protection, namely, an analytic system that performs anomaly and misuse detection based on inputs from audit trails, network traffic, and other sources.

In particular, we consider SRI's NIDES environment, which monitors a collection of target systems (for example, Sun Unix systems), but runs on a separate execution platform. NIDES runs on a platform that is not a target system. (We assume that target-system security is whatever it is.) Section B.4 refines the security goals specifically for NIDES.

NIDES is itself a sensitive application and has security requirements in addition to those of the systems whose use is being monitored [51].

Malicious tampering with the audit data or with the analysis system itself could lead to intrusions going undetected. If an intruder can read the NIDES rulebase, then the penetrated site and other sites using a substantially similar rulebase could be jeopardized, especially if such knowledge is shared within the intruder community.

As is any computer system, NIDES is potentially subject to attack. For example, an attacker may try to break the security officer's account, try to cause a core dump, or try to exploit potential weaknesses in the design and implementation. Once NIDES is compromised, the attacker may disable the system.

The NIDES rulebase represents explicit scenarios that are indicative of potential system misuse. An attacker who obtains the rulebase may be able to devise attack strategies to defeat the rules,

[22] This is sometimes known as the "knowledge paradox" — in which the availability of a tool for analyzing vulnerabilities itself permits the identification and knowledge of those vulnerabilities by would-be attackers. This paradox resurfaced again recently when Dan Farmer's SATAN tool was released.

and thus evade detection. Therefore, it is vital to protect the rulebase from disclosure to intruders and from unauthorized modification.

B.4 Tamperproofing NIDES via Subsystem Encapsulation

Section B.4 restates the generic goals of tamperproofing of Section B.1 in the specific context of NIDES.

- **Goal 1: Target-system integrity.** NIDES must not have adverse effects on the integrity of the target systems. NIDES should authenticate interactions with target systems. NIDES should continually validate the status of all analysis systems to detect involuntary shutdown.
- **Goal 2: Target-system data confidentiality.** It is the responsibility of the target systems to ensure the confidentiality of audit data as it is collected and in transit to NIDES. Apart from agreement on crypto keys, protocols from transmission, and performance effects on both the target systems and the analysis systems, this goal is largely independent of NIDES.
- **Goal 3: Audit data security.** NIDES must protect the audit data transmitted from the target systems to NIDES, from browsing, alteration, deletion, or spoofing.
- **Goal 4: System integrity.** NIDES must be protected from modification of its procedures and system data.
- **Goal 5: Availability.** NIDES must be protected from malicious denials of service.
- **Goal 6: Rulebase protection.** The NIDES rulebase must be protected from undesired reading and reverse engineering and from unauthorized modification.
- **Goal 7: User access.** Access to NIDES and its data must be restricted through user authentication within a tightly controlled set of authorized personnel.
- **Goal 8: User accountability.** User activities must be logged and analyzed in sufficient detail for the detection, by security officers and administrators, of malicious misuses of NIDES and its data, and of intrusions by unauthorized personnel.

B.5 Addressing the NIDES Security Goals

To address the goals stated in Section B.1, and amplified in Section B.4, NIDES must be protected at least as well as a TCSEC [151] C2 system, and better than a C2 system with respect to certain criteria elements such as authentication.

Various techniques can contribute to addressing these goals. A combination of physical, logical, and operational security controls can be used. For the proposed work, we consider only the logical controls with the highest payoff, and the least developed risk, although we assume the presence of both physical and operational controls. We concentrate here on techniques to reduce the risk of tampering with NIDES operation and techniques that make reverse engineering difficult.[23]

[23]Debra Anderson participated in the formulation of the enhancements described in Sections B.6 and B.7.

Table 16: Proposed NIDES user roles

Functionality	User Roles: Administrator	Security Officer	Experimenter
Assignment of roles to privileged users	•		
Review of system configuration logs	•		
NIDES start-up and entry of usage password	•	•	•
Configuration of alert reporting	•	•	
Configuration of alert filtering	•	•	
Configuration of target hosts	•	•	
Initiation/termination of real-time analysis	•	•	
Real-time result data review	•	•	
Audit record data review	•	•	•
Entry of encryption keys for rulebase reconfig	•		•
Configuration of NIDES real-time analysis	•		
Creation and configuration of test instances	•		•
Execution of test runs	•		•
Test result data review	•		•

Legend: • = functionality accessible to specified user role

B.6 Protecting NIDES from Tampering

- **Separation of roles.** NIDES user roles and their corresponding privileges must be made distinct by separately grouping routine security officer functions, security administrator functions, and experimentation functions. The current NIDES privileged user functionality must be enhanced to provide a role-separating facility that can be configured only by a NIDES system administrator. A particular user may be authorized to serve in several roles. For example, roles within NIDES could be divided as shown in Table 16, in which a bullet (•) indicates that the particular functionality would be accessible to the specified user role.

- **Interprocess and server authentication.** Via authentication in the IPC nameserver, we must ensure that the communication channels used by all NIDES host processes are not compromised by the insertion of bogus processes or data. The following nameserver checks would be added:

  1. Only processes known to belong to NIDES and running on the NIDES host would be allowed to register.
  2. Registration of processes with the NIDES ipc_nameserver would be verified. Processes registering with the ipc_nameserver must use an accepted password to be registered.
  3. Addresses for all known/registered processes would be verified during all ipc_nameserver transactions.
  4. Requested client/server connections would be verified for appropriateness before the connection is allowed.

- **Detection of and protection against denials of service.** To minimize the opportunities for denials of service, it would be desirable to make the following checks:

1. The NIDES Alert PopUp window can be used to deny service to other NIDES functions when the system is flooded with alerts. We might modify the window so that it will not require acknowledgment of each alert reported prior to executing any other NIDES functions. We would also enhance the window to provide a scrollable listing that summarizes recent alert activity.
2. NIDES should monitor, and report radical increases in, the volume of audit data produced by each active target host.
3. A new timeout mechanism can allow the NIDES Server processes to accept new Client process requests if the current Client request is not completed within a specific period of time.

- **Target system integrity/availability enhancements.** We recommend that the following functions be added to the NIDES target host interfaces to improve target host integrity and availability:

  1. Each `agen` program (`agen` runs on the target systems and converts the indigenous audit trails into the standard internal NIDES format) should produce periodic "heartbeat" messages to `arpool` (the program that runs on NIDES and merges all audit-trail information into a single stream) to detect outages of the link from the target system to NIDES.
  2. The security of `agen` should be increased by providing each `agen` a password, at start-up, that would be used by `arpool` to verify that the `agen` process has been properly started by NIDES and has not been tampered with.
  3. Using a closed-loop verification test, `agen` would periodically check the integrity of the audit trail data that it converts. The closed-loop test would entail `agen` generating an audit event, and then verifying that the event appears properly in the audit trail.
  4. `agen` would periodically check the Ethernet interface status to ensure that it is not allowing network packet snooping (i.e., when the Ethernet is set to promiscuous mode operation).
  5. `agen` would periodically check audit trail file modes for changes to file permissions.
  6. `agen` would periodically check audit configuration files for changes.

- **`arpool` target host authentication.** The `arpool` process would be enhanced to allow only target hosts that have been activated by NIDES and authenticated to connect. Any suspicious connection attempts would be denied and reported.

- **Smokescreen detection.** The resolver is intended to eliminate false positives and redundant alerts that might be generated by the rulebase or the statistical component. However, an intruder might deliberately flood NIDES with alerts so that his malicious deed is buried in the *smoke.* The NIDES resolver could be extended to detect and circumvent certain smokescreen activities that attempt to mask illicit activities, by noting unusually high volumes of audit data, suppressing redundant or very similar closely spaced alerts, and providing explicit cautionary messages for the security officer. When an attempted smokescreen is recognized by the resolver, an alert would be reported giving information about the suspected smokescreen.

- **Logging of NIDES user actions.** A NIDES action log would be implemented to audit all security-relevant NIDES user actions, including all attempted or successful configuration changes. This NIDES action log would be readable only by the NIDES security administrator role.

- **NIDES process integrity/security.** Processes associated with the NIDES host would be enhanced to ensure NIDES security and integrity as follows:

    1. **nagen process.** Monitoring of the NIDES hosts by NIDES itself can, for certain granularities of audit data reporting, enter a positive feedback loop where the processing of a single NIDES audit record generates multiple NIDES audit records. For this reason, the use of `agen` on a NIDES host has been strongly discouraged.

       While a manageable situation could, in principle, be achieved with an appropriate audit system configuration, the efficiency and reliability of NIDES could easily be degraded by audit configuration changes, either intentional (malicious) or unintentional (careless).

       A specialized `agen` process, `nagen`, would be developed that is designed explicitly to monitor the activity on the NIDES hosts and filter out, if present, extraneous and voluminous data that does not directly affect the integrity of NIDES. In this way, `nagen` would prevent NIDES from "flooding itself" with audit records. The `nagen` process would include these capabilities:

       - Filtering of a small set of relevant audit events from the NIDES host audit trail
       - Collection of data on network activity — in particular, attempts to access network services on the NIDES hosts from the network (e.g., `telnet`, `ftp`, `rlogin`).
       - A specialized rulebase that addresses NIDES vulnerabilities using `nagen`-provided data (for example, execution of `ptrace` on known NIDES processes or access of memory devices on the NIDES host)

    2. **Alert reporting integrity.** NIDES alert reporting can be compromised when the perpetrator of the alert is a NIDES user. In addition, since deactivation of `sendmail` on the NIDES host is recommended, a reliable mechanism for transfer of E-mail alerts is necessary.

       - A specialized reporting mechanism for alerts involving a compromise of NIDES security (e.g., cases where the perpetrator may be a NIDES user)
       - An enhanced E-mail alert reporting mechanism to route all E-mail alerts through a specific mail host

    3. **NIDES user authentication.** NIDES users would be verified through the following:

       - Enhanced NIDES start-up procedures requiring the user to enter a one-time password that would be used for NIDES-related authentications during the current NIDES execution
       - A NIDES modification allowing NIDES logins only from the NIDES host, not from any other systems
       - A requirement for users who are authorized to perform rulebase configuration functions to enter in the rulebase encryption key before performing any rulebase configuration functions in NIDES

    4. **TCP/IP services.** The integrity of the NIDES analysis host, software, and data must be maintained. Network access to the NIDES host should be completely disabled or at

least severely restricted except for communication between the `agen` processes on the individual target hosts and the `arpool` process.

Additional system administration guidelines would be provided on which TCP/IP services should be allowed/configured on the NIDES host via TCP wrappers. Guidelines for the disabling (or restriction) of network protocols and services such as `telnet`, `ftp`, `rlogin`, `sendmail` would also be provided.

NIDES would automatically review the status of TCP/IP connections (e.g., via `netstat`), analyzing changes in the status for suspicious behavior.

B.7 Protecting NIDES from Reverse Engineering

- **Protecting the rulebase from reverse engineering.** The most vulnerable area in NIDES with respect to reverse engineering is the rulebase. In NIDES, each rule is specified using a rulebase language, and is translated into machine-generated C code. These machine-generated rules are then compiled into object code that can be loaded into the operational rulebased system at run time.

  Various techniques can contribute to making it difficult to reverse engineer the NIDES rulebase:

  1. **Protection of the NIDES rulebase source code.** In NIDES installations the rulebase source code and configuration files are highly sensitive. The NIDES rule sources and their corresponding C source code should be maintained on a system separate from NIDES. A rule source encryption mechanism would be provided that would decrypt rules only during compilation. The rule-build script would be enhanced to prompt the user for the encryption key, decrypt the rule source, compile it, and then re-encrypt it.
  2. **Protection of the NIDES rulebase from unauthorized modification.** Activation/deactivation of the rules in the rulebase is safeguarded by the techniques enumerated in Section B.6, specifically use of the one-time password for each NIDES invocation and entry of the rulebase encryption key when rulebase configuration is initiated. Prevention of unauthorized users writing and installing rules into the NIDES rulebase would be addressed in two ways — the rulebase development directories would be accessible only to the NIDES system administrator and all rules would be developed using an encryption key, as mentioned earlier. Thus, any rule created with an improper encryption key cannot be decrypted by NIDES; when this occurs, the error would be reported and the rule files involved should be verified by the NIDES system administrator and probably removed.
  3. **Protection of rule object code and rulebase configuration files.** NIDES rule binaries and the rulebase configuration file, `rb_config`, would be stored in an encrypted form when they are on disk, and would be decrypted at run time as needed — if the correct key has been entered at NIDES start-up. This can be accomplished without inhibiting access to object files for rules during system execution. The rule-building script would also be enhanced to encrypt the rule object files immediately after compilation. A rule object file would be decrypted when the rule is needed by NIDES, and re-encrypted when the rule object file is no longer needed. This approach would add a slight delay to the NIDES start-up processing and rulebase reconfiguration.

- **Protecting NIDES software from reverse engineering.** The NIDES processes would be enhance in the following ways to deter reverse-engineering attempts:

  1. **Prevention of dumping.** Core dumping would be disabled systemwide. All signals that could cause a core dump and prevent the writing of the *core* file would be trapped.
  2. **Use of `ptrace` and `/dev/mem`.** In the NIDES `nagen` rulebase, rules would be included to detect reverse-engineering attempts using the `ptrace` system call or reading `/dev/mem`.

B.8 Coverage of the NIDES Security Goals

Table 17 lists the proposed enhancements and shows which of the goals given in Section B.4 are at least partially addressed by each enhancement. A solid bullet (•) indicates that the feature is primary in addressing the goal. An open bullet (○) indicates a secondary contribution.

These techniques would not make it impossible for NIDES to be compromised. However, they would greatly reduce the risk and increase the likelihood that such compromise will be detected.

B.9 Formal Methods Implications of NIDES

NIDES is a system that is normally intended to run separately from the systems whose activities it monitors, with no Internet or dial-up access, and with dedicated connectivity to the target systems that it monitors. As such, NIDES security can be considered somewhat independently of the security of the target systems — except for the possibility of compromises of the audit data on the target systems being monitored, before that data ever reaches NIDES.

From a RISSC architecture point of view, making the NIDES architecture largely separable from the target systems, and severely constraining the interface between the target systems and the data collection and analysis systems, alleviates many of the security problems commonly found in more highly accessible systems. However, numerous concerns remain as to the security of an environment such as the NIDES platform itself — as addressed here.

From a formal methods point of view, the goal is to apply formal methods where the payoff is greatest. In this case, if it were deemed important to provide high assurance for the NIDES antitampering properties noted in Section B.6, Goals 3, 4, and 7 would probably be of greatest interest, with 6 and 8 also appropriate for analysis. The formalization of these goals would very similar to typical operating-system properties, although with considerable emphasis on system integrity (which is important anyway). The security issues involved in protecting against reverse engineering also do not require any fundamentally new formalizations. The three rulebase properties and the two system properties enumerated in Section B.7 are essentially just system security properties.

Table 17: Relationship of proposed enhancements and security goals

	NIDES Requirement						
	1 Target System	2 Audit Data	3 Integrity	4 Avail.	5 Rules	6 Access	7 Users
Antitampering							
User Roles	○		○		○	●	●
IPC							
Allowed set			●			●	
Registration			●			●	
Verification			●			●	
Restrictions			●			●	
Nondenial							
Alert PopUp		○	●	●		●	
Audit data		○	●	●		●	
Server timeout			○	●			
Target Hosts							
Heartbeat	●	●	●	●			
Authentication	●	●	●				
Data verify	●	●	●	●			
Network checks		●	●				
Audit files		●	●				
Audit config		●	●				
`arpool`							
Target connect	●		●			●	
Smokescreen			●	●		●	
Logging			●		●	●	●
NIDES Host							
`nagen`							
audit data filter		●		●		●	
data collection			●			●	
special rulebase			●			●	
Alert reporting							
Insider alerts			●				
E-mail reports			●	●			
User authentication							
Session password		○	●			●	
Local execution			●	○		●	
Encryption key			●		●		
TCP/IP services		●	●	●		●	
Antireversing							
Rulebase							
Rule source integrity			●		●	●	
Unauthorized mods			○		●	●	
Object/config integrity			○		●	●	
NIDES							
No core dumps			●				
`ptrace` & `/dev/mem`			●				

Legend:
● = primary contribution to requirement fulfillment
○ = secondary contribution to requirement fulfillment

C Architectural Implications of Covert Channels

For convenience of the reader, the following article by Proctor and Neumann [192] is reprised from the 15th National Computer Security Conference, October 1992. (References are incorporated within the main body of references to this report.) That paper examines Distributed, Single-level-user-processor, Multilevel-secure (DSM) architectures, with particular attention to the elimination of end-user covert channels, and provides the basis for the Reduced Interfaces for Secure System Components (RISSC) architectures considered here. The covert-channel issues considered in the paper are actually secondary to the issues of (1) being able to readily configure multilevel-secure systems out of easily available end-user system components, and (2) being able to readily evaluate those systems as compositions and layerings of their components.

Architectural Implications of Covert Channels

Norman E. Proctor and Peter G. Neumann
Computer Science Laboratory
SRI International, Menlo Park CA 94025

Abstract This paper[24] presents an analysis of covert channels that challenges several popular assumptions and suggests fundamental changes in multilevel architectures. Many applications could benefit from a practical multilevel implementation but should not tolerate any compromise of multilevel security, not even through covert channels of low bandwidth. With the present state of the art, the applications either risk compromise or forgo the benefits of multilevel systems because multilevel systems without covert channels are grossly impractical. We believe that the presence of covert channels should no longer be taken for granted in multilevel systems.

Many covert channels are inherent in the strategies that multilevel systems use to allocate resources among their various levels. Alternative strategies would produce some sacrifice of efficiency but no inherent covert channels. Even these strategies are insufficient for general-purpose processor designs that are both practical and multilevel secure.

The implications for multilevel system architectures are far-reaching. Systems with multilevel processors seem to be inherently either impractical or unsecure. Research and development efforts directed toward developing multilevel processors for use in building multilevel systems should be redirected toward developing multilevel disk drives and multilevel network interface units for use with use only single-level processors in building multilevel distributed operating systems and multilevel distributed database management systems. We find that distributed systems are much easier to make both practical and secure than are nondistributed systems. The appropriate distributed architectures are, however, radically different from those of current prototype developments.

C.1 Introduction

This introduction describes covert channels and their exploitation. The next section gives some background on covert channel research and relevant standards. After that, we identify the circum-

[24] Copyright 1992, Norman E. Proctor and Peter G. Neumann. Presented at the 15th National Computer Security Conference, Baltimore, 13-16 October 1992, this paper is based on work performed under Contract F30602-90-C-0038 from the U.S. Air Force Rome Laboratory, Computer Systems Branch, Griffiss Air Force Base, NY 13441 [164].

stances in which covert channels need not be avoided when designing a system for an installation. First, we consider various reasons why covert channels might be tolerable in a multilevel system. Then, because covert channels are relevant primarily in multilevel systems, we consider when alternatives to multilevel systems are appropriate for an installation. This seems to leave a large class of installations that would want multilevel systems free of covert channels.

We next turn our attention to various reasons why multilevel systems have covert channels and consider how the needs of applications can be met without producing covert channels. We consider in particular how a multilevel system can allocate resources among levels without covert channels that compromise security and without inefficiencies that leave the system impractical. We describe the problems with dynamic allocation and identify three alternative strategies for secure and practical resource allocation: static allocation, delayed allocation, and manual allocation.

We describe some practical approaches to multilevel allocation for various devices, including multilevel disk drives, and explain why allocating software resources among levels is so troublesome. Finally, we present the implications for multilevel system architectures and suggest new directions for research and development.

To the Reader

Earlier versions of this paper were misinterpreted by some very knowledgeable readers, leading us to clarify the exposition. Nevertheless, we warn readers familiar with the problems of covert channels in multilevel systems that, because we are questioning some popular assumptions about covert channels, what you already know about covert channels may cause you to misunderstand our main points. Thus, please forgive our belaboring certain central issues and slighting other fascinating topics that seemed less central to the discussion.

Covert Channels

Covert channels are flaws in the multilevel security of a system.[25] A malicious user can exploit a covert channel to receive data that is classified beyond the user's clearance. Although a covert channel is a communication channel, it is generally not intended to be one and may require some sophistication to exploit. It may take considerable processing to send one bit of data through the channel; error control coding is needed to signal reliably through a noisy covert channel. Exploitation may require the help of two Trojan horses. One runs at a high level and feeds high data into the channel, and the other Trojan horse runs at a lower level and reconstructs the high data for the malicious user from the signals received through the covert channel. The low Trojan horse is not needed if the high one can send a straightforward signal that can be directly interpreted. Also, as we explain later, malicious users can exploit some special kinds of covert channels directly without using any Trojan horses at all.

A covert channel is typically a side effect of the proper functioning of software in the trusted computing base (TCB) of a multilevel system. Trojan horses are untrusted programs that malicious users have written or otherwise introduced into the system. A Trojan horse introduced at a low level can usually execute at any higher levels.[26]

A malicious user with a high clearance does not need to use covert channels to compromise high data. The mandatory access controls would permit reading the high data directly. Ordinary reading

[25] Similar flaws in other aspects of security are sometimes called covert channels, too, but a covert channel in this section is always a communication channel in violation of the intended multilevel policy of the system.

[26] If a program could run only at the level where it was installed, it would be harder for a malicious user with a low clearance to introduce the high-level Trojan horse. It would also be inconvenient to install legitimate software.

is certainly an easier way to receive the data if the discretionary access controls permit ordinary reading. If not, it is easier for a Trojan horse to copy the data into another place where the discretionary controls do permit the malicious user to read it than to exploit a covert channel to transmit the data.

The levels that concern us here are not necessarily hierarchical confidentiality levels. They may instead be partially ordered combinations of hierarchical levels with sets of compartments. We assume that a level might have some compartments. This means that two different levels may be comparable or incomparable. If comparable, one level is higher and the other is lower. If incomparable, neither level is higher or lower. A higher level denotes greater intended secrecy or confidentiality.[27]

Noise in Covert Channels

The bandwidth of a covert channel is the rate at which information or data passes through it. A noisy channel intentionally or accidentally corrupts the data signal with errors so that the information rate is slower than the data rate. A very noisy channel with an apparent bandwidth of one bit of data per second might actually leak only one millionth of a bit of usable information per second. Such a low bandwidth is beneath the notice of some. A malicious user who might have received the classified answer to a yes-or-no question almost immediately if the channel had no noise would expect to wait almost twelve days for the answer. Of course, the channel still compromises security even though extremely high noise makes for an extremely low effective bandwidth.

Noise in a covert channel may also make its information probabilistic. For example, consider a slower covert channel with a bandwidth of a thousandth of a data bit per second where each bit received has a seventy-five percent chance of being the same as what was sent and a twenty-five percent chance of being wrong. A malicious user exploiting the channel must receive the answer to a yes-or-no question many times before believing whichever answer was received more often. The expected wait for each answer is about seventeen minutes, but it takes around five hours for confidence in the answer to reach ninety-nine percent. Here again, compromise of security is postponed but not prevented.

C.2 Background

Various approaches exist for detecting and analyzing covert storage channels [61, 189] and for avoiding some of them [106]. For covert timing channels, additional approaches exist for detection, analysis, and avoidance [94, 251]. Some approaches attempt to address both types of covert channels [80]. The notions of restrictiveness and composability [137] seek to preserve the absence of covert channels under composition, assuming their absence in the underlying components.

In section C.7 we discuss some new directions for multilevel system designs that avoid all covert channels. The architectures themselves are not new, of course. Others have considered similar architectures for somewhat different reasons [188, 210].

Much of the research and development in covert channels for practical systems has been devoted to

[27] For simplicity, we assume that levels are for confidentiality although they could instead be for integrity or for both integrity and confidentiality. The levels for mandatory integrity are duals of confidentiality levels; covert channels can compromise mandatory integrity in a direct parallel to their compromise of mandatory confidentiality. For example, a Trojan horse running at a low integrity level might covertly contaminate high integrity data where overt contamination was prevented by multilevel integrity.

reducing bandwidths to what some consider to be slow rates. Sometimes delays are introduced to lower bandwidth, and sometimes noise is added to lower the usable bandwidth. These approaches merely ensure that malicious users exploiting the channels do not enjoy the same quick response times to their queries as legitimate users enjoy. The assumption may be that if it takes hours or days for an answer to a simple illicit question, malicious users will ignore the covert channel and prefer more traditional methods of compromise, such as blackmailing or bribing cleared users. Although we do recognize some situations where covert channels are tolerable, we believe the reason is rarely because of low bandwidths. For most installations, we believe that all covert channels should be completely avoided, not simply made small. A clever, malicious user can generally compromise classified information with even the narrowest covert channel.

Other research in covert channels for practical systems has addressed the elimination of some specific varieties of channels. The other varieties, typically including all timing channels, are permitted in a multilevel system because the developers could not find a way to eliminate them without rendering the system impractical for its legitimate functions. The assumption may be that any channel that is too hard for a developer to eliminate must also be too hard for a malicious user to exploit, but this assumption is so clearly fallacious that it is never explicit.

A cynical interpretation of this willingness to tolerate residual channels is that, because many users have simply accepted systems with covert channels despite the potential for security violations, developers treat a multilevel security policy as an ideal to approach, not as a requirement to meet. A more generous interpretation is that the developers intend to eliminate more and more kinds of covert channels with each new generation of multilevel designs hoping that someday they can actually design a system with no covert channels. We wish they would go straight for systems free of covert channels, and we believe the goal can be reached.

Standards

The U.S. Defense Department standards in the *Trusted Computer System Evaluation Criteria* [151], also known as the Orange Book, place restrictions on covert channels in secure systems. Systems evaluated at classes C1 and C2 would have no covert channels simply because they would always be run at a single level. There are no restrictions on covert channels in a class B1 system, even though the system would probably have plenty of them.

For a class B2 system, an attempt must be made to identify the covert storage channels, measure their bandwidths, and identify events associated with exploitation of the channels. The design must avoid all storage channels with bandwidths over one bit per second, and the audit must be able to record the exploitation events for any storage channels with bandwidths over one tenth of a bit per second. There are no restrictions on covert timing channels. In a class B3 system, the criteria for covert channels are extended to the timing channels.

In a class A1 system, the attempt to identify covert channels must use formal methods, but the criteria are otherwise the same as for class B3. The requirement of formal methods does imply that the informal methods acceptable for classes B2 and B3 may miss some covert channels. Among the channels that formal methods themselves tend to miss are the timing channels.

The criteria for covert channels in other security standards are similar to the Orange Book criteria. Although no standards require avoiding all covert channels, considerable theoretical work has been done on hypothetical systems free of covert channels. This is in part because absolute multilevel security would be better than multilevel security with potential compromise through covert channels. Another reason is surely that absolute security is far easier to express in a mathematical model than is compromised security.

We feel that the tolerance of covert channels in security standards is unnecessary and therefore inappropriate for most multilevel systems. In fairness, when the Orange Book was written, covert channels were believed to be inevitable. This belief remains widespread today. We do not accept the inevitability of covert channels in practical multilevel systems, and we fear that the current tolerance of covert channels is itself a major threat to classified information. The Orange Book and other standards are meant to promote the development of secure systems. The standards should not be used as excuses for developing systems with unnecessary flaws.

C.3 Tolerating Covert Channels

A malicious user who is cleared for certain classified data can always compromise the secrecy of the data. The problem with covert channels is that a malicious user with the help of one or more Trojan horse programs can exploit a covert channel to compromise data classified beyond the user's clearance. Installations without malicious users or without Trojan horses can tolerate whatever covert channels a multilevel system might have because the channels would not be exploited.

No Malicious Users

Of course, at any installation with more than one user, one can never be certain that no users are malicious, but a system-high installation might reasonably ignore its covert channels as if there were none. Because running system-high requires that all users be cleared for every level, the security officers of the installation would not expect users to exploit covert channels. To compromise any data in the system, a malicious user does not need a covert channel. Covert channels are tolerable in system-high installations because they do not increase the system vulnerability.

No Trojan Horses

The security officers of some installations will assume that they have no Trojan horses. They may be right only because conventional compromise remains easier than exploiting Trojan horses when malicious users have limited technical skills. Few malicious hackers have access to multilevel systems, and few multilevel systems are exposed to malicious hackers. But security officers cannot know whether their installations are among the unfortunate systems.

An installation cannot reasonably be assumed free of Trojan horses unless appropriately trained people rigorously check all the programs that run on the system to be sure that none harbor Trojan horses. All new applications and all changes to existing applications must be reviewed. Rigorous reviews are so expensive and time-consuming that the software on the system must be fairly stable. Also, the system must not have any compilers, command interpreters, or similar programs able to create code and bypass the review procedures. Because no Trojan horses are available to exploit them, most covert channels are tolerable in an installation that can afford to ensure that all software is trusted not to contain a Trojan horse. Such a multilevel installation, if any exists, is probably dedicated to one modest-size application program running on a bare processor.

Malicious users can exploit some special covert channels to compromise certain kinds of classified data without employing Trojan horses. Typically, the data might indicate how busy the system currently is at various levels. If the data were only nominally classified, its leakage would not be serious, but release of such data at lower levels could constitute a real compromise of some systems. These special covert channels are, of course, intolerable even when an installation is known to be free of Trojan horses.

Low Bandwidth

It may also be the case that leaks through covert channels are tolerable at some installations, provided the leaks are slow enough. The Orange Book suggests that covert channels with bandwidths under one bit per second are "acceptable in most application environments." This acceptability may simply be a concession to the sorry state of the art where some covert channels are sure to be present in any multilevel system and where merely identifying all the covert channels is generally infeasible.

It is difficult to believe that many security officers worry about how quickly data is compromised instead of worrying about whether it is compromised. Surely most worry about both problems. Nevertheless, a sufficiently low bandwidth could reasonably make covert channels tolerable at installations with special situations. Where all classified data is tactical data with ephemeral classifications, slow covert channels are tolerable if data would no longer be classified by the time it had been released. If leaking the answer to one crucial yes-or-no question is enough to compromise the system, either the classification of that answer must last only a split second or all covert channels must have extremely low bandwidth.

Similarly, at installations where a price tag can be placed on all classified data, some covert channels are tolerable because no Trojan horses to exploit the channels would be cost-effective or because any alternative without covert channels would be too expensive. If covert channel bandwidths are important in performing the cost-benefit analysis, some covert channels may be tolerable because of their low bandwidths. Where data is classified to protect national security, assigning prices is foolish and perhaps illegal.

Lack of Alternatives

Many installations tolerate covert channels simply because every multilevel system under consideration has some and because those in charge feel they need multilevel systems. Fortunately, these difficulties can be overcome. We believe that there can be multilevel systems without covert channels and that there are often suitable alternatives to multilevel systems. The accreditors of automated systems for multilevel applications should not have to tolerate covert channels.

Alternatives to Multilevel Systems

Not all applications have to run on multilevel systems. We mention first two unattractive options that must sometimes be taken. One is not to implement the application at all, and the other is to implement it with manual procedures only. The remaining alternatives are all automated implementations. The potential benefits of automation include convenience, accuracy, speed, and lower costs. These benefits have permitted the implementation of many applications that were infeasible before the advent of computers.

When an application involves only one level of data or when all users are cleared for every level of data, the best alternatives are a single-level system or a system-high system, respectively. But the applications that interest us here have some data classified at levels beyond the clearances of some users of the automated system. A single-level or system-high system cannot accommodate these applications, but a multilevel system is not the only alternative left. Another possibility is a system with an independent subsystem per level (ISPL). ISPL systems tend to be inefficient, but at least they are intrinsically free of covert channels. We present the ISPL architecture mostly because it is useful later for comparisons with more attractive alternatives.

In an ISPL system, there is a separate subsystem for any level where the system as a whole could have some data. Data is stored on the subsystem for the level matching the classification of the data. Additional upgraded copies of the data might be stored on some other subsystems at higher levels. A user has access to a subsystem only if its level is dominated by the user's clearance.

The subsystems are electronically independent. Each subsystem has its own hardware, and the hardware for the subsystem at one level is not connected to any hardware for subsystems at other levels. The subsystems are not completely independent, however. They are parts of a whole system with multiple levels because users sometimes refer to data on a lower subsystem in order to modify data on a higher subsystem. Users might also manually reenter data from a low subsystem into a high one, or operators might transfer data storage media to higher subsystems.

Like single-level systems, ISPL systems are inherently free of covert channels. Multilevel security is compromised only when people fail to follow proper procedures. The automated parts of the system cannot themselves reveal data to a user not cleared for it. However, trying to overcome some of the limitations of an ISPL system may lead to complex procedures, and the complexity brings serious dangers that accidental compromise would become frequent and that malicious compromise would become easy to arrange.

Because the subsystems are independent of each other, none of the coordination among subsystems can be automated. This tends to diminish all the potential benefits of automation. Unless the required coordination among subsystems is minor, an ISPL system may well be too inconvenient, inaccurate, slow, or expensive for an application. An integrated multilevel system may then be the only practical option. Unfortunately, multilevel systems typically have many covert channels.

Some Reasons for Covert Channels

Our aim is to avoid all covert channels in multilevel systems. Present experience, however, is that any practical multilevel system contains many covert channels, despite the attempts of developers to eliminate them. It has been so difficult to avoid covert channels because several highly desirable functions of a multilevel system seem to produce covert channels as a side effect. Fortunately, the essential multilevel functions can be implemented without building covert channels into the system.

The differences in functional capabilities between ISPL systems and multilevel systems highlight the major sources of covert channels in multilevel systems. In an ISPL system, which cannot have covert channels, the absence of connections among the independent subsystems for each level prevents the system from doing all that a multilevel system can do. Among the services requiring some manual assistance in an ISPL system are reading consistent data from lower levels, downgrading overclassified data, writing up reliably, and maintaining consistency among the values of data items at different levels. A multilevel system needs no manual assistance with these services, but the implementation techniques generally introduce covert channels.

Reading Down

An automated system might allow one process to change data that another process is currently reading. Then, the value the reading process receives could reflect neither the value before the change nor the value after the change, but some useless mixture of the two values. Such mixed results from reading are unacceptable in most applications. The usual technique to prevent the problem is for the reading process to lock the data before reading it. The lock is not granted if any other process is currently writing the data, but once the lock is granted, no other process is permitted to write the data until the reading process releases the lock.

In a multilevel system with support for reading down, this technique produces a covert channel. Lower-level processes can detect when a higher process reads down to lower data because the higher process holds a lock that prevents the lower processes from writing the lower-level data. Data cannot be locked for reading down without producing a covert channel.

Different techniques free of covert channels can ensure that high processes do not read inconsistent data [59, 107, 130]. The most popular technique is for the high process to check whether any lower process may have written the data between the time when the high process started to read the data and the time when it stopped reading the data. If so, the read is potentially inconsistent, and the high process repeats the entire read again until it is sure that no lower process wrote the data while it was being read. For some applications, there is a serious risk with this technique that a high process that tries to read a lengthy and volatile data item may keep trying to read the item for a long time without ever succeeding. Other techniques may be appropriate for those applications.

Downgrading

All downgrading is inherently an exploitation of a covert channel. When the downgrading is legitimate, one could say that the channel is not really "covert," but the intended downgrade of overclassified data is often accompanied by some incidental and unacknowledged downgrading of other data. A Trojan horse might exploit the channel by manipulating the other data. It may also be possible for a Trojan horse to hide other data in the overclassified data. Multilevel system designs cannot provide legitimate automated downgrading and still avoid all covert channels.

Writing Up

When a user working at a low level upgrades low data to a higher level, the data is said to be written up.[28] To make the writing reliable, the low user might be notified whether sufficient resources at the higher level are currently available to support the writing up. This notification produces an exploitable covert channel. Suppressing the notification makes writing up unreliable; the user or program that wants to upgrade data never knows whether the writing up worked or not. Applications that need writing up typically need reliable writing up, not hit-or-miss writing up. Reliable writing up can be achieved without covert channels by reserving sufficient resources at higher levels to accommodate all potential requests to write up. This is not easy to implement, and reserving the high resources may constitute a serious loss of efficiency. A practical multilevel system apparently cannot provide reliable writing up without covert channels.

Consistency Across Levels

When an application requires consistent values in two data items, a change to one may force a change to the other to keep them consistent, or alternatively, a change to one may be forbidden until after the other is changed to a consistent value. This can be problematic in a multilevel system when the two data items are classified at different levels [59]. If the levels are comparable, one approach is secure and the other produces a covert channel. Which is which depends on whether the data item changed is at the lower or higher level. Neither approach is secure if the levels are incomparable due to differing compartment sets.

Fortunately, one result of a rational classification of data is that any criterion of consistency applies to data items that are all at the same level. A data item would never have to be consistent with data items at any other levels. A requirement for consistency with a higher item implies that a user cleared to read the lower item can infer something about the higher item, which must have

[28] If the user were working at the higher level, the upgrade is from reading down, not writing up.

a consistent value. The existence of the inference suggests either that the lower data should be classified at the higher level or that the higher data should be classified at the lower level. If data were classified rationally, users cleared just for lower data could not infer anything about higher data.

In practice, however, classification is not purely rational, and some applications really may need consistency across levels. This can be achieved without covert channels, provided that reliable writing up is properly implemented and the levels involved are all comparable. The likely cost is gross inefficiency from keeping the writing up reliable and some inconvenience because users must always change the lowest items first. Data consistency across levels, freedom from covert channels, and practicality seem to be incompatible in a multilevel system.

Resource Allocation among Levels

We turn next to another distinction between ISPL systems and multilevel systems, their different abilities to allocate resources among levels. In an ISPL system, the allocation for a level is the hardware in the subsystem for the level. In order to change the allocation for a level, some piece of equipment must be replaced, and reallocating resources from one level to another is likely to involve bringing down two subsystems for a while. In a multilevel system, reallocating resources is more convenient. Resources can often be allocated to whichever level can make the best use of them at the time. This can greatly increase the efficiency of the system. With a multilevel system instead of an ISPL system, the users can get more service from the same hardware or equivalent service from less hardware.

Reading down, downgrading, writing up, and data consistency across levels, as we explained before, are not just functional distinctions between ISPL systems and multilevel systems, but also common reasons for covert channels in multilevel systems. Similarly, resource allocation is a common reason why multilevel systems have covert channels, as well as being a functional difference from ISPL systems.

Because a system often has many kinds of resources, resource allocation may be the reason for most of the covert channels in a multilevel system. Among the space resources to be allocated are physical memory, entries in operating system tables software, storage on disk, and bandwidth in a network connection. The allocable time resources include processor time (CPU time), service time from the operating system, disk access time, and access time to other multilevel devices such as terminals, printers, tape drives, and network interface units. Resource allocation is a primary function of operating systems, but multilevel networks, database management systems, and even applications have resources of their own to allocate among levels.

We consider four general strategies for resource allocation among levels: static allocation, dynamic allocation, delayed allocation, and manual allocation. Dynamic allocation is the most efficient but inherently produces covert channels. The other three strategies are free of covert channels but can be inefficient to the point of complete impracticality when used for the wrong resources. Static allocation is the simplest strategy and the least efficient. It is usually as inefficient as an ISPL system. Delayed allocation and manual allocation are more efficient, sometimes approaching the efficiency of dynamic allocation. Delayed allocation is better suited to some resources, manual allocation is better for other resources, and a combination of both may be better than either one in some cases. We use the allocation of processor time as the main example to illustrate the four strategies.

Static Allocation

With static allocation, a fixed portion of a resource is allocated to each level that shares the resource. One level cannot borrow from another level even when the first level could use more than its share and the other level has idle capacity.

If processor time is statically allocated, the share of time allocated to a level is generally determined through the initial system configuration. The configuration might assign time slots to each level. The schedule would consist of a sequence of time slots that is repeated for as long as the processor runs. The share for a level is the length of its time slot in the sequence or, if the level has several slots, their combined length. Only processes at the proper level run during the time slot for a level. The level gives up the processor at the end of its time slot even if some processes at that level still want processing time. On the other hand, during the time slot for a level, the processor is left idle whenever every process at the level is waiting for I/O or whenever there are no current processes at the level. This means that the processor may be idle during the time slot for one level when there are processes at another level that could have been serviced.

Dynamic Allocation

At the cost of producing a covert channel, dynamic allocation avoids such wasting of resources. Resources are allocated among levels based on the current needs at each level. The simplest algorithms allow one level to borrow freely as needed from other levels. More complicated dynamic allocation algorithms place some limits on how much can be shared or how frequently reallocation can occur.

If processor time is dynamically allocated, the current loads might freely determine the share of processor time for a level, or the system may adjust shares within configured limits. When the higher levels are busy, processes at lower levels cannot get as much processing time as when the higher levels are idle. Because lower processes can detect whether higher levels are relatively idle or relatively busy, there is an exploitable covert channel.

For example, a high Trojan horse could send a “one” bit during a particular period by requesting so much processor time that the processor would seem especially busy to the low Trojan horse receiving the signal. To send a “zero” bit instead, the high Trojan horse would refrain from requesting processor time so that the low Trojan horse would find the processor relatively idle. Irregular patterns of legitimate activity probably make the channel noisy, and the noise reduces the effective bandwidth of the channel. But the channel is not eliminated. Some bandwidth would still be available for leaking information to users who are not cleared to see it.

The covert channel from dynamic allocation is exploited by exhausting the resource. Processor time like any resource is finite, but in some cases, processor time is effectively inexhaustible. If the heaviest possible load on the processor would not consume all the available time, there is always time available whenever a level wants some. This eliminates the covert channel, but it makes dynamic and static allocation equally inefficient. Ensuring that processing time is always available with dynamic allocation would ensure that time is always available with static allocation, too. The same amount of processing time would go idle either way.[29]

[29] In some circumstances, dynamic allocation might always give enough time even though static allocation of the same total capacity did not always give enough. This may occur if the limits on the load yield a maximum combined load for all levels that is less than the sum of the maximum loads for individual levels. The most likely reason for such a pattern of loads is that some other dynamically allocated resources are exhausted. The allocation routines for the other resources would then have exploitable covert channels even though the allocation routine for processor time did not.

Delayed Allocation

Allocating resources to one level may entail denying the same resources to other levels that request them later. A dynamic allocation strategy that could support instant reclamation of resources need not have a covert channel. Each level would have a basic allocation, but when a lower level was not using all of its basic allocation, a higher level wanting more than its own allocation could borrow from the unused portion of the lower level allocation. If the lower level later became busy enough to want some of the borrowed allocation back, enough would be instantly reclaimed for the lower level.

Similarly, if an intermediate level wanted more than its allocation, it could also borrow from the lower level. When a higher level had already borrowed from the lower level, that would not influence how much the intermediate level could borrow. If necessary, resources that were borrowed for the higher level would be instantly reclaimed and reallocated to the intermediate level.

A higher level could not borrow resources from a lower level while the lower level was using them or while any intermediate level was already borrowing them. Also, a lower level could never borrow from a higher level although it would sometimes reclaim its own basic allocation from the higher level or usurp the resources of a still lower level that the higher level happened to be borrowing.[30]

When a process at a level is given resources, it might be told whether they come from the basic allocation for its own level, and if not, it could be told from which lower level it is borrowing them. It must not be informed whether the resources were reclaimed from a higher level. There is no covert channel because the borrowings of higher levels do not affect the resource amounts available for a lower level.

When requests for resources are satisfied, the resources are allocated with the same speed whether the resources are currently free or currently being borrowed at a higher level. If free resources might be allocated instantaneously, then borrowed resources must be reallocable to a lower level instantaneously, too. Because instantaneous reallocation is not feasible for most resources, instantaneous allocation of free resources usually cannot be provided either. If borrowed resources can be reallocated only slowly, free resources must be allocated just as slowly. The delayed allocation strategy is named for the sometimes substantial delays the strategy can introduce in the allocation of resources.

For a delayed allocation of processor time in a system with only comparable levels, throughput could be maximized by making a basic allocation of all the processor time to the lowest level. Each level would seem to have available to it all the time that lower levels were not already using. At the end of each time slice, the processor would be allocated to the lowest level with a process ready to run.[31] An interrupt for the currently allocated level could be serviced promptly, but an interrupt

[30]When a system involves incomparable levels, the rules for borrowing are more complex. Incomparable levels cannot borrow from each other, nor can they compete to borrow from another level lower than them. One way to avoid competition among incomparable levels is to allow only some of the higher levels to borrow from a lower level. The system configuration would select which higher levels can borrow from a level. The levels selected to have borrowing privileges for a resource at a lower level must be mutually comparable. For any two incomparable levels, the selections for a lower resource might contain one or the other of the two incomparable levels, or perhaps neither, but certainly not both. Because any level not selected could not borrow the lower resource at all, it would never compete for the resource with another incomparable level that was selected.

[31]All levels except the lowest level are borrowing their time from the basic allocation to the lowest level. Because two incomparable levels cannot compete for the same resource, a system with incomparable levels needs some changes to the algorithm. The simplest variation is to specify a repeating sequence of time slices. The slices in the sequence need not all be the same length of time, but for each cycle through the time slices, each slice must be the same length

for another level would not be serviced until the next time slice when no lower tasks were pending. With all time slices being of equal duration, this delay in servicing interrupts conceals whether the processor was idle when the interrupt occurred or was busy servicing a higher level. The delay clearly wastes some processor time in order to avoid the covert channel found with dynamic allocation.

Because a lower level would not be prevented from consuming all the time and shutting out all higher levels, some installations may prefer instead to give each level a basic allocation in order to guarantee some time for each level. This fairness comes at the cost of lower overall efficiency. Whenever multiple levels compete for a shared resource, any strategy to prevent denial of service to high levels will either require more resources or produce a covert channel, entailing compromise of multilevel security.

The advantage of dynamic allocation is its more efficient use of processor time than with static allocation. In fortunate circumstances, delayed allocation is essentially as efficient as dynamic allocation. But in ordinary circumstances, the delays introduced to conceal processor loads at higher levels make delayed allocation less efficient than dynamic allocation. And in unfortunate circumstances, delayed allocation could be even less efficient than static allocation.

Manual Allocation

A contributing factor in producing a covert channel with dynamic allocation is that the allocation is changed automatically based on data from untrusted software. Changes in the allocation based on trustworthy data do not necessarily produce a covert channel. The operators of a multilevel system could sometimes switch the system manually among a variety of different multilevel allocations appropriate for different situations. The operators would choose an allocation based on their expectations of the upcoming resource needs at each level. They must be careful to use information from outside the system, not simply the current loads at each level. Those loads may reflect the influence of Trojan horses instead of legitimate activity.

More automated variants of manual allocation are also possible. Some information within the system could be used for automatic changes in the allocations of resources among levels. The information that is safe to use is information that users or operators input manually and that comes through trusted paths to ensure freedom from the influence of any untrusted software. On a multilevel system, safe inputs may include user logins, user logouts, user requests to change to a new level, and possibly some other inputs through an operator console.

These inputs must follow trusted paths from the user or operator to the TCB. There is no covert channel because Trojan horses are incapable of spoofing what a user does through a trusted path. That is precisely what makes a path qualify as a trusted path. Because Trojan horses cannot produce any of the manual inputs that determine how allocations are updated in the manual allocation strategy, they cannot influence the changes in allocation to any level. It is crucial that the only information used to adjust the allocations is information the operating system receives directly from users through trusted paths.

Manual allocation of processor time can be reasonably efficient in a multilevel system used primarily for online processing. If the user inputs for logging in, logging out, and changing level all come via a trusted path, the allocation of processor time for a level can be proportional to the number of

as it was in the first cycle. All the time slices would still be in the basic allocation for the lowest level, but different sets of borrowing levels should be selected for different time slices in the sequence to ensure that each incomparable level has chances to borrow processor time.

user sessions currently logged in at a particular level. This is often a fair measure of the expected load at that level. No time would go to levels with no current user sessions. When all current sessions are at one level, that level would be allocated all the processor time. Allocations would be subject to change each time a user logged in or out or changed from one level to another.

The ratio of the number of current user sessions at a level to the total number of current sessions is a secure basis for manual allocation only on a system where the total number of users logged in is unclassified. If users with low clearances must not know how many users are logged in at higher levels, then the ratio determining the allocation for a level should instead compare the current sessions at the level to the sessions at or below the level. Manual allocation based on this ratio would be somewhat less efficient.

Efficiency might be enhanced by taking into account some other information about current user sessions that the trusted paths have validated. The user's name, the time of day, and, if the system is distributed, the processor supporting the user session could be used to anticipate different loads from different sessions and calculate allocations based on those expectations. The weights for the calculations should come from tables the operators have prepared in advance, not from the current demands of the sessions.

In a multilevel system where online processing predominates but there is some background or batch processing, this approach should be modified so that some time is allocated to levels that may have offline processing. Otherwise, offline processing at a level would cease whenever there happened to be no current user sessions at the level.

Reallocation based solely on manual inputs would not be as efficient as dynamic allocation based on all available information. It should still be more efficient than a static allocation that never changes. Manual allocation, like delayed allocation, is less efficient than dynamic allocation. Both allocation strategies are compromises between dynamic allocation and static allocation.

Manual and delayed allocation can be combined. The same kinds of inputs as the manual strategy uses to update allocations can be used to update the basic allocations for the delayed strategy. The hybrid allocation strategy improves the efficiency of delayed allocation, and with resources for which delayed allocation is appropriate, the hybrid strategy is more efficient than manual allocation, too. The hybrid strategy cannot outperform the best dynamic allocation algorithm, nor is it likely even to be equally efficient. However, the covert channels of dynamic allocation are absent from a combination of manual and delayed allocation, just as they are with static allocation, simple delayed allocation, and simple manual allocation.

C.4 Allocating Device Resources

We call a device multilevel if it ever stores or transmits data for more than one level. At one extreme, the device may always handle hundreds of levels, or at the other extreme, it may handle one level on some days and another level on the other days.

As a first example of a multilevel device, we consider a multilevel terminal. It is inconvenient for a user to move to a different terminal in order to work at a different level or for the user to have as many terminals on one desk as there are levels of work to do. With one multilevel terminal, terminal access time could be allocated to whichever level the user currently wants. Multilevel terminals would cost more than single-level terminals, but the convenience may justify the added cost. And if one multilevel terminal fully replaces several other terminals, there may even be a cost savings.

The multilevel terminal would need some special manual inputs for selecting the level where the user wants to allocate the terminal access time. A reset button, a dial or switch for indicating a level, and a ready button would be enough. When the user presses the reset button, the terminal clears its screen and any volatile memory, locks the keyboard, and unlocks the level dial. Then, the user can set the dial to the new level. When the user presses the ready button, the terminal locks the dial, selects the single-level communication line at the level corresponding to the setting of the dial, and unlocks the keyboard.

When the terminal is installed, the security administrators should make sure that the dial settings correctly label the processors that can be accessed through the corresponding single-level lines. The terminal must also be protected from sabotage, of course. We caution against making the multilevel terminal too sophisticated. A multilevel workstation is far less likely to be implemented free of covert channels than is a basic multilevel terminal. Pushing the reset button must remove all traces of whatever had been done before.

A similar approach would work for a multilevel printer or multilevel tape drive. The reset button of a printer must clear all physical traces of what was printed at the previous level. The justification for a multilevel printer or tape drive is probably lower cost or greater convenience again.

Trusted Network Interfaces

A network of multilevel lines is more convenient for operators to install and maintain than are separate networks of single-level lines for a variety of levels. The convenience may justify the cost of the trusted network interface (TNI) units to connect each single-level communication line to a multilevel line. Especially in a wide-area network, the savings from having fewer cables may also offset the cost of TNI units.

If a multilevel line is a radio-frequency cable, each level can be statically allocated its own frequency band. A TNI unit would tune to a band based on its control settings. Whoever installs or maintains a unit connecting a multilevel line to a single-level line must check that the control settings of the unit agree with the level of the single-level line.

TNI units should be connected to the communication lines of single-level processors and devices so that they can communicate over the multilevel network lines. Rather than having TNI units connected to the various single-level lines for a multilevel device such as the terminal described earlier, one TNI unit could be embedded in the multilevel device so that one multilevel line could replace all its single-level lines. The terminal would retune its frequency based on the current dial setting when the user pushed the ready button. Embedding a TNI unit is also an option for a multilevel printer or multilevel tape drive.

A network of multilevel lines with TNI units wherever processors and devices connect to the network is functionally equivalent to separate single-level networks. A single-level processor could communicate with other single-level processors and devices only if they are at the same level. It could communicate with the multilevel devices we described only when they were currently allocated to the same level, too.

More complex TNI units might support multiple single-level lines or support an allocation strategy for the multilevel lines more efficient than static allocation of frequency bands to levels. We suspect the added efficiency would not offset the problems of the extra complexity: a higher cost per unit and reduced assurance of multilevel security.

Cryptographic methods can supplement such TNI units but are never a substitute. If network lines are vulnerable, encryption can help preserve the confidentiality and integrity of messages transmitted over the network. However, if the network does not carefully allocate resources based on the levels of the decrypted messages, there are covert channels. Users communicating at low levels could detect heavier and lighter loads on the network from activity at higher levels, possibly due to Trojan horses. Encrypting messages does nothing to eliminate this covert channel.

Multilevel Disk Drives

Any multilevel application requires some support for reading down. Reading down can be implemented with multilevel processors, multilevel disk drives, some other multilevel storage media, or a combination. Disks are more generally useful for reading down than are other storage devices. Also, we believe that multilevel disk drives are much easier to build free of covert channels than are multilevel processors. We are not certain that multilevel drives really can be implemented without covert channels as nobody has yet tried, but we sketch a design that seems feasible.

The design uses manual allocation of the storage space on the disk and uses a combination of delayed and manual allocation for the access time to the disk drive. The interface for the operator has a reset button, a restore button, an accept button, and various browsing buttons to control a display panel. The interface to the rest of the multilevel system is through separate single-level lines for each level the drive supports.[32]

A special single-level line connects the disk drive to a single-level processor with a configuration table that the operator maintains. The table shows (1) the levels of the other single-level lines, (2) which levels are higher or lower than other levels,[33] (3) what level of data is to be stored in each sector of the disk, (4) how long each period in the access time schedule lasts, (5) which level is the basic level for each time period in the schedule, (6) which higher levels may borrow time during each time period,[34] and (7) what position the disk arm is to be in at the end of each time period.

When a configuration table takes effect, the allocation of storage space to a level is the sectors that the configuration assigns to that level. The allocation strategy for access time is a hybrid of delayed and manual allocation. The effective configuration gives the parameters for delayed allocation. The basic allocation of access time to a level is the time periods where that level is the basic level.

While the disk drive is providing its regular reading and writing services, the drive rejects any requests to change its internal configuration table. When the operator pushes the reset button, the disk drive locks all the buttons, stops regular reading and writing services, and waits to receive a new configuration table through its special line. The operator working on the processor where configuration tables are maintained should request a change to the new configuration. If the disk drive finds the new configuration unacceptable, it shows an error code in its display panel and unlocks the reset and restore buttons. The operator has a choice of fixing and resubmitting the new configuration or restoring the old configuration.

If the drive would accept the new configuration, it unlocks all buttons and prompts the operator to double check the changes. The operator uses the browsing buttons to check all parts of the new configuration and perhaps also the old configuration to be sure that the configuration the disk drive received is exactly as intended. This precaution means that the single-level processor where

[32] As before, the single-level lines could be replaced with an embedded TNI unit and a multilevel line.

[33] The level of the special line should be lower than the levels of the other lines.

[34] If the disk supports some incomparable levels, the borrowing levels for a time period must be chosen to be mutually comparable.

the table is maintained and the path connecting the processor and disk drive do not have to be completely trusted.

If the configuration does not look right, the operator pushes the restore button. The disk drive locks the restore and accept buttons, discards the new configuration, and resumes regular service with the old configuration. If the operator pushes the accept button instead, the restore and accept buttons are still locked, but it is the old configuration that is discarded and the new configuration that is used to resume regular services. Also, before resuming regular reading and writing services with a new configuration, the drive clears any disk sectors then allocated to levels lower than before.[35] During regular services, the reset and browsing buttons remain unlocked.

While the disk drive serves a level, it accepts inputs and returns outputs through the communication line for the level. The other communication lines are ignored. The drive honors any requests to read or write sectors at the current level. To support reading down, the drive also honors requests to read sectors at lower levels.

Within the disk drive itself, there is a scheduler that determines which level to serve and for how long. The scheduler cycles through the schedule of time periods in the current configuration. At the beginning of a time period, it serves the basic level for the period. When appropriate, the scheduler may change level before the period ends and allocate whatever remains of the period to the lowest level that can borrow time in the period. It may also change level more times and allocate the remainder of the period to the next highest borrowing level.[36] If the highest borrowing level for the period is reached, the level stays the same until the start of the next period – when it becomes the basic level for that period.

The scheduler in the disk drive changes to the next highest borrowing level when the current level has no more disk accesses to make. If the current level is already the highest borrowing level, the drive waits idly until the period ends or more requests are received at the highest level. The drive does not change level if there would not be enough time to establish the new higher level and still position the disk arm as the configuration requires before the period ends. Similarly, as the period draws to its end, the disk drive rejects any access request that could not be completed in time to position the disk arm properly afterward.

The covert channel that would be produced by a dynamic allocation of access time is not found in this design. The allocations of storage space on the disk and the parameters used for delayed allocation of access time change only when the configuration changes, and that is only when the operator pushes the appropriate buttons. While the configuration remains unchanged, the performance of a disk drive in one time period has no effect on its performance in later time periods. Within a time period, the service to a level depends just on the requests from that level and lower levels. The higher borrowing levels receive no service until the lower levels voluntarily release their claims on the time period.

The sometimes long delays while a multilevel disk drive is inaccessible from a level make the drive inappropriate for the I/O of many ordinary processes. We suggest that most data be kept on single-level disks and accessed there primarily. Multilevel disks would hold only replicas of data

[35] Any sector allocated to a level incomparable to its old level is also cleared. If the level of a sector is left unchanged, its contents are kept. The contents are also kept in a sector whose level increases. In such a sector, the contents are effectively upgraded to the higher level.

[36] Because levels that may borrow time within a period are chosen to be mutually comparable even when the drive supports incomparable levels, the next highest borrowing level is uniquely defined until the highest borrowing level is reached.

that is sometimes read down. The following scenario explains how this might work.

A Scenario with Upgraded Replicas

An ordinary process running on a single-level processor at some low level writes to a file stored on a single-level disk at the low level. When the process releases its write lock, a new value of the file is available for other processes at the low level to read from the same disk. But if the file header indicates the file is replicated, the replicas do not yet have the new value.

A replica management (RM) process on the same processor sends the updates to RM processes for any other disks that the file header indicates keep replicas at the low level. Although some of these RM processes may run on other processors, all run on single-level processors at the low level. The RM processes update the replicas on their disks to reflect the new value of the file. Multiple copies at the low level increases the availability of the file to users throughout the system. If its disk is multilevel, an RM process also records the new time stamp of the updated replica in a special disk segment for the low level.

Periodically, each process of another kind, the upgraded replica management (URM) processes, reads down on a multilevel disk in the time stamp segments for any levels lower than the level of the processor where the URM process runs. For each file with an upgraded replica at the high level of its processor, the URM process checks whether the time stamp of the lower replica has changed since last checked. If so, the URM process reads the updated lower replica of the file. It is again reading down on the multilevel disk.

The URM process sends the updates to the appropriate RM processes at the high level. As before, the RM processes write the new value of the file into the replicas on their disks at the high level. If any of these disks are multilevel, that may trigger another round of propagating the updates to replicas at still higher levels.

The new value of the file becomes available to ordinary processes running on single-level processors at a variety of levels. A process running on a processor at one of those levels can read any replica of the file found on a single-level disk at the same level.[37]

In the scenario above, all processes can run on single-level processors. Ordinary processes can do all their reading and writing on single-level disks. The only processes that must access multilevel disks are the replica management (RM) and updated replica management (URM) processes. An RM process reads and writes time stamp segments and replicas at its own level, and the URM processes read down to lower time stamp segments and lower replicas.[38]

The inefficiencies of the allocation strategy for access time to the multilevel disk drives may hinder the upgrading of new or changed files. To update the upgraded replicas at the same time as the changes are made in the file itself would require reliable writing up, not just reading down. Because a covert-channel-free system is not expected to have reliable writing up, there will be some lag between the writing of a file and the updating of the upgraded replicas. The choice of an allocation strategy for the multilevel disk drives would affect only how long that lag can be. It does not affect any other processing. In particular, the I/O of ordinary processes and the propagation of replicas within a level are unaffected. They can benefit from all the efficiencies of high-performance, single-level disks.

[37] If the single-level disk is remote from the process, processes on other processors at the same level would help with the reading.

[38] A disk controller process on the same processor as the RM or URM process might mediate its reading and writing of the multilevel disk.

C.5 Allocating Software Resources

While discussing multilevel devices, we have ignored multilevel processors and assumed that the multilevel devices would have to communicate with single-level processors. We now consider some of the resources of a multilevel processor. A multilevel processor has a trusted computing base (TCB), typically consisting of a kernel and some trusted processes. The software for the kernel and most trusted processes runs multilevel. The resources of that software are allocated among the various levels that the software serves.

As with hardware resources, dynamically allocating these resources on the basis of current demand creates an exploitable covert channel. Because the resources are limited, a low process employing the services of the multilevel software can detect how much has been allocated to higher levels, and a high process can send signals by modulating its demands on the multilevel software services. Static, delayed, or manual allocation, on the other hand, would produce no covert channels. Static allocation is feasible for most TCB software resources but is relatively inefficient. Manual allocation is often feasible and more efficient. Delayed allocation is also more efficient but would be too difficult to implement correctly for many software resources.

Kernel Resources

The innermost layers of a trusted operating system for a multilevel processor are called a trusted executive or kernel. The layers that concern us include the layer presenting the abstraction of processes and all lower layers. These are the layers that do not run as processes. The kernel is inherently multilevel, and many of its resources are also multilevel. The execution time of the kernel is allocated among the levels. An allocation of processor time to a level includes the time the kernel spends serving that level, not just the execution time of single-level processes at the level. The storage resources of the multilevel kernel in a multilevel processor include most of the system data space. At any given moment, some of these resources would be fully allocated to the same level as is the processor time. Other storage resources might be partially allocated among levels.

It is extremely difficult to avoid every covert channel in the allocation of kernel time and storage in a multilevel processor. Some kernel resources can easily be allocated among levels using a static or manual allocation strategy, but it is unlikely that all resources of a practical multilevel kernel would be so safely allocated, especially in the lowest layers of the kernel.

A multilevel processor embedded in a special-purpose device such as a disk drive, printer, terminal, or network interface unit should need such a simple executive that safe allocation of all resources can be achieved without sacrificing practicality. The executive probably would not even support real processes.

A more general-purpose multilevel processor supporting user processes, however, seems doomed to have some covert channels at least within its kernel. The service time and data spaces for the lowest kernel layers could not avoid load-influenced dynamic allocation. The covert channels might all have small bandwidths or high noise, but they would still be there for malicious users to exploit, however slowly. Even some special-purpose multilevel processors, such as file servers, may be too sophisticated to be reliably free of covert channels.

To date, no designers have even come close to producing a covert-channel-free kernel for a multilevel operating system. In a typical design for a multilevel kernel, many low-bandwidth covert channels are not even identified.

Trusted Process Resources

Secure allocation among levels is somewhat easier for the resources of multilevel trusted processes than for kernel resources. This may be largely irrelevant, however, because multilevel processes exist only on multilevel processors with more sophisticated kernels. Because the kernels already would have introduced some covert channels, the effort to avoid all covert channels in the trusted processes may be futile. The result would still be a TCB with some covert channels.

As with the kernel, the allocation of the execution time of a trusted process to a level must be considered part of the allocation of processor time to the level. Static allocation of trusted process time is simpler, but the efficiencies of manual allocation might justify the extra complexity.

The virtual address space of a trusted process in a multilevel processor gives it storage resources that can be allocated among the levels that the process serves. Some variables in the address space would be fully allocated at any moment to the same level as the process time. Other storage resources, especially structures such as tables, lists, and buffers, might be partially allocated among levels based on a static allocation, or perhaps a manual allocation. Dynamic allocation based on current demand would create a covert channel, of course.

Memory management for the address spaces of trusted processes differs from the memory management for single-level process address spaces. Because the storage resources of a trusted multilevel process are allocated among multiple levels, it is not safe to handle them like those of untrusted single-level processes. The level of an untrusted process labels its whole address space, but the labeling of trusted process storage is not so simple.

The data of a trusted process must always be clearly labeled when it is stored in physical memory, when it is communicated over the memory bus, when it is kept on a paging disk, or when it is sent over communication lines between the processor and the paging disk. Otherwise, it becomes impossible to maintain control over the allocations among levels for various resources, including space in physical memory, access time to the memory bus, storage space on the paging disk, access time to the paging disk, and access time to the lines connecting the processor and the disk. Without explicit labels on trusted process data at all times, current demands would influence the allocation of those resources. Their allocation strategies would degenerate into some variety of dynamic allocation with covert channels and compromise of multilevel security.

C.6 Architectural Implications

Avoiding all covert channels in multilevel processors would require static, delayed, or manual allocation of all the following resources: processor time, space in physical memory, service time from the memory bus, kernel service time, service time from all multilevel processes, and all storage within the address spaces of the kernel and the multilevel processes. We doubt that this can be achieved in a practical, general-purpose processor. Perhaps the simplest strategy, static allocation, would be possible, but then the multilevel processor is not significantly more efficient than a set of single-level processors. It would be better to replace it with single-level processors and have real assurance of freedom from covert channels in processors. We suggest that multilevel systems not have any multilevel processors.

Having no multilevel processors certainly helps to minimize the TCB for mandatory security. This is especially appropriate for the high-assurance systems at the Orange Book classes B3 and A1. Because of the rapid drop in prices for processors and memories and the relatively wide selection of secure single-level processors, limiting a multilevel system to single-level processors may impose little or no penalty in efficiency. We believe the best architecture for most multilevel applications is a Distributed, Single-level-processor, Multilevel-secure (DSM) system. Even if a multilevel application does not need a distributed architecture for any other reason, we feel it should be distributed in order to be multilevel secure.

The network in a DSM system must not introduce covert channels. A simple option is a separate network for each level to connect the single-level processors at that level. A potentially less costly network has multilevel lines connecting all the processors and has the trusted network interface (TNI) units sketched earlier ensuring covert-channel-free allocation of the lines. The two options are functionally equivalent. The difference is in the number and capacity of the lines and in the hardware at the interface between the processors and the network.

Multilevel System Benefits in DSM Systems

Each processor of a DSM systems handles just one level, as in an ISPL system. An important question is whether a DSM system is as limited in its functionality as an ISPL system.

Downgrading, writing up reliably, and maintaining data consistency across levels cannot be fully automated as they can be in systems with multilevel processors and covert channels, but they can at least be more automated than in an ISPL system. Many, perhaps most, multilevel applications require none of these functions, but some do need one or more of them. Manual contributions to reliable writing up or to data consistency are inconvenient, but the only practical alternatives compromise multilevel security. Downgrading is so fraught with risk that it is reasonable to insist that some critical step be performed manually. The inconvenience is worthwhile.

Reading down is the essence of multilevel processing. Users perceive a system as multilevel if they have a choice of levels at which to work and if they can refer to the data at lower levels while creating or updating data at the current working level. Reading down and ordinary single-level services are sufficient for most multilevel applications. DSM systems need not have the same problems with reading down as ISPL systems do. Reading down can be supported with multilevel disk drives similar to those described earlier. However, most disk drives in a practical DSM system should probably still each service a single level.

Some multilevel hardware in DSM systems can also escape the limitations on resource allocation in ISPL systems. Cost and convenience arguments justify static allocation of multilevel network lines and manual allocation of such resources as terminals, tape drives, and printers.

Partitioning Levels

In the classification scheme of the U.S. Department of Defense, there are four hierarchical levels: unclassified, confidential, secret, and top secret. A level at which data is classified might also be one of the four hierarchical levels plus a set of nonhierarchical compartments. Many other classification schemes are similar. A user's clearance is the highest level of data the user may see. The clearance is the hierarchical level to which the user is cleared plus any compartments for which the user is cleared.

As noted above, it is best to run a multilevel application as system high if every user has the same clearance, covering all data levels in the application, no matter how many. However, a DSM system is appropriate when some users have different clearances and data is classified over a range of levels. Normally, a DSM system has different processors for each different data level. This is practical for many multilevel applications, ones with data at only two levels or at only a few levels. Some other applications, though, involving various nonhierarchical compartments use dozens or even hundreds of data levels. Processor prices may be falling, but a DSM system with at least one single-level processor for each of hundreds of levels would be impractical. However, a DSM solution may still be reasonable, provided that the number of different user clearances is fairly small, even though the number of different data levels is large.

We describe a DSM system with many data levels, many users, and a handful of different user clearances. A few users, perhaps just the system administrators, might be cleared for all levels, but most would have limited clearances. Probably, those clearances differ in their sets of compartments. The data levels are partitioned based on the overlaps and differences between pairs of clearances. Each partition contains one or more data levels; each data level belongs in one partition; and each clearance includes one or more complete partitions. In the best case, there are exactly as many partitions as clearances, but usually there would be more partitions.[39]

The processors are allocated, not to a single level, but rather to a single partition. A processor may handle data at every level within its partition and may communicate with any other processors sharing the same partition. It should have functionality similar to that required for class B1 in the Orange Book.

A user of a single-partition processor could be anybody whose clearance includes the partition. Because of how the levels are partitioned, the user's clearance will include all or none of the levels in the partition. This is why multilevel security is not compromised even though we expect the processor to have plenty of covert channels. The channels are tolerable because their exploitation could leak information only between levels in the same partition. A malicious user cleared for one level in a partition would not bother to exploit a covert channel in order to access another level in the partition because the user's clearance must include the other level, too.

Because covert channels can still leak within a partition, printed output from a partitioned DSM system can safely be released without review only if the label that the system generated is the highest level of the partition. Users can release output with other labels after manually confirming the labels.

C.7 Conclusions

Until feasible techniques are found to develop a covert-channel-free TCB for a practical multilevel processor, most multilevel systems should be DSM systems with some multilevel disks and perhaps other multilevel devices, but with no general-purpose, multilevel processors. The current research and development efforts on multilevel systems seem to focus on operating systems for multilevel processors, database management systems for multilevel processors, multilevel networks among multilevel processors, and distributed operating systems with multilevel processors. These systems are suitable only for installations that really must tolerate compromises of multilevel security through covert channels.

Promising directions for new efforts to serve secure installations include the development of multilevel disk drives and trusted network interfaces without covert channels. Other efforts should examine how single-level processors can use the multilevel disks and networks to build basic DSM systems that provide reading down in addition to the regular services of single-level distributed

[39] In the worst case, n mutually incomparable clearances form $2^n - 1$ partitions. Probably, the levels in most of those partitions would never be used to classify any data in the system and so would never need resources. Partitions with no resource needs can be ignored.

systems. Further efforts should enhance the basic DSM systems to build more sophisticated DSM systems or multilevel database management systems.

Because these implications for multilevel system architectures represent such a radical shift from the predominant direction of research and development, we encourage readers to dispute our conclusions. Optimists may wish to explain why most installations should tolerate covert channels or how a practical, general-purpose, multilevel processor can be developed with no covert channels. Pessimists may wish to explain why multilevel disk drives or trusted network interfaces cannot be developed without covert channels or why they could not be used to build practical DSM systems. We feel that avoiding all covert channels makes good sense for multilevel systems, that the current dismal state of the art is sufficient evidence of the unsuitability of architectures with multilevel processors, and that it is worth a serious effort to build a prototype of a covert-channel-free, multilevel system that has multilevel disk drives and single-level processors instead of multilevel processors.

D Contributions to the Early Verification Workshops

Following are the lists of contributions for each of the first three verification workshops. The page number, author(s), and title are given for each item in the indicated issue of the ACM SIGSOFT *Software Engineering Notes.* This material is of considerable historical interest and importance, especially because the early work and the experience gained therefrom seem to be too easily forgotten.

D.1 VERkshop I

VERkshop I was held at SRI, Menlo Park, California, 21-23 April 1980 [165].

Summary statements

5 Don Good, Verification environments

5 Bob Boyer, Theorem provers

6 Karl Levitt, Methodology and specifications

6 Susan Gerhart, Verifying systems and networks

7 Jon Millen, Formal models and security

7 Peter Neumann, Conclusions

Contributions

8 Stephen T. Walker, Department of Defense, Thoughts on the impact of verification technology on trusted computer systems

9 V.G. Cerf, W.E. Carlson, L.E. Druffel, ARPA, ARPA interests in applications of program verification

9 Karl N. Levitt, Peter G. Neumann, SRI, An overview of SRI work in verification

11 Gregory A. Haynes, Texas Instruments, Position paper on program verification

12 Dan Craigen, David Bonyun, I.P. Sharp Associates, Ottawa, Canada, Two projects in program verification

13 Richard M. Cohen, The University of Texas at Austin, A review of the Gypsy verification environment

13 Steven German, Friedrich von Henke, David Luckham, Derek Oppen, Wolfgang Polak, Stanford AI, Program verification at Stanford

16 Bob Boyer, J Moore, SRI, The Fortran verification system

17 Mark Moriconi, SRI, Toward incremental and language-independent program verification systems

18 Stephen D. Crocker, USC-ISI, Toward practical verification systems

21 David Thompson, USC/ISI, User interfaces, user models, and user habitability

22 Donald I. Good, The University of Texas at Austin, The problem with program verification is computer science

23 J Moore, SRI, A statement of position

24 Avra Cohn, Edinburgh/ISI, Remarks on machine proof

26 Bob Boyer, J Moore, SRI, A theorem-prover for recursive functions

27 David R. Musser, Computer Science, GE Research & Development Center, The unique termination method of program verification

28 V.R. Pratt, MIT, Modeling as a paradigm for verification

29 Joseph Goguen, SRI, Thoughts on specification, design and verification

33 Jim Keeton-Williams, The MITRE Corporation, Needed: verifiable guidelines on how to design a methodology

33 John Scheid, System Development Corporation, INA JO: SDC's formal development methodology

34 Susan Owicki (Stanford), Leslie Lamport (SRI), Concurrent program verification

36 Gregor v.Bochmann, Stanford and Univ. Montreal, On the construction of submodule specifications

36 Dick Kemmerer, Bruce Walker, Gerald Popek, UCLA, Retrospective: verification experiences with the UCLA operating system kernel

38 S.L. Gerhart, USC/ISI, Applications of Affirm to protocol specification and verification

38 Donald I. Good, Michael K. Smith, The University of Texas at Austin, A verified distributed system

40 R.A. Kemmerer, M. Schaefer, System Development Corporation, Applications of SDC's formal development methodology

41 W.E. Boebert, Honeywell Sys&Res Center, and Univ. Minnesota, Formal verification of embedded software

42 Midge Corasick, The MITRE Corporation, MITRE computer security verification activities

42 Jonathan K. Millen, The MITRE Corporation, Verification of security properties

43 Stanley R. Ames, Jr., James G. Keeton-Williams, The MITRE Corporation, *Excerpts from* Demonstrating security for trusted applications on a security kernel base

44 R.J. Feiertag, SYTEK Inc., Automated proof of multilevel security

46 Carl Landwehr, NRL, Assertions for verification of multilevel-secure military message systems

D.2 VERkshop II

VERKshop II was held at NIST, Gaithersburg, MD, 21-23 April 1981 [166].

2 Peter G. Neumann, Retrospective introduction to VERkshop II

3 Stephen T. Walker, Introductory comments

3 Vinton G. Cerf, A view of verification technology

4 Donald I. Good, Toward building verified, secure systems

8 Ben DiVito, A mechanical verification of the alternating bit protocol

13 Michael K. Smith, Ann E. Siebert, Benedetto L. DiVito, Donald I. Good, A verified encrypted packet interface

16 Susan L. Gerhart, ISI AFFIRM summary

24 Susan L. Gerhart, Research avenues verification is not pursuing, but maybe should be

24 Susan L. Gerhart, ISI high-level theories

25 D.C. Luckham, F.W. von Henke, Program verification at Stanford

27 Karl N. Levitt, Peter G. Neumann, recent SRI work in verification

35 J Moore, SRI experience in writing VCG systems

38 J.A. Goguen, More thoughts on specification and verification

41 Richard A. Kemmerer, Status report on SDC's formal development methodology

43 J.R. Landauer, Applications of FDM to KVM and COS/NFE

45 R. Feiertag, T. Berson, An avenue for exploitation and development of verification technology

50 Jim Keeton-Williams, Anne-Marie G. Disceolo, A practical verification system

55 J.K. Millen, Recent verification work at MITRE

55 Gregory A. Haynes, Program verification at Texas Instruments

57 Joseph J. Tardo, Verification: inside/outside views

58 Robert Constable, VERking in constructive set theory

60 David Bonyun, An untitled Canada goose

61 Clark Weissman, Verification targets: birds to shoot at

63 Marv Schaefer, Egrets and regrets

D.3 VERkshop III

VERkshop III was held at Pajaro Dunes, Watsonville, California, 18-21 February 1985 [117].

Bibliography

We include here some of the primary references relevant to system architectures and formal methods with applications to secure systems and networks. All of these references are cited in context within the report. A reader interested in a particular subset of the references should see the relevant sections of the text. Many additional references are of course contained within the cited works. The determined reader is encouraged to follow areas of particular interest, and chase down the references that unfold.

Of particular historical interest are the VERkshop proceedings identified in Appendix D. In addition, the works of Parnas, Dijkstra, and Hoare are important for methodology. The body of literature is growing rapidly. Some of these cited works will be of less interest in another decade — except to the historians. However, we have attempted to focus on fundamental concepts and principles, and many of the references thereto should remain valid for a long time to come – at least to those interested in the historical evolution of the field.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. Technical Report 70, DEC Systems Research Center, Palo Alto, California, 28 February 1991.

[2] M. Abadi and L. Lamport. Composing specifications. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 1–41, REX Workshop, Mook, The Netherlands, May-June 1989. Springer-Verlag, Lecture Notes in Computer Science Vol. 230.

[3] M. Abadi and R.M. Needham. Prudent engineering practice for cryptographic protocols. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 122–136, Oakland, California, May 1994.

[4] M.D. Abrams, E. Amoroso, L.J. LaPadula, T.F. Lunt, and J.N. Williams. Report of an integrity working group. Technical report, MITRE Corp. (Abrams), McLean, Virginia, November 1991.

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

[6] D. Anderson, T. Frivold, A. Tamaru, and A. Valdes. Next-generation intrusion-detection expert system (NIDES): User manual for security officer user interface (SOUI). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 19 May 1994.

[7] D. Anderson, T. Frivold, and A. Valdes. Next-generation intrusion-detection expert system (NIDES): Final technical report. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 16 November 1994.

[8] R.S. Arbo, E.M. Johnson, and R.L. Sharp. Extending mandatory access controls to a networked mls environment. In *Proc. 12th National Computer Security Conference*, pages 286–295, Baltimore, 10-13 October 1989. NCSC/NIST.

[9] J.M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[10] L. Badger, D.F. Sterne, D.L. Sherman, K.M. Walker, and S.A. Haghighat. Practical domain and type enforcement for Unix. In *Proceedings of the 1995 Symposium on Security and Privacy*, pages 66–77, Oakland, CA, May 1993. IEEE Computer Society.

[11] G. Barrett. Model checking in practice: The t9000 virtual channel processor. *IEEE Transactions on Software Engineering*, 21(2):69–78, February 1995. Special section on Formal Methods Europe '93.

[12] D.E. Bell and L.J. La Padula. Secure computer systems : Volume I – mathematical foundations; volume II – a mathematical model; volume III – a refinement of the mathematical model. Technical Report MTR-2547 (three volumes), The Mitre Corporation, Bedford, MA, March–December 1973.

[13] D.E. Bell and L.J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, The Mitre Corporation, Bedford MA, March 1976.

[14] S.M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. In *USENIX Conference Proceedings, Winter '91*, January 1991. A version of this paper appeared in *Computer Communications Review,* October 1990.

[15] T.C. Vickers Benzel and D.A. Tavilla. Trusted software verification: A case study. In *Proceedings of the 1985 Symposium on Security and Privacy*, pages 14–31, Oakland, CA, April 1985. IEEE Computer Society.

[16] 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.

[17] T.A. Berson, R.J. Feiertag, and R.K. Bauer. Processor-per-domain guard architecture. In *Proceedings of the 1983 IEEE Symposium on Security and Privacy*, page 120, Oakland, CA, April 1983. IEEE Computer Society. (Abstract only).

[18] W.R. Bevier. A verified operating system kernel. Technical report, Ph.D. thesis, Department of Computer Science, The University of Texas at Austin, 1987.

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

[20] J. Bicarregui and B. Ritchie. Invariants, frames, and postconditions: A comparison of the VDM and B notations. *IEEE Transactions on Software Engineering*, 21(2):79–89, February 1995. Special section on Formal Methods Europe '93.

[21] M. Blaze. Protocol failure in the escrowed encryption standard. In *Second ACM Conference on Computer and Communications Security*, pages 59–67, Fairfax, Virginia, November 1994. ACM SIGSAC.

[22] C. Blundo, A. De Santis, G. Di Crescenzo, A.G. Gaggia, and U. Vaccaro. Multi-secret sharing schemes. In *Advances in Cryptology: Proceedings of CRYPTO '94 (Y.G. Desmedt, editor)*, pages 150–163, Berlin, 1994. Springer-Verlag LCNS 839.

[23] 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.

[24] D. Bonyun. Rules as the basis of access control in database management systems. In *7th DoD/NBS Computer Security Initiative Conf., NBS, Gaithersburg, Maryland*, pages 38–47, 24-26 September 1984.

[25] A. Boswell. Specification and validation of a security policy model. *IEEE Transactions on Software Engineering*, 21(2):63–69, February 1995. Special section on Formal Methods Europe '93.

[26] R.S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[27] R.S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[28] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[29] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[30] R.W. Butler. An elementary tutorial on formal specification and verification using PVS. Technical report, NASA Langley Research Center, Hampton, Virginia, June 1993.

[31] R.W. Butler. Formal methods and NASA. In *Proceedings of the Eighteenth National Computer Security Conference*, Baltimore, Maryland, 10–13 October 1995. NIST/NCSC.

[32] R.W. Butler, J.L. Caldwell, V.A. Carreño, C.M. Holloway, P.S. Miner, and B.L. DiVito. NASA Langley's research and technology-transfer program in formal methods. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, pages 135–149. IEEE, June 1995.

[33] R.W. Butler, J.L. Caldwell, V.A. Carreño, C.M. Holloway, P.S. Miner, and B.L. DiVito. NASA Langley's research and technology-transfer program in formal methods. In *Proceedings of the Third NASA Langley Formal Methods Workshop, May 10-12, 1995*, pages 247–268. NASA Langley Research Center, June 1995. This is a longer but earlier version of [32], and includes 28 more references.

[34] *Canadian Trusted Computer Product Evaluation Criteria*. Canadian Systems Security Centre, Communications Security Establishment, Government of Canada., January 1993. Final Draft, version 3.0e.

[35] R.A. Carlson and T.F. Lunt. The trusted domain machine: A secure communication device for security guard applications. In *Proceedings of the 1986 Symposium on Security and Privacy*, pages 182–186, Oakland CA, April 1986. IEEE Computer Society.

[36] T.A. Casey, Jr., S.T. Vinter, D.G. Weber, R. Varadarajan, and D. Rosenthal. A secure distributed operating system. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, April 1988.

[37] NASA Langley Research Center. *Formal Methods Specification and Verification, Volume I.* NASA, June 1995.

[38] NASA Langley Research Center. *Formal Methods Specification and Verification, Volume II.* NASA, Fall 1995.

[39] Secure Computing Technology Center. LOCK formal top level specification, volumes 1-6. Technical report, SCTC, 1988.

[40] Secure Computing Technology Center. LOCK software B-specification, vol. 2. Technical report, SCTC, 1988.

[41] W.R. Cheswick and S.M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, Reading, Massachusetts, 1994.

[42] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987. IEEE Computer Society.

[43] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[44] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[45] D. Craigen. A formal specification of the LSI Guard. Technical Report TR-5031-82-2, I.P. Sharp Associates, Ottawa, August 1982.

[46] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical report, U.S. National Institute of Standards and Technology, Gaithersburg, Maryland, March 1993. Also available from U.S. Naval Research Laboratory and the Atomic Energy Board of Canada.

[47] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995. Special section on Formal Methods Europe '93.

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

[49] 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.

[50] D.E. Denning, T.F. Lunt, R.R. Schell, W.R. Shockley, and M. Heckman. The SeaView security model. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, April 1988.

[51] D.E. Denning and P.G. Neumann. Requirements and model for IDES – a real-time intrusion-detection expert system. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1985.

[52] Y. Desmedt, Y. Frankel, and M. Yung. Multi-receiver/multi-sender network security: Efficient authenticated multicast/feedback. In *Proceedings of IEEE INFOCOM*. IEEE, 1992.

[53] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(5), November 1976.

[54] E.W. Dijkstra. Co-operating sequential processes. In *Programming Languages, F. Genuys (editor)*, pages 43–112. Academic Press, 1968.

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

[56] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[57] D.L. Dill. Model checking. In *Proceedings of the Third NASA Langley Formal Methods Workshop, May 10-12, 1995*, pages 211–216. NASA Langley Research Center, June 1995.

[58] J.E. Dobson and B. Randell. Building reliable secure computing systems out of unreliable unsecure components. In *Proceedings of the 1986 Symposium on Security and Privacy*, pages 187–193, Oakland, CA, April 1986. IEEE Computer Society.

[59] A. Downing, I. Greenberg, and T. Lunt. Issues in distributed system security. In *Proceedings of the Fifth Aerospace Computer Security Conference*, December 1989.

[60] European Communities Commission. *Information Technology Security Evaluation Criteria (ITSEC), Harmonised Criteria of France, Germany, the Netherlands, and the United Kingdom*, 2 May 1990. Draft, Version 1, Available from UK CLEF, CESG Room 2/0805, Fiddlers Green Lane, Cheltenham UK GLOS GL52 5AJ, or GSI/GISA, Am Nippenkreuz 19, D 5300 Bonn 2, Germany.

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

[62] 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.

[63] W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is our Business, A Birthday Salute to Edsger W. Dijkstra*. Springer-Verlag, 11 May 1990.

[64] J. Fellows, J. Hemenway, and N. Kelem. The architecture of a distributed trusted computing base. In *10th National Computer Security Conference*, pages 68–77, Baltimore, Maryland, 21-24 September 1987. Reprinted in Rein Turn (ed.), *Advances in Computer System Security*, Vol. 3, Artech House, Dedham MA, 1988.

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

[66] S. Garfinkel. *PGP: Pretty Good Privacy.* O'Reilly & Associates, Sebastopol CA 95472, 1995.

[67] M. Gasser. An optimization for automated information flow analysis. *Cipher (Newsletter of the IEEE Technical Committee on Security and Privacy)*, pages 32–36, January 1989.

[68] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital distributed system security architecture. In *Proceedings of the Twelfth National Computer Security Conference*, pages 305–319, Baltimore, Maryland, 10–13 October 1989. NIST/NCSC.

[69] V. Gligor. A note on the denial-of-service problem. In *Proceedings of the 1983 Symposium on Security and Privacy*, pages 139–149, Oakland, CA, April 1983. IEEE Computer Society.

[70] V.D. Gligor, R. Kailar, S. Stubblebine, and L. Gong. Logics for cryptographic protocols – virtues and limitations. In *Proceedings of the 4th IEEE Computer Security Foundations Workshop*, pages 219–226, Franconia, New Hampshire, June 1991.

[71] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1988.

[72] B.D. Gold, R.R. Linde, and P.F. Cudney. KVM/370 in retrospect. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 13–23, Oakland, CA, April 1984. IEEE Computer Society.

[73] 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.

[74] L. Gong. Using one-way functions for authentication. *ACM Communications Review*, 19(5):8–11, October 1989.

[75] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of the 1990 Symposium on Research in Security and Privacy*, pages 234–248, Oakland, CA, May 1990. IEEE Computer Society.

[76] D.I. Good. *Revised Report on Gypsy 2.1: DRAFT, July 1984.* Institute for Computing Science, The University of Texas at Austin, 1984.

[77] D.I. Good, R.L. Akers, and L.M. Smith. *Report on Gypsy 2.05: October 1986.* Computational Logic Inc., 1986.

[78] D.I. Good, A.E. Siebert, and L.M. Smith. The message flow modulator, final report. Technical Report 34, Institute for Computing Science and Computer Applications, The University of Texas, Austin, TX, December 1982.

[79] R.M. Graham. Protection in an information processing utility. *Communications of the ACM*, 11(5), May 1968.

[80] J.W. Gray III. Toward a mathematical foundation for information flow security. In *Proceedings of the 1991 Symposium on Research in Security and Privacy*, pages 21–34, Oakland, CA, May 1991. IEEE Computer Society.

[81] M.J. Grohn. A model of a protected data management system. Technical Report ESD-TR-76-289, I.P. Sharp Associates Ltd., June 1976.

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

[83] J.T. Haigh et al. Assured service concepts and models, final technical report, vol. 3: Security in distributed systems. Technical report, Secure Computing Technology Corporation, July 1991.

[84] J.T. Haigh et al. Assured service concepts and models, final technical report, vol. 4: Availability in distributed MLS systems. Technical report, Secure Computing Technology Corporation, July 1991.

[85] J.T. Haigh et al. Assured service concepts and models, final technical report, volume 1: Summary. Technical report, Secure Computing Technology Corporation, July 1991.

[86] R.W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29:147–60, 1950.

[87] Z. Har'El and R.P. Kurshan. Software for analytic development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, January-February 1990.

[88] B.A. Hartman. A Gypsy-based kernel. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 219–225, Oakland, CA, April 1984. IEEE Computer Society.

[89] T.H. Hinke and M. Schaefer. Secure data management system. Technical report, Rome Air Development Center, November 1975. RADC-TR-266 (NTIS AD A019201).

[90] H.M. Hinton. *Composable Safety and Progress Properties.* PhD thesis, University of Toronto, 1995.

[91] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[92] C.M. Holloway, editor. *Third NASA Langley Formal Methods Workshop*, Hampton, Virginia, May 10-12 1995. NASA Langley Research Center. NASA Conference Publication 10176, June 1995.

[93] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[94] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 Symposium on Research in Security and Privacy*, pages 8–20, Oakland, CA, May 1991. IEEE Computer Society.

[95] J. Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, February 1995. Special section on Formal Methods Europe '93.

[96] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In *In Advanced Topics in Dataflow Computing and Multithreading (edited by L. Bic, J-L. Gaudiot, and G. Gao).* IEEE Computer Society, April 1995.

[97] R. Jagannathan and C. Dodd. GLU programmer's guide v0.9. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1994. CSL Technical Report CSL-94-06.

[98] R. Jagannathan, T.F. Lunt, D. Anderson, C. Dodd, F. Gilham, C. Jalali, H.S. Javitz, P.G. Neumann, A. Tamaru, and A. Valdes. System Design Document: Next-generation intrusion-detection expert system (NIDES). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 9 March 1993.

[99] H.S. Javitz, A. Valdes, D.E. Denning, and P.G. Neumann. Analytical techniques development for a statistical intrusion-detection system (SIDS) based on accounting records. Technical report, SRI International, Menlo Park, CA, July 1986. not available for distribution.

[100] A. Jirachiefpattana and R. Lai. An Estelle-NPN based system for protocol verification. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, pages 245–259. IEEE, June 1995.

[101] D.R. Johnson, F.F. Saydjari, and J.P. Van Tassel. MISSI security policy: A formal approach. Technical report, NSA R2SPO-TR001-95, 18 August 1995.

[102] R. Kailar, V.D. Gligor, and L. Gong. On the security effectiveness of cryptographic protocols. In *Proceedings of the 1994 Conference on Dependable Computing for Critical Applications*, pages 90–101, San Diego, CA, January 1994.

[103] 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.

[104] P.A. Karger. Implementing commercial data integrity with secure capabilities. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 130–139, Oakland, CA, April 1988. IEEE Computer Society.

[105] 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.

[106] P.A. Karger and J.C. Wray. Storage channels in disk arm optimization. In *Proceedings of the 1991 Symposium on Research in Security and Privacy*, pages 52–61, Oakland, CA, May 1991. IEEE Computer Society.

[107] T.F. Keefe and W.T. Tsai. Multiversion concurrency control for multilevel secure database systems. In *Proceedings of the 1990 Symposium on Research in Security and Privacy*, pages 369–383, Oakland, CA, May 1990. IEEE Computer Society.

[108] S. Kent. Privacy enhancement for Internet electronic mail: Part I. RFC 1113. Technical report, Internet Activities Board Privacy Task Force, August 1989.

[109] S. Kent and J. Linn. Privacy enhancement for Internet electronic mail: Part II: Certificate based key management. RFC 1114. Technical report, Internet Activities Board Privacy Task Force, August 1989.

[110] S.T. Kent. Internet privacy enhanced mail. *Communications of the ACM*, 36(8):48–60, August 1993.

[111] J.C. Knight, J.C. Prey, and W.A. Wulf. Undergraduate computer science education: A new curriculum philosophy and overview. In *Proceedings of the ACMCSE*, Phoenix, Arizona, March 1994.

[112] R. Kurshan. Algorithmic verification. In *Proceedings of the Eighteenth National Computer Security Conference*, Baltimore, Maryland, 10–13 October 1995. NIST/NCSC.

[113] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, July 1982.

[114] T.M.P. Lee. Using mandatory integrity. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 140–146, Oakland, CA, April 1988. IEEE Computer Society.

[115] W. Legato. Formal methods: Changing directions. In *Proceedings of the Eighteenth National Computer Security Conference*, Baltimore, Maryland, 10–13 October 1995. NIST/NCSC.

[116] N.G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, Reading, Massachusetts, 1995.

[117] K.N. Levitt, S. Crocker, and D. Craigen, editors. VERkshop III: Verification workshop. *ACM SIGSOFT Software Engineering Notes*, 10(4):1–136, August 1985.

[118] P.D. Lincoln and J.M. Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, 1993.

[119] P.D. Lincoln and J.M. Rushby. Formally verified algorithms for diagnosis of manifest, symmetric, link, and byzantine faults. Technical Report SRI-CSL-95-14, Computer Science Laboratory, SRI International, Menlo Park, California, October 1995.

[120] P.D. Lincoln, J.M. Rushby, N. Suri, and C. Walter. Hybrid fault algorithms. In *Proceedings of the Third NASA Langley Formal Methods Workshop, May 10-12, 1995*, pages 193–209. NASA Langley Research Center, June 1995.

[121] J. Linn. Privacy enhancement for Internet electronic mail: Part III: Algorithms, models and identifiers. RFC 1115. Technical report, Internet Activities Board Privacy Task Force, August 1989.

[122] J. Linn. Practical authentication for distributed computing. In *Proceedings of the 1990 Symposium on Research in Security and Privacy*, pages 31–40, Oakland, CA, May 1990. IEEE Computer Society.

[123] S.B. Lipner. Non-discretionary controls for commercial applications. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 2–10. IEEE, 1982. Oakland, CA, 26–28 April 1982.

[124] T.F. Lunt, D.E. Denning, R.R. Schell, M. Heckman, and W.R. Shockley. Secure distributed data views: Formal security policy model. Technical Report RADC-TR-89-313, vol. II (of five), Rome Air Development Center, 1989.

[125] T.F. Lunt, D.E. Denning, R.R. Schell, M. Heckman, and W.R. Shockley. The SeaView security model. *IEEE Transactions on Software Engineering*, 16(6):593–607, June 1990.

[126] T.F. Lunt and D. Hsieh. The SeaView secure database system: A progress report. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS 90)*, Toulouse, France, October 1990. IEEE Computer Society.

[127] T.F. Lunt, R. Jagannathan, R. Lee, S. Listgarten, D.L. Edwards, P.G. Neumann, H.S. Javitz, and A. Valdes. Development and application of IDES: A real-time intrusion-detection expert system. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1988.

[128] T.F. Lunt, R.R. Schell, W.R. Shockley, M. Heckman, and D. Warren. A near-term design for the SeaView multilevel database system. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 234–244, Oakland, CA, April 1988. IEEE Computer Society.

[129] D. MacKenzie. The automation of proof: A historical and sociological explanation. *IEEE Annals of the History of Computing*, 17(3):7–29, Fall 1995.

[130] W.T. Maimone and I.B. Greenberg. Single-level multiversion schedulers for multilevel secure database systems. In *Proceedings of the Sixth Annual Computer Security Applications Conference*, December 1990.

[131] A.P. Maneki. Algebraic properties of system composition in the Loral, Ulysses and McLean trace models. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, Kenmare, County Kerry, Ireland, June 1995.

[132] W. Mao. An augmentation of BAN-like logics. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, Kenmare, County Kerry, Ireland, June 1995.

[133] 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.

[134] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 161–166, Oakland, CA, April 1987. IEEE Computer Society.

[135] D. McCullough. Noninterference and composability of security properties. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 177–186, Oakland, CA, April 1988. IEEE Computer Society.

[136] D. McCullough. Ulysses security properties modeling environment: The theory of security. Technical report, Odyssey Research Associates, Ithaca, NY, July 1988.

[137] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6), June 1990.

[138] J. McHugh and D.I. Good. An information flow tool for Gypsy. In *Proceedings of the 1985 Symposium on Security and Privacy*, pages 46–48, Oakland, CA, April 1985. IEEE Computer Society.

[139] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 Symposium on Research in Security and Privacy*, pages 79–93, Oakland, CA, May 1994. IEEE Computer Society.

[140] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, Massachusetts, 1993.

[141] 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.

[142] S. Micali. Fair public-key cryptosystems. In *Advances in Cryptology: Proceedings of CRYPTO '92 (E.F. Brickell, editor)*, pages 512–517, Berlin, 1992. Springer-Verlag LCNS 740.

[143] E.F. Moore and C.E. Shannon. Reliable circuits using less reliable relays. *Journal of the Franklin Institute*, 262:191–208, 281–297, September, October 1956.

[144] M. Moriconi. A designer/verifier's assistant. *IEEE Transactions on Software Engineering*, SE-5(4):387–401, July 1979. Reprinted in *Artificial Intelligence and Software Engineering*, edited by C. Rich and R. Waters, Morgan Kaufmann Publishers, Inc., 1986. Also reprinted in *Tutorial on Software Maintenance*, edited by G. Parikh and N. Zvegintzov, IEEE Computer Society Press, 1983.

[145] M. Moriconi and X. Qian. Correctness and composition of software architectures. *ACM Software Engineering Notes*, 19(5):164–174, December 1994. Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering.

[146] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

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

[148] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba, a distributed operating system for the 1990s. *IEEE Computer*, 33(5):44–53, May 1990.

[149] S.J. Mullender (ed.). *Distributed Systems.* ACM Press, New York, and Addison-Wesley, Reading, Massachusetts, 1989.

[150] NCSC. *Trusted Network Interpretation (TNI).* National Computer Security Center, 1 August 1990. NCSC-TG-011 Version-1, Red Book.

[151] NCSC. *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC).* National Computer Security Center, December 1985. DOD-5200.28-STD, Orange Book.

[152] R.M. Needham and M.D. Schroeder. Using encryption for authentication and authorization systems. *Communications of the ACM*, 21(12):993–999, December 1978.

[153] R.M. Needham and M.D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(1):7, 1987.

[154] R.B. Neely and J.W. Freeman. Structuring systems for formal verification. In *Proc. 1985 Symposium on Security and Privacy*, Oakland CA, April 1985. IEEE Computer Society.

[155] B.C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[156] P.G. Neumann. On the design of dependable computer systems for critical applications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, October 1990. CSL Technical Report CSL-90-10.

[157] P.G. Neumann. Rainbows and arrows: How the security criteria address computer misuse. In *Proceedings of the Thirteenth National Computer Security Conference*, pages 414–422, Washington, DC, 1-4 October 1990. NIST/NCSC.

[158] P.G. Neumann. Can systems be trustworthy with software-implemented crypto? Technical report, SRI International, Menlo Park, California, October 1994.

[159] P.G. Neumann. *Computer-Related Risks.* ACM Press, New York, and Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0-201-55805-X.

[160] P.G. Neumann. The future of formal methods for security: Overview statement. In *Proceedings of the Eighteenth National Computer Security Conference*, Baltimore, Maryland, 10–13 October 1995. NIST/NCSC.

[161] 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, CA, May 1980. second edition, Report CSL-116.

[162] P.G. Neumann, R.S. Fabry, K.N. Levitt, L. Robinson, and J.H. Wensley. On the design of a provably secure operating system. In *Proceedings of the International Workshop On Protection in Operating Systems*, pages 161–175, August 1974.

[163] P.G. Neumann and L. Gong. Minimizing trust in multilevel-secure systems. Technical report, SRI International, Menlo Park, California, 15 March 1994.

[164] P.G. Neumann, N.E. Proctor, and T.F. Lunt. Preventing security misuse in distributed systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1992. Issued as Rome Laboratory report RL-TR-92-152 (for official use only), Rome Lab C3AB, Griffiss AFB NY 13441-5700. Contact Emilie Siarkiewicz, Internet: siarkiewicze@LONEX.RL.AF.MIL, phone 315-330-3241.

[165] P.G. Neumann, editor. VERkshop I: Verification Workshop. *ACM SIGSOFT Software Engineering Notes*, 5(3):4–47, July 1980.

[166] P.G. Neumann, editor. VERkshop II: Verification Workshop. *ACM SIGSOFT Software Engineering Notes*, 6(3):1–63, July 1981.

[167] NIST. Data encryption standard. Technical report, National Institute of Standards and Technology (formerly NBS), 1977.

[168] ORA Corp. Final report for the (THETA) Experimental Secure Distributed Operating System development. Technical report, ORA Corporation, Ithaca, NY, 15 July 1991. Rome Laboratory Contract F30602-88-C-0146, CDRL A021.

[169] ORA Corp. Formal security model specification for the (THETA) Experimental Secure Distributed Operating System development. Technical report, ORA Corporation, Ithaca, NY, 15 July 1991. Rome Laboratory Contract F30602-88-C-0146, CDRL A009.

[170] ORA Corp. Software requirements specification for the (THETA) Experimental Secure Distributed Operating System development. Technical report, ORA Corporation, Ithaca, NY, 15 July 1991. Rome Laboratory Contract F30602-88-C-0146, CDRL A008.

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

[172] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[173] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. Special section on Formal Methods Europe '93.

[174] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), December 1972.

[175] D.L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5), May 1972.

[176] D.L. Parnas. On a 'buzzword': Hierarchical structure. In *Information Processing 74 (Proceedings of the IFIP Congress 1974)*, pages Software: 336–339. North-Holland, 1974.

[177] D.L. Parnas. The influence of software structure on reliability. In *Proceedings of the International Conference on Reliable Software*, pages 358–362, April 1975. Reprinted with improvements in R. Yeh, *Current Trends in Programming Methodology I*, Prentice Hall, 1977, 111–119.

[178] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[179] D.L. Parnas. Mathematical descriptions and specification of software. In *Proc. of IFIP World Congress 1994, Volume I*, pages 354–359. IFIP, August 1994.

[180] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, March 1985.

[181] D.L. Parnas and G. Handzel. More on specification techniques for software modules. Technical report, Fachbereich Informatik, Technische Hochschule Darmstadt, Research Report BS I 75/1, Germany, April 1975.

[182] D.L. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20(12):948–976, December 1994.

[183] D.L. Parnas and W.R. Price. The design of the virtual memory aspects of a virtual machine. In *Proceedings of the ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems.* ACM, March 1973.

[184] D.L. Parnas and W.R. Price. Design of a non-random access virtual memory machine. In *Proceedings of the International Workshop On Protection in Operating Systems*, pages 177–181, August 1974.

[185] D.L. Parnas and D.L. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, July 1975.

[186] D.L. Parnas and Y. Wang. Simulating the behaviour of software modules by trace rewriting systems. *IEEE Transactions of Software Engineering*, 19(10):750–759, October 1994.

[187] W.W. Peterson and E.J. Weldon, Jr. *Error-Correcting Codes, second edition.* MIT Press, Cambridge, Massachusetts, 1972.

[188] R. Pike, D. Resotto, K. Thompson, H. Trickey, T. Duff, and G. Holzmann. Plan 9: The early papers. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, July 1991. (Computing Science Technical Report 158. This report contains seven conference papers presented during 1990 and 1991.).

[189] P.A. Porras and R.A. Kemmerer. Analyzing covert storage channels. In *Proceedings of the 1991 Symposium on Research in Security and Privacy*, pages 36–51, Oakland, CA, May 1991. IEEE Computer Society.

[190] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice-Hall International, Hemel Hempstead, Great Britain, 1991.

[191] N.E. Proctor. SeaView formal specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, April 1991.

[192] N.E. Proctor and P.G. Neumann. Architectural implications of covert channels. In *Proceedings of the Fifteenth National Computer Security Conference*, pages 28–43, Baltimore, Maryland, 13–16 October 1992.

[193] 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.

[194] B. Randell and J.E. Dobson. Reliability and security issues in distributed computing systems. In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles CA, January 1986.

[195] T.R.N. Rao. *Error-Control Coding for Computer Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[196] M. Reiter and K. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[197] R. Rivest. The MD4 message digest algorithm. Technical report, MIT Laboratory for Computer Science, October 1990. TM 434.

[198] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

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

[200] 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 (ed.), Current Trends in Programming Methodology I, Prentice Hall, 61–110*, 1977.

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

[202] A.W. Roscoe and L. Wulf. Composing and decomposing systems under security properties. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, Kenmare, County Kerry, Ireland, June 1995.

[203] J.M. Rushby. A trusted computing base for embedded systems. In *Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference*, pages 294–311, Gaithersburg, Maryland, September 1984.

[204] J.M. Rushby. Mathematical foundations of the MLS tool for revised special. Forthcoming, Computer Science Laboratory, SRI International, Menlo Park, California, 1986.

[205] J.M. Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989. Proceedings of a Symposium held in Glasgow, October 1986.

[206] J.M. Rushby. Composing trustworthy systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1991.

[207] J.M. Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Proceedings of the Thirteenth Conference on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, Aug 1994. ACM.

[208] J.M. Rushby. Fault-tolerant algorithms and the design of PVS. In *Proceedings of the Third NASA Langley Formal Methods Workshop, May 10-12, 1995*, pages 93–104. NASA Langley Research Center, June 1995.

[209] J.M. Rushby. Formal methods and their role in digital systems validation for airborne systems. Technical report, SRI International, Menlo Park, California, CSL-95-01, March 1995.

[210] J.M. Rushby and B. Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, July 1983.

[211] J.M. Rushby and B. Randell. A distributed secure system (extended abstract). In *Proceedings of the 1983 IEEE Symposium on Security and Privacy*, pages 127–135, Oakland, CA, April 1983. IEEE Computer Society.

[212] J.M. Rushby and D.W.J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical report, SRI International, Menlo Park, California, CSL-95-10, October 1995.

[213] J.M. Rushby and F. von Henke. Formal verification of the interactive convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989. Also available as NASA Contractor Report 4239.

[214] T.T. Russell and M. Schaefer. Toward a high B level security architecture for the IBM ES/3090 processor resource/systems manager (PR/SM). In *Proceedings of the Twelfth National Computer Security Conference*, pages 184–196, Baltimore, Maryland, 10–13 October 1989. NIST/NCSC.

[215] 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. Reprinted in Rein Turn (ed.), *Advances in Computer System Security*, Vol. 3, Artech House, Dedham MA, 1988.

[216] O.S. Saydjari, S.J. Turner, D.E. Peele, J.F. Farrell, P.A. Loscocco, W. Kutz, and G.L. Bock. Synergy: A distributed, microkernel-based security architecture. Technical report, NSA INFOSEC Research and Technology, November 22 1993.

[217] M. Schaefer and R.R. Schell. Toward an understanding of extensible architectures for evaluated trusted computer system products. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, April 1984.

[218] M. Schaefer (editor). *Multilevel Data Management Security.* National Academy Press, Air Force Studies Board, National Research Council, Washington DC, 1983. Report of the 1982 Summer Study (For Official Use Only), National Academy of Sciences, Air Force Studies Board, Marvin Schaefer, Chairman; published in 1983).

[219] R. Schell. A security kernel for a multiprocessor microcomputer. *IEEE Computer*, 16(7):47–53, July 1983.

[220] R.R. Schell and T.F. Tao. Microcomputer-based trusted systems for communications and workstation applications. In *Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference*, pages 277–290, Gaithersburg, Maryland, September 1984.

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

[222] F.B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[223] B. Schneier. *Applied Cryptography.* John Wiley and Sons, New York, 1994.

[224] B. Schneier. *E-Mail Security with PGP and PEM.* John Wiley and Sons, New York, 1995.

[225] M.D. Schroeder, D.D. Clark, and J.H. Saltzer. The Multics kernel design project. In *Proceedings of the Sixth Symposium on Operating System Principles*, November 1977. ACM Operating Systems Review 11(5).

[226] M. Seagar, D. Guaspari, M. Stillerman, and C. Marceau. Formal methods in the theta kernel. In *Proceedings of the 1995 Symposium on Security and Privacy*, pages 88–100, Oakland, CA, May 1993. IEEE Computer Society.

[227] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[228] W.R. Shockley. Implementing the Clark/Wilson integrity policy using current technology. Technical report, Gemini Computers, P.O. Box 222417, Carmel CA, 1988. GCI-88-6-01.

[229] J.M. Silverman. Reflections on the verification of the security of an operating system. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 143–154, October 1983.

[230] M.K. Smith, D.I. Good, and B.L. DiVito. *Using the Gypsy Methodology: DRAFT, November 1987.* Computational Logic Inc., 1987.

[231] J.M. Spitzen, K.N. Levitt, and L. Robinson. An example of hierarchical design and proof. *Communications of the ACM*, 21(12):1064–1075, December 1978.

[232] J.M. Spivey. *Understanding Z: a specification language and its formal semantics.* Cambridge University Press, Cambridge, England, 1988.

[233] SRI-CSL. *HDM Verification Environment Enhancements, Interim Report on Language Definition.* Computer Science Laboratory, SRI International, Menlo Park, California, 1983. SRI Project No. 5727, Contract No. MDA904-83-C-0461.

[234] SRI-CSL. EHDM *Specification and Verification System Version 4.1: Preliminary Definition of the* EHDM *Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, California, September 6, 1988.

[235] SRI-CSL. EHDM *Specification and Verification System – Version 6.1: User's Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, March 27, 1992.

[236] J.G. Steiner, C. Neuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, February 1988.

[237] S.G. Stubblebine and V.D. Gligor. On message integrity in cryptographic protocols. In *Proceedings of the 1992 Symposium on Research in Security and Privacy*, pages 85–104, Oakland, CA, May 1992. IEEE Computer Society.

[238] D.I. Sutherland. A model of information flow. In *Proceedings of the Ninth National Computer Security Conference*, pages 175–183, September 1986.

[239] A.S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[240] A.S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[241] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[242] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[243] K. J. Turner, ed. *Using Formal Description Languages.* John Wiley, Chichester, 1993.

[244] S.T. Vinter. Extended discretionary access controls. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, April 1988.

[245] F. von Henke and J.M. Rushby. *Introduction to* EHDM. Computer Science Laboratory, SRI International, Menlo Park, California, September 1988.

[246] F. von Henke, N. Shankar, and J.M. Rushby. *Formal Semantics of* EHDM. Computer Science Laboratory, SRI International, Menlo Park, California, September 28, 1988.

[247] F.W. von Henke, J.S. Crow, R. Lee, J.M. Rushby, and R.A. Whitehurst. The EHDM verification environment: An overview. In *Proceedings of the Eleventh National Computer Security Conference*, pages 147–155, Baltimore, Maryland, October 1988. NBS/NCSC.

[248] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, pages 43–98, Princeton, N.J., 1956. Princeton University.

[249] C. Weissman. Blacker: Security for the DDN. Examples of A1 security engineering trades. In *Proceedings of the 1992 Symposium on Research in Security and Privacy*, pages 286–292, Oakland, CA, May 1992. IEEE Computer Society.

[250] J.H. Wensley et al. Design study of software-implemented fault-tolerance (SIFT) computer. NASA contractor report 3011, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1982.

[251] J.C. Wray. An analysis of covert timing channels. In *Proceedings of the 1991 Symposium on Research in Security and Privacy*, pages 2–7, Oakland, CA, May 1991. IEEE Computer Society.

[252] R. Yahalom, B. Klein, and Th. Beth. Trust relationships in secure systems: A distributed authentication procedure. In *Proceedings of the 1993 Symposium on Research in Security and Privacy*, pages 150–164, Oakland, CA, May 1993. IEEE Computer Society.

[253] W.D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, Computational Logic Incorporated, Austin, TX, October 1988.

[254] C.-F. Yu and V.D. Gligor. A formal specification and verification method for the prevention of denial of service. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 187–202, Oakland, CA, April 1988. IEEE Computer Society.

[255] A. Zakinthinos and E.S. Lee. The composability of non-interference. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, Kenmare, County Kerry, Ireland, June 1995.

[256] P.R. Zimmermann. *The Official PGP User's Guide.* MIT Press, Cambridge, Massachusetts, 1995.