CDRL A001 Final Report December 28, 2004

Principled Assuredly Trustworthy Composable Architectures

Final Report
Contract number N66001-01-C-8040
DARPA Order No. M132
SRI Project P11459

Submitted by: Peter G. Neumann, Principal Investigator
Principal Scientist, Computer Science Laboratory
SRI International EL-243, 333 Ravenswood Ave
Menlo Park, California 94025-3493, USA
Neumann@csl.sri.com; http://www.csl.sri.com/neumann
Phone: 1-650-859-2375; Fax: 1-650-859-2844

Prepared for:
Contracting Officer, Code D4121
SPAWAR Systems Center
San Diego, California

Approved:
Patrick Lincoln, Director
Computer Science Laboratory
William Mark, Vice President
Information and Computing Sciences Division

This report is available on-line for browsing
  http://www.csl.sri.com/neumann/chats4.html
and also for printing or displaying
  http://www.csl.sri.com/neumann/chats4.pdf
  http://www.csl.sri.com/neumann/chats4.ps

Contents

List of Tables

List of Figures

Preface

This document is the final report for Task 1 of SRI Project 11459, Architectural Frameworks for Composable Survivability and Security, under DARPA Contract No. N66001-01-C-8040 as part of DARPA's Composable High-Assurance Trustworthy Systems (CHATS) program. Douglas Maughan was the DARPA Program Manager through the first two years of the project. He has been succeeded by Tim Gibson.

Acknowledgments are given at the end of the body of this report. However, the author would like to give special mention to the significant contributions of Drew Dean and Virgil Gligor.

Abstract

This report presents the results of our DARPA CHATS project. We characterize problems in and approaches to attaining computer system and network architectures. The overall goal is to be better able to develop and more rapidly configure highly trustworthy systems and networks able to satisfy critical requirements (including security, reliability, survivability, performance, and other vital characteristics). We consider ways to enable effective systems to be predictably composed out of interoperable subsystems, to provide the required trustworthiness — with reasonably high assurance that the critical requirements will be met under the specified operational conditions, and (we hope) that do something sensible outside of that range of operational conditions. This work thus spans the entire set of goals of the DARPA CHATS program — trustworthiness, composability, and assurance — and much more.

By *trustworthiness,* we mean simply *worthy of being trusted to fulfill whatever critical requirements may be needed* for a particular component, subsystem, system, network, application, mission, enterprise, or other entity. Trustworthiness requirements might typically involve (for example) attributes of security, reliability, performance, and survivability under a wide range of potential adversities. Measures of trustworthiness are meaningful only to the extent that (a) the requirements are sufficiently complete and well defined, and (b) can be accurately evaluated.

This report should be particularly valuable to system developers who have the need and/or the desire to build systems and networks that are significantly better than today's conventional mass-market and custom software. The conclusions of the report can also be useful to government organizations that fund research and development efforts, and to procurers of systems that must be trustworthy.

Executive Summary

Anyone will renovate his science who will steadily look after the irregular phenomena. And when the science is renewed, its new formulas often have more of the voice of the exceptions in them than of what were supposed to be the rules. William James

In this report, we confront an extremely difficult problem — namely, how to attain demonstrably trustworthy systems and networks that must operate under stringent requirements for security, reliability, survivability, and other critical attributes, and that must be able to evolve gracefully and predictably over time — despite changes in requirements, hardware, communications technologies, and radically new applications. In particular, we seek to establish a sound basis for the creation of trustworthy systems and networks that can be easily composed out of subsystems and components, with predictably high assurance, and also do something sensible when forced to operate predictably outside of the expected normal range of operational conditions. Toward this end, we examine a set of *principles* for achieving trustworthiness, consider *constraints* that might enhance composability, pursue *architectures* and *trustworthy subsystems* that are inherently likely to result in trustworthy systems and networks, define constraints on *administrative practices* that reduce operational risks, and seek approaches that can significantly increase *assurance.* The approach is intended to be theoretically sound as well as practical and realistic. We also outline directions for new research and development that could significantly improve the future for high-assurance trustworthy systems.

With respect to the future of trustworthy systems and networks, perhaps the most important recommendations involve the urgent establishment and use of *soundly based, highly disciplined, and principle-driven architectures,* as well as *development practices that systematically encompass trustworthiness and assurance as integral parts of what must become coherent development processes and sound subsequent operational practices.* Only then can we have any realistic assurances that our computer-communication infrastructures — and indeed our critical national infrastructures — will be able to behave as needed, in times of crisis as well as in normal operation. The challenges do not have easy turn-the-crank solutions. Addressing them requires considerable skills, understanding, experience, education, and enlightened management. Success can be greatly increased in many ways, including the availability of *reliable hardware components, robust and resilient network architectures and systems, consistent use of good software engineering practices, careful attention to human-oriented interface design, well-conceived and sensibly used programming languages, compilers that are capable of enhancing the trustworthiness of source code, techniques for increasing interoperability among heterogeneous distributed systems and subsystems, methods and tools for analysis and assurance, design and development of systems that are inherently easier to administer and that provide better support for operational personnel, and many other factors.*

The absence or relative inadequacy with respect to each of these factors today represents a potential weak link in a process that is currently riddled with too many weak links. On the other hand, much greater emphasis on these factors can result in substantially greater trustworthiness, with predictable results.

The approach taken here is strongly motivated by historical perspectives of promising research efforts and extensive development experience (both positive and negative) relating to the development of trustworthy systems. It is also motivated by the practical needs and limitations of commercial developments as well as some initial successes in inserting significantly greater discipline into the open-source world. It provides useful guidelines for disciplined system developments and future research.

This report cannot be everything for everyone, although it should have some appeal to a relatively broad range of readers. As a consequence of the inherent complexity associated with the challenges of developing and operating trustworthy systems and networks, we urge readers with experience in software development to read this report thoroughly, to see what resonates nicely with their experience. However, to the inexperienced developer or to the experienced developer who believes in seat-of-the-pants software creation, we offer a few words of caution. Many of the individual concepts should be well known to many of you. However, if you are looking for easy answers, you may be disappointed; indeed, each chapter should in turn convince you that there are no easy answers. On the other hand, if you are looking for some practical advice on how to develop systems that are substantially more trustworthy than what is commercially available today, you may find many encouraging directions to pursue.

Although there are some novel concepts in this report, our main thrust involves various approaches that can make better use of what we have learned over the past many years in the research community and that can be used to better advantage in production systems. Many of the lessons relating to serious trustworthiness can be drawn from past research and prototype development. However, those lessons have been largely ignored in commercial development communities, and perhaps have also been insufficiently observed by the developers of source-available software. There are many directions herein — both new and old — for fruitful research and development that can help to fill in the gaps.

We believe that observance of the approaches described here would greatly improve the present situation. The opportunities for this within the open-source community are considerable, although they are also applicable to closed-source proprietary systems (despite various caveats).

Roadmap of This Report

The outline of this report is as follows.

- Chapter 1 presents some of the terminology and foundations on which this report is based.
- Chapter 2 considers the roles of principles in the conceptualization, design, implementation, operation, and use of information systems and networks having critical requirements for security, reliability, and survivability. On one hand, some of the principles under discussion are well established in the literature and in certain educational curricula, and some are even

intuitively appealing to experienced developers. On the other hand, very few of these principles are seriously observed in conventional commercial programming practice, and therefore are sorely missing in many system architectures, system implementations, programming languages, compilers, software engineering disciplines, and software development tools. Indeed, although these principles are sometimes knee-jerkingly pooh-poohed as being impractical, they are also potentially very valuable.

- Chapter 3 outlines some of the obstacles to achieving facile composability and interoperability, and considers approaches that can contribute to the development of significantly greater composability in systems with critical requirements. In some cases, compositions of seemingly compliant subsystems can actually compromise the ability of the resulting system to satisfy its requirements. Of considerable interest is the concept of combining subsystems in ways that actually increase the resulting trustworthiness, or at least do not diminish it. Also relevant are the concepts of software engineering discipline, programming-language constructs, execution compatibility and interoperability, and development tools.

- Chapter 4 considers characteristics of architectures that are likely to predictably satisfy the CHATS goals, based on the discussion of principles and the analysis of composability. Although various different architectures are needed for different classes of applications, they can share many common principles and attributes.

- Chapter 5 considers further characteristics of architectures for which the interfaces at various layers are relatively perspicuous — that is, understandable because of the way in which they have been designed and implemented.

- Chapter 6 examines techniques for achieving the required trustworthiness with some significant measure of assurance. It stresses the importance of incorporating the principled use of assurance techniques throughout development and operation.

- Chapter 7 reexamines the collected wisdom of the earlier chapters in the light of experience, and seeks to provide practical guidelines for applying that wisdom to the system development process.

- Chapter 8 considers some potentially significant areas for future research and development.

- Chapter 9 provides a summary of the report, the needs for future research and prototype development, and conclusions.

- Appendix A summarizes the work on formally based static code analysis carried out by Hao Chen and David Wagner, under a project subcontract to the University of California at Berkeley. It also notes subsequent work carried out by Hao Chen.

- Appendix B provides some useful background for the material in Chapters 2, 3, and 4, and particularly Chapter 5, relating to system structural and correctness properties associated with modularity. Appendix B is based on material written by Virgil Gligor, in connection with joint work he did with Drew Dean and Peter Neumann for Lee Badger's Visibly Controllable Computing initiative.

Chapter 1

The Foundations of This Report

We essay a difficult task; but there is no merit save in difficult tasks.
Ovid

In the context of this report, the term "trustworthy" is used in a broad sense that is meaningful with respect to any given set of requirements, policies, properties, or other definitional entities. It represents the extent to which those requirements are likely to be satisfied, under specified conditions. That is, trustworthiness means *worthy of being trusted to satisfy the given expectations.* For example, typical requirements might relate to attributes of security, reliability, performance, and survivability under a wide range of potential adversities. Each of these attributes has expectations that are specific to each layer of abstraction (and differing from one layer to another) — for example, with respect to hardware, operating systems, applications, systems, networks, and enterprise layers.

- **Security** has been defined in many different ways — some of which are rather general, and some of which are meaningful only within their own specific contexts. In its strictest sense, security might be thought of as the absence of some set of vulnerabilities. In a more operational sense, it might be thought of as the absence of opportunities for misuse, despite certain vulnerabilities. However, in the realities of expected use, essentially nothing is perfectly secure — particularly in the face of insider misuse. Consequently, security is more commonly thought of as the result of a collection of attempts to prevent bad things from happening and the integrated anticipation of being able to recover suitably when bad things do happen. Security requirements typically specify properties relating to system integrity, data integrity, data confidentiality, and the ability to withstand systemic denial-of-service attacks, along with other more detailed properties such as user authentication, system authentication, user access control, accountability, monitoring, real-time misuse detection, and appropriate responses to detected security violations — among others. The list of detailed requirements may be very inclusive. Security is a comprehensive notion, relevant throughout the processes of system conceptualization, requirements formulation, design, development, use, and operation; it applies to systems in the large as well as subsystems applications, and entire enterprises. Measures of security often tend to be of questionable value — for example, probabilistic measures of how secure something is, which may be meaningful in the small, but almost useless in the large. In any event, there is a need for useful metrics.

- **Reliability** requirements might include properties relating to the ability to tolerate hardware failures and software flaws, the characterization of acceptable degradation in the face of untolerated faults, probabilities of success, expected mean times between failures, and so on. Measures of reliability typically represent the extent to which flaws, failures, and errors [26, 309] can be avoided or tolerated.

- **Performance** requirements might include aggregate throughput measures, processing speeds, storage capacities, and guaranteed real-time response (for example).

- **Survivability** requirements typically address overall system or enterprise availability despite numerous adversities that could compromise the intended goals, and thus (for example) necessarily encompass relevant aspects of security, reliability, performance, and other critical requirements (e.g., [264]). The range of anticipated adversities may be extensive in extremely critical applications.

- **Trustworthiness** can then be thought of as the well-founded assessment of the extent to which a given system, network, or component will satisfy its specified requirements, and particularly those requirements that are critical to an enterprise, mission, system, network, or other entity. Trustworthiness is meaningful only with respect to those expectations.

- **Assurance** provides some sort of measure or indication of the likelihood that the desired trustworthiness is *actually* well founded, typically through a combination of well-specified requirements, structured architectures, formal and nonformal evaluations, and operational practices. A system can be said to be trustworthy only with respect to its stated requirements, with some level of assurance that it will behave as expected relative to those requirements. If the requirements are fundamentally incomplete, the concept of trustworthiness is similarly inherently incomplete.

- **Risk** is sometimes thought of as a mathematical product of aggregate measures of threats, vulnerabilities, and potential quantifiable losses or asset values. However, reducing risk to a single quantifiable number is almost always overly simplistic, because the space being considered is highly multidimensional. More intuitively, **risks** are events that in one sense or another are undesirable. Clearly, some risks are more important than others, and efforts should be made to ensure that the most serious risks are avoided wherever practicable.

Note that these concepts are sometimes interrelated. Achieving survivability in turn requires security, reliability, and some measures of guaranteed performance (among other requirements). Human safety typically does as well. Many of these properties are meaningful in different ways at various layers of abstraction. At the highest layers, they tend to be emergent properties of systems in the large, or indeed entire enterprises — that is, they are meaningful only in terms of the entire system complex rather than as lower-layer properties.

The concept of **trustworthiness** is essentially indistinguishable from what is termed **dependability** [24, 25, 26, 202, 309], particularly within the IEEE and European communities. In its very early days, dependability was focused primarily on hardware faults, subsequently extended to software faults, and then generalized to a notion of faults that includes security threats. In that framework, dependability's generalized notions of fault prevention, fault tolerance, fault removal, and fault forecasting (the last of which has some of the elements of assurance) seem to encompass everything that trustworthiness does, albeit with occasionally different terminology. However, a recent paper, Basic Concepts and Taxonomy of Dependable and Secure Computing, by Avižienis, Laprie, Randell, Landwehr [26] (which is a gold mine for serious researchers) attempts to distinguish security as a specifically identifiable subset of dependability, rather than more generally treating it as one of various sets of application-relevant requirements subsumed under trustworthiness, as we do in this report. (Their new reformulation of security encompasses primarily confidentiality, integrity, and availability — which in this report are only part of the necessary trustworthiness aspects that are required for security — although it also alludes to other attributes of security. However, any differences between their paper and this report are largely academic — we are each approaching the same basic problems.)

We make a careful distinction throughout this report between *trust* and *trustworthiness*. **Trustworthiness** implies that something is worthy of being trusted. **Trust** merely implies that you trust something whether it is trustworthy or not, perhaps because you have no alternative, or because you are naïve, or perhaps because you do not even realize that trustworthiness is necessary, or because of some other reason. We generally eschew the terms *trust* and *trusted* unless we specifically mean trust rather than trustworthiness. (The slogan on an old T-shirt worn around the National Security Agency was "In trust we trust.")

A prophetic definition long ago due to Leslie Lamport can be paraphrased in the context of this report as follows: a *distributed system* is a system in which you have to trust components whose mere existence may be unknown to you. This is increasingly a problem on the World Wide Web, which is today's ultimate distributed system.

There are many R&D directions that we believe are important for the short- and long-term futures — for the computer and network communities at large, for DARPA developers, and for system and network developers generally. (We outline some recommendations for future R&D in Chapter 9.) The basis of our project is the exploration and exploitation of a few of the potentially most timely and significant research and development directions.

- **Principles.** We revisit fundamental principles of trustworthy system development, cull out those likely to be most effective, explore their practical limitations, and seek a basis for principled architectures, principled development, and principled operation.

- **Composability.** We explore existing obstacles to achieving seamless composability and techniques for attaining practical composability in the future. Composability is meaningful at many layers of abstraction, for components, subsystems, networked systems, and networks of networks. It is also applicable to policies, protocols, specifications, formal representations, and proofs. Subsystem composability takes on a variety of forms, including sequential (with or without feedback, with or without recursion, etc.) and parallel execution.

- **Reusability.** One of the side benefits of the effort to ensure composability is the possibility of easier reusability of modules, subsystems, and other components, and particular high-assurance components — in contexts other than their initially conceived applicability.

- **Compatibility and interoperability.** In this report, *compatibility* implies that different entities can coexist without adverse side effects, perhaps without even knowing of each other's existence. *Interoperability* further implies that different entities are actually able to work constructively with one another.

- **Architecture.** By the term *architecture,* we specifically encompass the *structure* of systems and networks at various layers of abstraction (in terms of composable components, subsystems, etc.), the design of the *functional interfaces* particularly at the visible layers of abstraction, and the *interdependencies* between, among, and within those components and layers of abstraction. Architectures can be defined with increasingly refined degrees of specificity — from conceptual coarse-grain representations down to extremely explicit schematics that are sufficiently detailed to being transformed into precise design specifications from which implementations may be carried out relatively unambiguously. In effect, successive degrees of architectural specificity, design specifications, and implementations represent the iterative development process of refinement, instantiation, and necessary revision.

- **Trustworthy principled composable architectures.** We seek to establish principle-based composable distributed-system network-oriented open architectures inherently capable of fulfilling critical security, reliability, survivability, and performance requirements, while being readily evolvable over time to accommodate widely differing applications, different hardware and software providers, and changing technologies.

- **Trustworthy system development and its foundations.** In addition to principled architectures, we seek to provide a sound basis for system requirements, specifications, system development, implementation, trustworthiness, and assurance of that trustworthiness for composable interoperable components, with predictable behavior when composed.

- **Trustworthy protocols.** We need to develop new protocols and/or extend existing protocols that effectively mask the peculiarities of various networking technologies wherever possible, but are able to accommodate a wide range of technologies (e.g., wireless and wired, optical, and electronic), and capable of addressing all relevant critical requirements. Existing protocols (e.g., IPv4, IPSEC, IPv6, secure routing, secure name service) are not adequate for critical-system and network trustworthiness. This is a very difficult challenge, and necessarily needs the involvement of the IETF, NIST standards efforts, and development communities.

- **Principled operational practice.** We need to bring the above concepts into the realm of operational practice, which is seriously in need of greater trustworthiness and controllability. Many of the concepts considered here have considerable potential toward that end, particularly for system and network management.

Throughout the history of efforts to develop trustworthy systems and networks, there is an unfortunate shortage of observable long-term progress relating specifically to the multitude of requirements for security. (See, for example, an interview with Richard Clarke [149] in the *IEEE Security and Privacy.*) Blame can be widely distributed among governments, industries, and users — both personal and corporate. Significant research and development results are typically soon forgotten or else deprecated in practice. Systems have come and gone, programming languages have come and (sometimes) gone, and certain specific systemic vulnerabilities have come and gone. However, many generic classes of vulnerabilities seem to persist forever — such as buffer overflows, race conditions, off-by-one errors, mismatched types, divide-by-zero crashes, and unchecked procedure-call arguments, to name just a few. Overall, it is primarily only the principles that have remained inviolable — at least in principle — despite their having been widely ignored in practice. It is time to change that unfortunate situation, and honor the principles.

There is an unfortunate shortage of fundamental books that provide useful background for the material discussed in this report. Two recent books by Matt Bishop, *Computer Security: Art and Science* [47] and *Introduction to Computer Security* [48], are worthy of particular note — the former for its rather comprehensive and somewhat rigorous computer-science-based treatment of security, and the latter for its less formal approach that should be more accessible to others who are not computer scientists. Chuck Pfleeger's *Security in Computing* [303], Ross Anderson's *Security Engineering* [14], and Morrie Gasser's *Building a Secure Computer System* [127] are also worthy sources. A recent book by Ian Sommerville [359] provides extensive background on software engineering.

A paper [266] summarizing our conclusions as of early 2003 is part of the DISCEX3 proceedings, from the April 2003 DARPA Information Survivability Conference and Exposition.

Chapter 2

Fundamental Principles of Trustworthiness

Synopsis

Enormous benefits can result from basing requirements, architectures, implementations, and operational practices on well-defined and well-understood generally accepted principles.

In this chapter, we itemize, review, and interpret various design and development principles that if properly observed can advance composability, trustworthiness, assurance, and other attributes of systems and networks, within the context of the CHATS effort. We consider the relative applicability of those principles, as well as some of the problems they may introduce.

2.1 Introduction

Everything should be made as simple as possible — but no simpler.
Albert Einstein

A fundamental hypothesis motivating this report is that achieving assurable trustworthiness requires much greater observance of certain underlying principles. We assert that careful attention to such principles can greatly facilitate the following efforts.

- **Principled architectures.** Establishment of composable open distributed-system network-oriented architectures capable of fulfilling critical security, reliability, survivability, and performance requirements, while being readily adaptable to widely differing applications, different hardware, heterogeneous software providers, and changing technologies. As noted in Chapter 1, *architecture* here specifically implies both the structure of systems and networks and the design of their functional interfaces and interconnections, at various layers of abstraction.
- **Principled system development.** Development of specifications, implementation, trustworthiness, and assurance of that trustworthiness for composable interoperable components, with predictable behavior when those components are composed.
- **Principled assurance.** Attainment of assuredly trustworthy systems and networks, capable of addressing all relevant critical requirements, with new or extended protocols that mask the peculiarities of various networking technologies wherever advantageous.

The benefits of disciplined and principled system development cannot be overestimated, especially in the early stages of the development cycle. Principled design and software development can stave off many problems later on in implementation, maintenance, and operation. Huge potential cost savings can result from diligently observing relevant principles throughout the development cycle and maintaining discipline. But the primary concept involved is that of disciplined development; there are many methodologies that provide some kind of discipline, and all of those can be useful in some cases.

In concept, most of the principles discussed here are fairly well known and understood by system cognoscenti. However, their relevance is often not generally appreciated by people with little development or operational experience. Not wishing to preach to the choir, we do not dwell on elaborating the principles themselves, which have been extensively covered elsewhere (see Section 2.3). Instead, we concentrate on the importance and applicability of these principles in the development of systems with critical requirements — and especially secure systems and networks. The clear implication is that disciplined understanding and observance of the most effective of these principles can have enormous benefits to developers and system administrators, and also can aid user communities. However, we also explore various potential conflicts within and among these principles, and emphasize that those conflicts must be thoroughly understood and respected. System development is intrinsically complicated in the face of critical requirements. For example, it is important to find ways to manage that complexity, rather than to mistakenly believe that intrinsic complexity is avoidable by pretending to practice "simplicity".

2.2 Risks Resulting from Untrustworthiness

As noted above, trustworthiness is a concept that encompasses being worthy of trust with respect to whatever critical requirements are in effect, often relating to security, reliability, guarantees of real-time performance and resource availability, survivability in spite of a wide range of adversities, and so on. Trustworthiness depends on hardware, software, communications media, power supplies, physical environments, and ultimately people in many capacities — requirements specifiers, designers, implementers, users, operators, maintenance personnel, administrators, and so on.

There are numerous examples of untrustworthy systems, networks, computer-related applications, and people. We indicate the extensive diversity of cases reported in the past with just a few tidbits relevant to each of various categories. See Computer-Related Risks [260] and the Illustrative Risks index [267] for numerous further examples and references involving many different types of system applications. (In the Illustrative Risks document, descriptors indicate relevance to loss of life, system survivability, various aspects of security, privacy, development problems, human interface confusions, and so on.) Some of these examples are revisited in Section 6.9, in considering how principled architectures and assurance-based risk reduction might have avoided the particular problems.

- **Safety**
  - Aviation disasters, attributable to problems with airframes, avionics computer hardware and software, badly designed human interfaces, pilots, air-traffic control systems, air-

traffic controllers, maintenance crews, airport security lapses, and so on: KAL 007 (flying on erroneous autopilot course), Air New Zealand crash into Mount Erebus (erroneous course data), Lauda Air (thrust reverser accidentally deployed in flight), Iranian Airbus shootdown (bad operational interfaces); Black Hawk helicopter problems, to name just a few.

  - Medical disasters, attributable to hardware flaws, malfunctions, software bugs, and confusing human interfaces: Therac 25 (nonatomic transition from high-intensity to low-intensity mode); database errors resulting in operation failures; electromagnetic interference (pacemakers, defibrillators).

- **Reliability and availability**
  - Failures in defense systems, control systems, telecommunications systems, space, financial systems, etc.: Patriot missiles missing Scuds (excessive clock drift); Yorktown Aegis missile cruiser disabled (Windows NT crashed by divide by zero in an application); ARPANET collapse (1980); AT&T long-distance collapse (1990); first Shuttle launch (Columbia backup computer synchronization problem); massive electrical power outages, with large-scale propagation (e.g., 13-hour northeast power outage of 9 November 1965; lower New York state outage on 13 July 1977; blackout of 10 western U.S. states on 2 October 1984; western U.S. power blackouts on 2 July 1996; North American west-coast power outages on 10 Aug 1996, including eight states, Canada and Baja; massive northeast U.S. power-grid overload and blackout on 14 August 2003.

- **Security**
  - Unintentional security flaws in delivered software, including many whose origins have been well understood for many years but that keep recurring
  - Intentionally installed trapdoors, Trojan horses, other malware
  - Insider and outsider exploitations involving loss of confidentiality, loss of integrity, denials of service, viruses, worms, spam, malware introduced by Web browsing, financial frauds and misuse
  - Operational problems, such as configuration errors, administrator oversights
  - Many other problems. Overall, the situation here is truly deplorable and the risks very diverse. The Illustrative Risks index [267] includes many cases of reported security problems.

- **Survivability**
  - Survivability despite diverse adversities ultimately depends on system and network reliability, security, computational adequacy and bandwidth, database availability, and various other attributes. (For example, see [264].) Some of the problems noted above involve failures of system and network survivability, as a result of hardware and software malfunctions, exploitations of security vulnerabilities, accidents, malice, electromagnetic interference and other environmental events, etc.

- **Privacy**

- Privacy is often relegated to a second-order consideration. Privacy can in some cases be aided by appropriate technology, but many of the misues are the result of misuse by trusted insiders or are extrinsic — involving indirect misuse external to computer systems. Identity theft is an increasingly pervasive example. (For example, see [11, 104, 342].)

• **Maintainability**

- Design flaws and software bugs are ubiquitous, either lingering or emerging in upgrades. Overrreliance on patch management is just one consequence. Inadequate architectural approaches to managing complexity exacerbate the problems (such as bloatware with inordinate interdependence on untrustworthy components), as does inattention to human interfaces — particularly system and network administrators.

Many systems actually have critical requirements that span multiple areas such as security, reliability, safety, and survivability. Although the cases listed above generally result from a problem in primarily one of these areas, there are many cases in which a maliciously induced security problem could alternatively have resulted from an accidentally triggered reliability problem, or — similarly — where a reliability/availability failure could also have been triggered intentionally. (For example, see Chapter 4 of [260].)

One such application area with critical multidisciplinary requirements has become of particular interest since the 2000 November election, resulting from the emerging desire for completely electronic voting systems that ideally should have stringent requirements for system integrity, voter privacy, and accountability, and — perhaps most important — the impossibility of uncontrolled human intervention during elections. Some of today's major existing all-electronic systems permit unmonitored human intervention (to recover from election-day glitches and to "fix" problems — including during the voting and vote-counting procedures!), with no meaningful accountability. Some systems even routinely undergo code changes *after* the software has been certified! Thus, we are confronted with all-electronic paperless voting systems that have no independent audit record of what has happened in the system internals, with no real assurance that your vote was correctly recorded and counted, with no alternative recount, no systemic way of determining the presence of internal errors and fraud, and no evidence in case of protests. The design specs and code are almost always proprietary, and the system has typically been certified against very weak voluntary standards that do not adequately detect fraud and internal accidents, with evaluations that are commissioned and paid for by the vendors. In contrast, gambling machines are regulated with extreme care (for example, by the Nevada Gaming Commission), and held to extremely high standards.

For a partial enumeration of recorded cases of voting-system irregularities over the past more than twenty years, see the online html version of [267], clicking on *Election Problems,* or see the corresponding section in the .pdf and .ps versions.

Section 5.2.2 reconsiders some of the above cases as well as others in which problems arose specifically because of problems involving the human interfaces.

2.3 Trustworthiness Principles

Willpower is always more efficient than mechanical enforcement, when it works. But

there is always a size of system beyond which willpower will be inadequate.
Butler Lampson

Developing and operating complex systems and networks with critical requirements demands a different kind of thinking from that used in routine programming. We begin here by considering various sets of principles, their applicability, and their limitations.

We first consider the historically significant Saltzer–Schroeder principles, followed by several other approaches.

2.3.1 Saltzer–Schroeder Security Principles, 1975

The ten basic security principles formulated by Saltzer and Schroeder [334] in 1975 are all still relevant today, in a wide range of circumstances. In essence, these principles are summarized with a CHATS-relevant paraphrased explanation, as follows:

- **Economy of mechanism:** Seek design simplicity (wherever and to whatever extent it is effective).
- **Fail-safe defaults:** Deny accesses unless explicitly authorized (rather than permitting accesses unless explicitly denied).
- **Complete mediation:** Check every access, without exception.
- **Open design:** Do not assume that design secrecy will enhance security.
- **Separation of privileges:** Use separate privileges or even multiparty authorization (e.g., two keys) to reduce misplaced trust.
- **Least privilege:** Allocate minimal (separate) privileges according to need-to-know, need-to-modify, need-to-delete, need-to-use, and so on. The existence of overly powerful mechanisms such as *superuser* is inherently dangerous.
- **Least common mechanism:** Minimize the amount of mechanism common to more than one user and depended on by all users. Avoid sharing of trusted multipurpose mechanisms, including executables and data — in particular, minimizing the need for and use of overly powerful mechanisms such as *superuser* and FORTRAN `common`. As one example of the flaunting of this principle, exhaustion of shared resources provides a huge source of covert storage channels, whereas the natural sharing of real calendar-clock time provides a source of covert timing channels.
- **Psychological acceptability:** Strive for ease of use and operation — for example, with easily understandable and forgiving interfaces.
- **Work factor:** Make cost-to-protect commensurate with threats and expected risks.
- **Recording of compromises:** Provide nonbypassable tamperproof trails of evidence.

Remember that these are principles, not hard-and-fast rules. By no means should they be interpreted as ironclad, especially in light of some of their potential mutual contradictions that require development tradeoffs. (See Section 2.6.)

The Saltzer–Schroeder principles grew directly out of the Multics experience (e.g., [277]), discussed further at the end of this section. Each of these principles has taken on almost mythic proportions among the security elite, and to some extent buzzword cult status among many fringe parties. Therefore, perhaps it is not necessary to explain each principle in detail — although there is considerable depth of discussion underlying each principle. Careful reading of the Saltzer–Schroeder paper [334] is recommended if it is not already a part of your library. Matt Bishop's security books [47, 48] are also useful in this regard, placing the principles in a more general context. In addition, Chapter 6 of Matt Curtin's book [89] on "developing trust" — by which he might really hope to be "developing trustworthiness" — provides some useful further discussion of these principles.

There are two fundamental caveats regarding these principles. First, each principle by itself may be useful in some cases and not in others. The second is that when taken in combinations, groups of principles are not necessarily all reinforcing; indeed, they may seem to be mutually in conflict. Consequently, any sensible development must consider appropriate use of each principle in the context of the overall effort. Examples of a principle being both good and bad — as well as examples of interprinciple interference — are scattered through the following discussion. Various caveats are considered in the penultimate section.

Table 2.1 examines the applicability of each of the Saltzer–Schroeder principles to the CHATS goals of composability, trustworthiness, and assurance (particularly with respect to security, reliability, and other survivability-relevant requirements).

Table 2.1: CHATS Relevance of Saltzer–Schroeder to CHATS Goals

Principle	Composability	Trustworthiness	Assurance
Economy of mechanism	Beneficial within a sound architecture; requires proactive design effort	Vital aid to sound design; exceptions must be completely handled	Can simplify analysis
Fail-safe defaults	Some help, but not fundamental	Simplifies design, use, operation	Can simplify analysis
Complete mediation	Very beneficial with disjoint object types	Vital, but hard to achieve with no compromisibility	Can simplify analysis
Open design	Design documentation is very beneficial among multiple developers	Secrecy of design is, a bad assumption; open design requires strong system security	Assurance is mostly irrelevant in badly designed systems; open design enables open analysis (+/-)
Separation of privileges	Very beneficial if preserved by composition	Avoids many common flaws	Focuses analysis more precisely
Least privilege	Very beneficial if preserved by composition	Limits flaw effects; simplifies operation	Focuses analysis more precisely
Least common mechanism	Beneficial unless there is natural polymorphism	Finesses some common flaws	Modularizes analysis
Psychological acceptability	Could help a little — if not subvertible	Affects mostly usability and operations	Ease of use can contribute
Work factor	Relevant especially for crypto algorithms, but not their implementations; may not be composable	Misguided if system easily compromised from below, spoofed, bypassed, etc.	Gives false sense of security under nonalgorithmic compromises
Compromise recording	Not an impediment if distributed; real-time detection/response needs must be anticipated	After-the-fact, but useful	Not primary contributor

In particular, complete mediation, separation of privileges, and allocation of least privilege are enormously helpful to composability and trustworthiness. Open design can contribute significantly to composability, when subjected to internal review and external criticism. However, there is considerable debate about the importance of open design with respect to trustworthiness, with some people still clinging tenaciously to the notion that security by obscurity is sensible — despite risks of many flaws being so obvious as to be easily detected externally, even without reverse engineering. Indeed, the recent emergence of very good decompilers for C and Java, along with the likelihood of similar reverse engineering tools for other languages, both suggest that such attacks are becoming steadily more practical. Overall, the assumption of design secrecy and the supposed unavailability of source code is often not a deterrent, especially with ever-increasing skills among black-box system analysts. However, there are of course cases in which security by obscurity is unavoidable — as in the hiding of private and secret cryptographic keys, even where the cryptographic algorithms and implementations are public.

Fundamental to trustworthiness is the extent to which systems and networks can avoid being compromised by malicious or accidental human behavior and by events such as hardware malfunctions and so-called acts of God. In [264], we consider **compromise from outside, compromise from within,** and **compromise from below,** with fairly intuitive meanings. These notions appear throughout this report.

There are other cases in theory where weak links can be avoided (e.g., zero-knowledge protocols that can establish a shared key without any part of the protocol requiring secrecy), although in practice they may be undermined by compromises from below (e.g., involving trusted and supposedly trustworthy insiders subverting the underlying operating systems) or from outside (e.g., involving penetrations of the operating systems and masquerading as legitimate users). For a fascinating collection of papers on vulnerabilities and ways to exploit weak links, see Ross Anderson's website:
http://www.cl.cam.ac.uk/users/rja14/

From its beginning, the Multics development was strongly motivated by a set of principles — some of which were originally stated by Ted Glaser and Peter Neumann in the first section of the very first edition of the Multics Programmers' Manual in 1965. (See http://multicians.org.) It was also driven by extremely disciplined development. For example, with almost no exceptions, no coding effort was begun until a written specification had been approved by the Multics advisory board; also with almost no exceptions, all of the code was written in a subset of PL/I just sufficient for the initial needs of Multics, for which the first compiler (early PL, or EPL) had been developed by Doug McIlroy and Bob Morris.

In addition to the Saltzer–Schroeder principles, further insights on principles and discipline relating to Multics can be found in a paper by Fernando Corbató, Saltzer, and Charlie Clingen [85] and in Corbató's Turing lecture [84].

2.3.2 Related Principles, 1969 and Later

Another view of principled system development was given by Neumann in 1969 [255], relating to what is often dismissed as merely "motherhood" — but which in reality is both very profound and difficult to observe in practice. The motherhood principles under consideration in that paper (alternatively, you might consider them just as desirable system attributes) included automatedness, availability, convenience, debuggability, documentedness, efficiency, evolvability, flexibility, forgivingness, generality, maintainability, modularity, monitorability, portability, reliability, simplicity, and uniformity. Some of those attributes indirectly affect security and trustworthiness, whereas others affect the acceptability, utility, and future life of the systems in question. Considerable discussion in [255] was also devoted to (1) the risks of local optimization and the need for a more global awareness of less obvious downstream costs of development (e.g., writing code for bad — or nonexistent — specifications, and having to debug really bad code), operation, and maintenance (see Section 7.1 of this report); and (2) the benefits of higher-level implementation languages (which prior to Multics were rarely used for the development of operating systems [84, 85]).

In later work and more recently in [264], Neumann considered some extensions of the Saltzer–

Schroeder principles. Although most of those principles might seem more or less obvious, they are of course full of interpretations and hidden issues. We summarize an extended set of principles here, particularly as they might be interpreted in the CHATS context.

- **Sound architecture.** Recognizing that it is much better to avoid design errors early than to attempt to fix them later, the importance of architectures inherently capable of evolvable, maintainable, robust implementations is enormous — even in an open-source environment. The value of a well-thought-out architecture is considerable in open-source systems. The value in closed-source proprietary systems could also be significant, if it were thought through early on, although architectural foresight is often impeded by legacy compatibility requirements that tend to lock system evolution into inflexible architectures. Good interface design is as fundamental to good architectures as is their structure. Both the architectural structure and the architectural interfaces (particularly the visible interfaces, but also some of the internal interfaces that must be interoperable) benefit from careful early specification.

- **Minimization of what must be trustworthy.** Appropriate trustworthiness should be situated where it is most needed, suitable to overall system requirements, rather than required uniformly across widely distributed components (with potentially many weak links) or totally centralized (with creation of a single weak link and forgetting other vulnerabilities). Trustworthiness is expensive to implement and to ensure, and as a consequence significant benefits can result from minimizing what has to be trustworthy. This principle can contribute notably to sound architectures. In combination with economy of mechanism, this suggests avoidance of bloatware and unfortunate dependence on less trustworthy components.

- **Abstraction.** The primitives at any given logical or physical layer should be relevant to the functions and properties of the objects at that layer, and should mask lower-layer detail where possible. Ideally, the specification of a given abstraction should be in terms of objects meaningful at that layer, rather than requiring lower-layer (e.g., machine dependent) concepts. Abstractions at one layer can be related to the abstractions at other layers in a variety of ways, thus simplifying the abstractions at each layer rather than collapsing different abstractions into a more complex single layer. Horizontal and vertical abstractions are considered in Section 3.3. Six types of abstraction are discussed in Section B.4.1.

- **Encapsulation.** Details that are relevant to a particular abstraction should be local to that abstraction and subsequently isolated within the implementation of that abstraction and the lower layers on which the implementation depends. One example of encapsulation involves information hiding — for example, keeping internal state information hidden from the visible interfaces. Another example involves masking the idiosyncrasies of physical devices from higher-layer system interfaces. and of course from the user interfaces as well.

- **Modularity.** Modularity relates to the characteristic of system structures in which different entities (modules) can be relatively loosely coupled and combined to satisfy overall system requirements, whereby a module could be modified or replaced as long as the new version satisfies the given interface specification. In general, modularity is most effective when the modules reflect specific abstractions and provide encapsulation within each module. See Section B.2 for an extensive discussion of modularity.

- **Layered and distributed protection.** Protection (and generally defensive design for security, reliability, and so on) should be distributed to where it is most needed, and should reflect the semantics of the objects being protected. With respect to the reality of implementations that transit entities of different trustworthiness, layers of protection are vastly preferable to flat concepts such as single signon (that is, where only a single authentication is required). With respect to psychological acceptability, single signon has enormous appeal; however, it can leave enormous security vulnerabilities as a result of compromise from outside, from within, or from below, in both distributed and layered environments. Thus, with respect to the apparent user simplicity provided by single signon, psychological acceptability conflicts with other principles, such as complete mediation, separation of privileges, and least common privilege. (Of particular interest here are work in distributed system protection and digital certificates such as begun by SDSI/SPKI, and continuing through recent work on digital rights management — e.g., [70, 71, 72, 151, 374] — all which are relevant in Chapter 4.)
- **Constrained dependency.** Improperly guarded dependencies (see Section 3.4) on less trustworthy entities should be avoided. However, it is possible in some cases to surmount the relative untrustworthiness of mechanisms on which certain functionality depends — as in the types of trustworthiness-enhancing mechanisms enumerated in Section 3.5. In essence, do not trust anything on which you must depend — unless you are seriously satisfied with demonstrations of its trustworthiness.
- **Object orientation.** The OO paradigm bundles together abstraction, encapsulation, modularity of state information, inheritance (subclasses inheriting the attributes of their parent classes — e.g., for functionality and for protection), and subtype polymorphism (subtype safety despite the possibility of application to objects of different types). This paradigm facilitates programming generality and software reusability, and if properly used can enhance software development. This is a contentious topic, in that most of the OO methodologies and languages are somewhat sloppy with respect to inheritance. (Jim Horning notes that the only OO language he knows that takes inheritance of specifications seriously was the DEC/ESL OWL/Trellis, which was a descendant of CLU.)
- **Separation of policy and mechanism.** Statements of policy should avoid inclusion of implementation-specific details. Furthermore, mechanisms should be policy-neutral where that is advantageous in achieving functional generality. However, this principle must never be used in the absence of understanding about the range of policies that might be usefully implemented. There is a temptation to avoid defining meaningful policies, deferring them until later in the development — and then discovering that the desired policies cannot be realized with the given mechanisms. This is a characteristic chicken-and-egg problem with abstraction.
- **Separation of duties.** In relation to separation of privileges, separate classes of duties of users and computational entities should be identified, so that distinct system roles can be assigned accordingly. Distinct duties should be treated distinctly, as in system administrators, system programmers, and unprivileged users.
- **Separation of roles.** Concerning separation of privileges, the roles recognized by protection mechanisms should correspond in some readily understandable way to the various duties. For

example, a single all-powerful superuser role is intrinsically in violation of separation of duties, separation of roles, separation of privilege, and separation of domains. The separation of would-be superuser functions into separate roles as in Trusted Xenix is a good example of desirable separation. Once again (as with single signon, noted above), there is a conflict between principles: the monolithic superuser mechanism provides economy of mechanism, but violates other principles. In practice, all-powerful mechanisms are sometimes unavoidable, and sometimes even desirable despite the negative consequences (particularly if confined to a secure subenvironment). However, they should be avoided wherever possible.

- **Separation of domains.** Concerning separation of privileges, domains should be able to enforce separate roles. For example, a single all-powerful superuser mechanism is inherently unwise, and is in conflict with the notion of separation of privileges. However, separation of privileges is difficult to implement if there is inadequate separation of domains. Separation of domains can help enforce separation of privilege, but can also provide functional separation as in the Multics ring structure, a kernelized operating system with a carefully designed kernel, or a capability-based architecture.

- **Sound authentication.** Authentication is a pervasive problem. Nonbypassable authentication should be applicable to users, processes, procedures, and in general to any active entity or object. Authentication relates to evidence that the identity of an entity is genuine, that procedure arguments are legitimate, that types are properly matched when strong typing is to be invoked, and other similar aspects.

- **Sound authorization and access control.** Authorizations must be correctly and appropriately allocated, and nonsubvertible (although they are likely to assume that the identities of all entities and objects involved have been properly authenticated — see sound authentication, above). Crude all-or-nothing authorizations are often riskful (particularly with respect to insider misuse and programming flaws). In applications for which user-group-world authorizations are inadequate, access-control lists and role-based authorizations may be preferable. Finer-grained access controls may be desirable in some cases, such as capability-based addressing and field-based database protection. However, knowing who has access to what at any given time should be relatively easy to determine.

- **Administrative controllability.** The facilities by which systems and networks are administered must be well designed, understandable, well documented, and sufficiently easy to use without inordinate risks.

- **Comprehensive accountability.** Well-designed and carefully implemented facilities are essential for comprehensive monitoring, auditing, interpretation, and automated response (as appropriate). Serious security and privacy issues must be addressed relating to the overall accountability processes and audit data.

Table 2.2 summarizes the utility of the extended-set principles with respect to the three goals of the CHATS program acronym, as in Table 2.1.

Table 2.2: CHATS Relevance of Extended-Set Principles to CHATS Goals

Principle	Composability	Trustworthiness	Assurance
Sound architecture	Can considerably facilitate composition	Can greatly increase trustworthiness	Can increase assurance of design and simplify implementation analysis
Minimization of trustworthiness	Beneficial, but not fundamental	Very beneficial with sound architecture	Simplifies design and implementation analysis
Abstraction	Very beneficial with suitable independence	Very beneficial if composable	Simplifies analysis by decoupling it
Encapsulation	Very beneficial if properly done, enhances integration	Very beneficial if composable, avoids certain types of bugs	Localizes analysis to abstractions and their interactions
Modularity	Very beneficial if interfaces and specifications well defined	Very beneficial if well specified; overmodularization impairs performance	Simplifies analysis by decoupling it and if modules are well specified
Layered protection	Very beneficial, but may impair performance	Very beneficial if noncompromisible from above/within/below	Structures analysis according to layers and their interactions
Robust dependency	Beneficial: can avoid compositional conflicts	Beneficial: can obviate design flaws based on misplaced trust	Robust architectural structure simplifies analysis
Object orientation	Beneficial, but labor-intensive; can be inefficient	Can be beneficial, but complicates coding and debugging	Can simplify analysis of design, possibly implementation also
Separation of policy/mechanism	Beneficial, but both must compose	Increases flexibility and evolution	Simplifies analysis
Separation of duties	Helpful indirectly as a precursor	Beneficial if well defined	Can simplify analysis if well defined
Separation of roles	Beneficial if roles nonoverlapping	Beneficial if properly enforced	Partitions analysis of design and operation
Separation of domains	Can simplify composition and reduce side effects	Allows finer-grain enforcement and self-protection	Partitions analysis of implementation and operation
Sound authentication	Helps if uniformly invoked	Huge security benefits, aids accountability	Can simplify analysis, improve assurance
Sound authorization	Helps if uniformly invoked	Controls use, aids accountability	Can simplify analysis, improve assurance
Administrative controllability	Composability helps controllability	Good architecture helps controllability	Control enhances operational assurance
Comprehensive accountability	Composability helps accountability	Beneficial for post-hoc analysis	Can provide feedback for improved assurance

At this point in our analysis, it should be no surprise that all of these principles can contribute in varying ways to security, reliability, survivability, and other -ilities. Furthermore, many of the principles and -ilities are linked. We cite just a few of the interdependencies that must be considered.

For example, authorization is of limited use without authentication, *whenever identity is important.* Similarly, authentication may be of questionable use without authorization. In some cases, authorization requires fine-grained access controls. Least privilege requires some sort of separation of roles, duties, and domains. Separation of duties is difficult to achieve if there is no separation of roles. Separation of roles, duties, and domains each must rely on a supporting architecture.

The comprehensive accountability principle is particularly intricate, as it depends critically on many other principles being invoked. For example, accountability is inherently incomplete without authentication and authorization. In many cases, monitoring may be in conflict with privacy requirements and other social considerations [101], unless extremely stringent controls are enforceable. Separation of duties and least privilege are particularly important here. All accountability procedures are subject to security attacks, and are typically prone to covert channels as well. Furthermore, the procedures themselves must be carefully monitored. Who monitors the monitors? *(Quis auditiet ipsos audites?)*

2.3.3 Principles of Secure Design (NSA, 1993)

Also of interest here is the 1993 set of principles (or perhaps metaprinciples?) of secure design [56], which emerged from an NSA ISSO INFOSEC Systems Engineering study on rules of system composition. The study was presented not as a finished effort, but rather as something that needed to stand the test of practice. Although there is some overlap with the previously noted principles, the NSA principles are enumerated here as they were originally documented. Some of these principles are equivalent to "the system should satisfy certain security requirements" — but they are nevertheless relevant. Others might sound like motherhood. Overall, they represent some collective wisdom — even if they are fairly abstract and incompletely defined.

- The security engineering of a system must not be done independently from the total engineering of the system.
- A system without requirements cannot fail; it merely presents surprises [386].
- The system is for the users and not the system designers.
- A systems seldom fully satisfies all of its requirements.
- Many failures of a system to meet its overall requirements are often obvious. However, failures to meet security requirements are often not obvious.
- In an operational system, it is the users' mission and information that are at risk, not the developers' or evaluators' information. The accreditor accepts those risks when deciding to use a system operationally.
- It is only in the context of a system and a security policy that the "security characteristics" of a component can be defined and evaluated.

- Every component in a system must operate in an environment that is a subset of its specified environment; [in particular,] every component in a system must operate in a security environment that is a subset of its specified security environment. (A component should not be asked to respond to events for which it was not designed — and evaluated.) [This is a gross oversimplification, particularly for systems relying on other components on the Internet.]
- Security is a system problem.
- Keep it simple to make it secure.
- There is no security in uncertainty. [This needs to be reconsidered. Random keys — or at least keys that have the appearance of randomness — are fundamental to cryptographic security. Furthermore, random sampling — testing, inspections, etc. — can be effective.]
- A system should be evaluatable and evaluated.
- Architectural analysis should not be treated lightly.
- A system is only as strong as its weakest link; the fortress walls of security should all be high enough. (Note that weak links are often not obvious.) [A system may be even weaker than its weakest link!]
- A component should protect itself from other components by adhering to the principle of mutual suspicion.
- A system should be manageable and managed.
- A system should be able to come up in a recognizably secure state.
- A system should recognize error conditions.
- Pay special attention to information flow.
- Secure systems should protect the confidentiality of user data.
- Secure systems should protect the integrity of user data.
- Secure systems should protect the reliability of user processes.

Considerable discussion of these metaprinciples is warranted. For example, "Every component in a system must operate in a security environment that is a subset of its specified environment" implies iteratively that maximum trust is required throughout design and implementation of the other components, which is a gross violation of our notion of minimization of what must be trustworthy. It would be preferable to require that each component check that the environment in which it executes is a subset of its specified environment — which is closely related to Schroeder's notion of mutual suspicion [343], noted further down the list.

"A system is only as strong as its weakest link" is generally a meaningful statement. However, some weak links may be more devastating than others, so this statement is overly simplistic. In combination with least privilege, separation of domains, and some of the other principles noted previously, the effects of a particular weak link might be contained or controlled. But then, you might say, the weak link was not really a weak link. However, to a first approximation, as we noted above, weak links should be avoided where possible, and restricted in their effects otherwise, through sound architecture and sound implementation practice.

2.3.4 Generally Accepted Systems Security Principles (I^2F, 1997)

The 1990 report of the National Research Council study group that produced *Computers at Risk* [83] included a recommendation that a serious effort be made to develop and promulgate a set of Generally Accepted Systems Security Principles (GASSP). That led to the creation of the International Information Security Foundation (I^2SF). A draft of its GASSP document [279] is available online. A successor effort is under way, after a long pause.

The proposed GASSP consists of three layers of abstraction, nine Pervasive Principles (relating to confidentiality, integrity, and availability), a set of 14 Broad Functional Principles, and a set of Detailed Principles (yet to be developed, because the largely volunteer project ran out of steam, in what Jim Horning refers to as a last gassp!). The GASSP effort thus far actually represents a very worthy beginning, and one more approach for those interested in future efforts. The top two layers of the GASSP principle hierarchy are summarized here as follows.

Pervasive Principles
- PP-1. **Accountability**
- PP-2. **Awareness**
- PP-3. **Ethics**
- PP-4. **Multidisciplinary**
- PP-5. **Proportionality**
- PP-6. **Integration**
- PP-7. **Timeliness**
- PP-8. **Assessment**
- PP-9. **Equity**

Broad Functional Principles
- BFP-1. **Information Security**
- BFP-2. **Education and Awareness**
- BFP-3. **Accountability**
- BFP-4. **Information Management**
- BFP-5. **Environmental Management**
- BFP-6. **Personnel Qualifications**
- BFP-7. **System Integrity**
- BFP-8. **Information Systems Life Cycle**
- BFP-9. **Access Control**
- BFP-10. **Operational Continuity and Contingency Planning**
- BFP-11. **Information Risk Management**
- BFP-12. **Network and Infrastructure Security**
- BFP-13. **Legal, Regulatory, and Contractual Requirements of Info Security**
- BFP-14. **Ethical Practices**

The GASSP document gives a table showing the relationships between the 14 Broad Functional

Principles and the 9 Pervasive Principles. That table is reproduced here as Table 2.3.

Table 2.3: GASSP Cross-Impact Matrix

PP:	PP-1	PP-2	PP-3	PP-4	PP-5	PP-6	PP-7	PP-8	PP-9
BFP-1	X	X	X	X	X	X	X	X	X
BFP-2	X	X	X	X					X
BFP-3	X	X	X	X					X
BFP-4	X	X		X				X	
BFP-5	X	X	X	X	X			X	
BFP-6	X	X		X					X
BFP-7	X			X	X	X	X	X	
BFP-8	X			X	X	X	X	X	
BFP-9	X			X	X	X	X	X	
BFP-10	X			X	X	X		X	
BFP-11	X	X		X	X	X	X	X	
BFP-12	X			X	X		X	X	
BFP-13	X	X	X	X					X
BFP-14		X	X	X					X

2.3.5 TCSEC, ITSEC, CTCPEC, and the Common Criteria (1985 to date)

Any enumeration of relevant principles must note the historical evolution of evaluation criteria over the past decades — from the 1985 DoD Trusted Computer System Evaluation Criteria (TCSEC, a.k.a. The Orange Book [249]) and the ensuing Rainbow Books, to the 1990 Canadian Trusted Computer Product Evaluation Criteria (CTCPEC, [64]), and the 1991 Information Technology Security Evaluation Criteria (ITSEC, [116]). These efforts have resulted in an international effort to produce the Common Criteria framework (ISO 15408 [172]), which represents the current state of the art in that particular evolutionary process. (Applicability to multilevel security is also addressed within the Common Criteria framework, although it is much more deeply embedded in the higher-assurance levels of the TCSEC.)

2.3.6 Extreme Programming, 1999

A seemingly radical approach to software development is found in the Extreme Programming (XP) movement [33]. (Its use of "XP" considerably predates Microsoft's.) Although XP appears to run counter to most conventional programming practices, it is indeed highly disciplined. XP might be thought of as very small chief programmer teams somewhat in the spirit of a Harlan Mills' Clean-Room approach, although it has no traces of formalism and is termed a *lightweight methodology.* It involves considerable emphasis on disciplined planning throughout (documented user stories, scheduling of relatively frequent small releases, extensive iteration planning, and quickly fixing XP whenever necessary), designing and redesigning throughout (with simplicity as a driving force, the selection of a system metaphor, and continual iteration), coding and recoding as needed (paired programmers working closely together, continual close coordination with the customer, adherence to agreed-upon standards, only one programmer pair may integrate at one time, frequent integration, deferred optimization, and no overtime pay), and testing repeatedly throughout (code must pass unit tests before release, tests must be created for each bug found, acceptance tests are run often, and the results are published).

In essence, Extreme Programming seeks to have something running at the end of each period (e.g., each week) by deferring less important concepts until later. There is a stated desire to let business folks decide which features to implement, based on the experience with the ongoing development.

Questions of how to address architecture in the large seem not to be adequately addressed within Extreme Programming (although these questions are absolutely fundamental to the approach that we are taking in this report, but perhaps are considered extraneous to XP). The concept of deferring architectural design until later in the process may work well in small systems (where dynamic changes tend to be relatively local), but can seriously complicate development of highly complex systems. Perhaps if coupled with principled architectures recommended here, Extreme Programming could be effective for larger development efforts. See the Web site noted in [33] for considerable background on the XP movement, including a remarkably lucid Frequently Asked Questions document contrasting XP with several other approaches (UML, RUP, CMM, Scrum, and FDD — although this is a little like comparing apples and oranges). Wikipedia also has a useful analysis of socalled *agile* or *lightweight* methodologies, with relevant references (http://en.wikipedia.org/wiki/Agile_software_development).

2.3.7 Other Approaches to Principled Development

There are too many other design and development methodologies to enumerate here, ranging from very simple to quite elaborate. In some sense, it does not matter which methodology is adopted, as long as it provides some structure and discipline, and is relatively compatible with the abilities of the particular design and development team. For example, Dick Karpinski hands out a business card containing his favorite, Tom Gilb's Project Management Rules: (1) Manage critical goals by defining direct measures and specific targets; (2) Assure accuracy and quality with systematic project document inspections; (3) Control major risks by limiting the size of each testable delivery. These are nice goals, but depend on the skills and experience of the developers — with only subjective evaluation criteria. Harlan Mills' "Clean-Room" technology has some elements of formalism that are of interest with respect to increasing assurance, although not specifically oriented toward security. In general, *good development practice is a necessary prerequisite for trustworthy systems, as are means for evaluating that practice.*

2.4 Design and Implementation Flaws, and Their Avoidance

Nothing is as simple as we hope it will be. Jim Horning

Some characteristic sources of security flaws in system design and implementation are noted in [260], elaborating on earlier formulations and refinements (e.g., [5, 271]). There are various techniques for avoiding those flaws, including sound architectures, defensively oriented programming languages, defensively oriented compilers, better runtime environments, and generally better software engineering practice.

- **Identification and authentication.** The lack of nonspoofable identities and inter-subsystem authentication within user systems and network infrastructures presents some major obstacles to the robust networking of systems. In addition to permitting denial-of-service attacks and penetrations, it also makes it very difficult if not impossible for traceback abilities to identify the source of misuse — assuming that the misuse can be detected. The pervasive use of fixed/reusable passwords (especially those that traverse networks unencrypted or are otherwise exposed) is also a high-risk problem. Elaborate schemes for managing these passwords (such as avoiding dictionary words) ignore many of the risks. An enormous improvement can be achieved by using one-time authenticators such as cryptographic tokens, and — in certain constrained user environments — biometrics, at least within supposedly trustworthy subsystems and subnetworks. The pervasive use of unauthenticated IP addresses that are easily spoofed is another area of risk. Remote sites and remote users are frequently not properly identified and authenticated. Meaningful authentication is a precursor to the avoidance or restriction of many kinds of misuse.

- **Authorization.** Our systems and networks suffer from a serious lack of context-sensitive authorization. Monolithic access controls tend to grant all-or-nothing or extremely coarse permissions. The development and consistent use of finer-grained authorization techniques would be very helpful in enforcing separation of privilege and least privilege. In the classified world, gross levels (e.g., Top Secret, Secret, Confidential, and Unclassified) are clearly too inclusive, which is why finer-grained compartments are invoked. An excellent article on the principle of least privilege is written by David A. Wheeler [380], with specific application to the FreeBSD `jail()`, Linux Security Modules (LSM), Security-Enhanced Linux (SELinux), and Wietse Venema's Postfix mail transfer agent. This article is mandatory reading. (See also Wheeler's free online book on secure programming for Linux and Unix [379].)

- **Initialization and allocation.** Failures in the initialization of procedures, processes, and indeed stable system and network configuration management represent a large class of system flaws. Consistency checking on entry, determination of suitable availability of appropriate resources, and deletion of possible residues are examples of techniques that can provide improved initialization and allocation.

- **Finalization.** In many programming languages, the lack of graceful termination and complete deallocation is inadequately recognized as a source of flaws. For example, deletion of leftover residues from previous executions is often ignored or relegated to an initialization problem, rather than treated systematically on termination (perhaps on the grounds that it might be avoided altogether in some circumstances). In general, finalization should be symmetrically matched with initialization. Whatever is done in initialization may need to be explicitly undone or at least checked for consistent status at finalization. Programming languages that incorporate garbage collection (GC) attempt to do this implicitly, although not always perfectly. For example, note that Java's finalizers based on pointer unreachability are inherently imprecise. Various other GC-based languages have subtle finalization problems, as do non-GC-based programming languages. Also, note that WinXP zeroizes free pages in an idle loop, instead of waiting until reuse. Overall, the need for secure and robust finalization remains a research topic.

- **Runtime validation.** A large class of flaws results from inadequate runtime validation. Careful attention to techniques such as argument validation and bounds checks (especially to prevent insertion of Trojan horses such as executables added to arguments, causing buffer overflows), divide-by-zero checks, and strong typing of arguments can have enormous benefits. Brian Randell long ago suggested the benefits of moving checking closer to the operations being performed (whether in space, in time, or in layer of abstraction), to reduce the intervening infrastructure that must be trustworthy. This is also applicable to end-to-end checks and end-to-end security.

- **Consistent naming.** Aliases, pointers, links, caches, and dynamic changes without relinking, and various redundant representations all represent common sources of security vulnerabilities. Symmetric treatment of aliases, symbolic naming and dynamic linking, strong type checking, use of globally unique names, and recognition of stale caches and cache invalidation are examples of beneficial techniques.

- **Encapsulation.** Exposure of procedure and process internals may allow leakage of supposedly protected information or externally induced interference. Proper encapsulation requires a combination of system architecture, programming language design, software engineering, static checking, and dynamic checking.

- **Synchronization consistency.** Many vulnerabilities arise as a result of timing and sequencing, such as dependence on relative ordering, race conditions, synchronization, and deadlocks — in both synchronous and asynchronous contexts. Note that many of these problems arise because of sharing of state information (particularly in real time or in sequential ordering) across abstractions that otherwise seem disjoint. Atomic transactions, multiphase commits, and hierarchical locking strategies noted in Section 3.4 are examples of constructive design techniques. A classical kind of vulnerability is time-of-check to time-of-use (TOCTTOU) flaws, which result from a lack of atomicity, to which inadequate encapsulation can also be a contributing factor.

- **Adverse dependencies.** Dependence on untrustworthy programs or subsystems is another considerable source of vulnerabilities. It can emerge as a result of flawed compilers and flawed runtime library programs, as well as program bugs — including those resulting from improper program changes and upgrades, but also from Trojan horses.

- **Other logic errors.** There are also many common logic errors (such as off-by-one counting and omitted negations) that need to be avoided. Many of these arise in the design process, but some involve bad implementation.

Useful techniques for detecting some of these vulnerabilities include defensive programming language design, compiler checks, and formal methods analyzing consistency of programs with specifications. Of particular interest is the use of static checking. Such an approach may be formally based, as in the use of model checking by Hao Chen, Dave Wagner, and Drew Dean (in the MOPS system, developed in part under our CHATS project). (See Appendix A.) Alternatively, there are numerous approaches that do not use formal methods, ranging in sophistication from `lint` to `LCLint` (Evans) to Extended Static Checking (Nelson, Reino, et al., DEC/Compaq SRC). Note that ESC is completely formally based, including use of a theorem prover; indeed, it is a formal method that has some utility even in the absence of formal software specifications.

Jim Horning notes that even partial specifications increase the power of the latter two, and provide a relatively gentle way to incorporate additional formalism into development. Strong type checking and model checking tend to expose various flaws, some of which are likely to be consequential to security and reliability. For example, `purify` and similar tools are useful in catching memory leaks, array-bound violations, and related memory problems. These and other analytic techniques can be very helpful in improving design soundness and code quality — as long as they are not relied on by themselves as silver bullets.

All of the principles have some bearing on avoiding these classes of vulnerabilities. Several of these concepts in combination — notably modularity, abstraction, encapsulation, device independence where advantageous, information hiding, complete mediation, separation of policy and mechanism, separation of privilege, least privilege, and least common mechanism — are relevant to the notion of virtual interfaces and virtual machines. The basic notion of virtualization is that it can mask many of the underlying details, and makes it possible to change the implementation without changing the interface. In this respect, several of these attributes are found in the object-oriented paradigm.

Several examples of virtual mechanisms and virtualized interfaces are worth noting. Virtual memory masks physical memory locations and paging. A virtual machine masks the representation of process state information and processor multiplexing. Virtualized input-output masks device multiplexing, device dependence, formatting, and timing. Virtual multiprocessing masks the scheduling of tasks within a collection of seemingly simultaneous processes. The Multics operating system [277] provides early illustrations of virtual memory and virtual secondary storage management (with demand paging hidden from the programs), virtualized input-output (with symbolic stream names and device independence where commonalities exist), and virtual multiprogramming (with scheduling typically hidden from the programming interfaces). The GLU environment [177] is an elegant illustration of virtual multiprocessing. GLU allows programs to be distributed dynamically among different processing resources without explicitly programmed processor allocation, based on precompiling of embedded guidance in the programs.

2.5 Roles of Assurance and Formalism

> *In principle, everything should be simple.*
> *In reality, things are typically not so simple.*
> (Note: The SRI CSL Principal Scientist is evidently both a Principle Scientist and a Principled Scientist, as well as Principal Scientist. PGN)

In general, the task of providing some meaningful assurances that a system is likely to do what is expected of it can be enhanced by any techniques that simplify or narrow the analysis — for example, by increasing the discipline applied to system architecture, software design, specifications, code style, and configuration management. Most of the cited principles tend to do exactly that —

if they are applied wisely. Techniques for increasing assurance are considered in greater detail in Chapter 6, including the potential roles of formal methods.

2.6 Caveats on Applying the Principles

For every complex problem, there is a simple solution. And it's always wrong.
H.L. Mencken

As we noted above, the principles referred to here may be in conflict with one another if each is applied independently; in certain cases, the principles are not composable. In general, each principle must be applied in the context of the overall development. Ideally, greater effort might be useful to reformulate the principles to make them more readily composable, or at least to make their potential tradeoffs or incompatibilities more explicit.

There are also various potentially harmful considerations that must be considered — for example, overuse, underuse, or misapplication of these principles, and certain limitations inherent in the principles themselves. Merely paying lipservice to a principle is clearly a bad idea; principles must be sensibly applied to the extent that they are appropriate to the given purpose. Similarly, all of the criteria-based methodologies have many systemic limitations (e.g., [257, 372]); for example, formulaic application of evaluation criteria is always subject to incompleteness and misinterpretation of requirements, oversimplification in analysis, and sloppy evaluations. However, when carefully applied, such methodologies can be useful and add discipline to the development process. Thus, we stress here the importance of fully understanding the given requirements and of creating an overall architecture that is appropriate for realizing those requirements, before trying to conduct any assessments of compliance with principles or criteria. And then, the assessments must be taken for what they are worth — just one piece of the puzzle — rather than overendowed as definitive results out of context. Overall, there is absolutely no substitute for human intelligence, experience, and foresight.

The Saltzer–Schroeder principle of keeping things simple is one of the most popular and commonly cited. However, it can be extremely misleading when espoused (as it commonly is) in reference to systems with critical requirements for security, reliability, survivability, real-time performance, and high assurance — especially when all of these requirements are necessary within the same system environment. Simplicity is a very important concept in principle (in the small), but complexity is often unavoidable in practice (in the large). For example, serious attempts to achieve fault-tolerant behavior often result in roughly doubling the size of the overall subsystem or even the entire system. As a result, the principle of simplicity should really be one of managing complexity rather than trying to eliminate it, particularly where complexity is in fact inherent in the combination of requirements. Keeping things simple is indeed a conceptually wonderful principle, but often not achievable in reality. Nevertheless, *unnecessary* complexity should of course be avoided. The back-side of the Einstein quote at the beginning of Section 2.1 is indeed both profound and relevant, yet often overlooked in the overzealous quest for perceived simplicity.

An extremely effective approach to dealing with intrinsic complexity is through a combination of the principles discussed here, particularly abstraction, modularity, encapsulation, and careful hierarchical separation that architecturally does not result in serious performance penalties, well conceived virtualized interfaces that greatly facilitate implementation evolution without requiring changes to the interfaces or that enable design evolution with minimal disruption, and far-sighted optimization. In particular, hierarchical abstraction can result in relative simplicity at the interfaces of each abstraction and each layer, in relative simplicity of the interconnections, and perhaps even relative simplicity in the implementation of each module. By keeping the components and their interconnections conceptually simple, it is possible to achieve conceptual simplicity of the overall system or networks of systems despite inherent complexity. Furthermore, simplicity can sometimes be achieved through design generality, recognizing that several seemingly different problems can be solved symmetrically at the same time, rather than creating different (and perhaps incompatible) solutions. Such approaches are considered further in Chapter 4.

Note that such solutions might appear to be a violation of the principle of least common mechanism, but not when the common mechanism is fundamental — as in the use of a single uniform naming convention or the use of a uniform addressing mode that transcends different subtypes of typed objects. In general, it is riskful to have multiple procedures managing the same data structure for the same purposes. However, it can be very beneficial to separate reading from writing — as in the case of one process that updates and another process that uses the data. It can also be beneficial to reuse the same code on different data structures, although strong typing is then important.

Of considerable interest here is David Musser's notion of *Generic Programming,* or programming with concepts. His Web site defines a *concept* as "a family of abstractions that are all related by a common set of requirements. A large part of the activity of generic programming, particularly in the design of generic software components, consists of concept development — identifying sets of requirements that are general enough to be met by a large family of abstractions but still restrictive enough that programs can be written that work efficiently with all members of the family. The importance of the C++ Standard Template Library, STL, lies more in its concepts than in the actual code or the details of its interfaces." (http://www.cs.rpi.edu/ musser/gp/)

One of our primary goals in this project is to make system interfaces conceptually simple while masking complexity so that the complexities of the design process and the implementation itself can be hidden by the interfaces. This may in fact increase the complexity of the design process, the architecture, and the implementation. However, the resulting system complexity need be no greater than that required to satisfy the critical requirements such as those for security, reliability, and survivability. It is essential that tendencies toward bloatware be strongly resisted. (They seem to arise largely from the desire for bells and whistles — extra features — and fancy graphics, but also from a lack of enlightened management of program development.)

A networking example of the constructive use of highly principled hierarchical abstraction is given by the protocol layers of TCP/IP (e.g., [169]). An operating system example is given by the capability-based Provably Secure Operating System (PSOS) [120, 268, 269]) in which the functionality at each of more than a dozen layers was specified formally in only a few pages each, with at least the bottom seven layers intended to be implemented in hardware. The underlying addressing is based on a capability mechanism (layer 0) that uniformly encompasses and protects objects of arbitrary types — including files, directories, processes, and other system- and user-defined types. The PSOS design is particularly noteworthy because a single capability-based operation at layer 12 (user processes) could be executed as a single machine instruction at layer 6 (system processes), with no iterative interpretation required unless there were missing pages or unlinked files that require operating system intervention (e.g., for dynamic linking of symbolic names, à la Multics). To many people, hierarchical layering instantly brings to mind inefficiency. However, the

PSOS architecture is an example in which the hierarchical design could be implemented extremely efficiently — because of the power of the capability mechanism, strong typing, and abstraction, and its intended hardware implementation.

We note that formalism for its own sake is generally counterproductive. Formal methods are not likely to reduce the overall cost of software development, but can be helpful in decreasing the cost of software quality and assurance. They can be very effective in carefully chosen applications, such as evaluation of requirements, specifications, critical algorithms, and particularly critical code. Once again, we should be optimizing not just the cost of writing and debugging code, but rather optimizing more broadly over the life cycle.

There are many other common pitfalls that can result from the unprincipled use of principles. Blind acceptance of a set of principles without understanding their implications is clearly inappropriate. (Blind rejection of principles is also observed occasionally, particularly among people who establish firm requirements with no understanding of whether those requirements are realistically implementable — and among strong-willed developers with a serious lack of foresight.)

Lack of discipline is clearly inappropriate in design and development. For example, we have noted elsewhere [264, 265] that the open-source paradigm by itself is not likely to produce secure, reliable, survivable systems in the absence of considerable discipline throughout development, operation, and maintenance. However, with such discipline, there can be many benefits. (See also [126] on the many meanings of *open source,* as well as a Newcastle Dependable Interdisciplinary Research Collaboration (DIRC) final report [125] on dependability issues in open source, part of ongoing work.)

Any principle can typically be carried too far. For example, excessive abstraction can result in overmodularization, with enormous overhead resulting from intermodule communication and nonlocal control flow. On the other hand, conceptual abstraction through modularization that provides appropriate isolation and separation can sometimes be collapsed (e.g., for efficiency reasons) in the implementation — as long as the essential isolation and protection boundaries are not undermined. Thus, modularity should be considered where it is advantageous, but not merely for its own sake.

Application of each principle is typically somewhat context dependent, and in particular dependent on specific architectures. In general, principles should always be applied relative to the integrity of the architecture.

One of the severest risks in system development involves local optimization with respect to components or individual functions, rather than global optimization over the entire architecture, its implementation, and its operational characteristics. Radically different conclusions can be reached depending on whether or not you consider the long-term complexities and costs introduced by bad design, sloppy implementation, increased maintenance necessitated by hundreds of patches, incompatibilities between upgrades, noninteroperability among different components with or without upgrades, and general lack of foresight. Furthermore, unwise optimization (whether local or global) must not collapse abstraction boundaries that are essential for security or reliability — perhaps in the name of improved performance. As one example, real-time checks (such as bounds checks, type checking, and argument validation generally) should be kept close to the operations involved, for obvious reasons. This topic is pursued further in Sections 7.1, 7.2, and 7.3. As another example, the Risks Forum archives include several cases in which multiple alternative communication paths were specified, but were implemented in the same or parallel conduits — which were then all wiped out by a single backhoe!

Perhaps most insidious is the *a priori* lack of attention to critical requirements, such as any that might involve the motherhood attributes noted in [255] and listed above. Particularly in dealing with security, reliability, and survivability in the face of arbitrary adversities, there are few if any easy answers. But if those requirements are not dealt with from the beginning of a development, they can be extremely difficult to retrofit later. One particularly appealing survivability requirement would be that systems and networks should be able to reboot, reconfigure, and revalidate their soundness following arbitrary outages, without human intervention. That requirement has numerous architectural implications that are considered in Chapter 4.

Once again, everything should be made as simple as possible, but no simpler. Careful adherence to principles that are deemed effective is likely to help achieve that goal.

2.7 Summary

> *In theory, there is no difference between theory and practice. In practice, there is an enormous difference.* (Many variants of this concept are attributed to various people. This is a personal adaptation.)

What would be extremely desirable in our quest for trustworthy systems and networks is theory that is practical and practice that is sufficiently theoretical. Thoughtful and judiciously applied adherence to sensible principles appropriate for a particular development can greatly enhance the security, reliability, and overall survivability of the resulting systems and networks. These principles can also contribute greatly to operational interoperability, maintainability, operational flexibility, long-term evolvability, higher assurance, and many other desirable characteristics.

To illustrate some of these concepts, we have given a few examples of systems and system components whose design and implementation are strongly principled. The omission of other examples does not in any way imply that they are less relevant. We have also given some examples of just a few of the potential difficulties in trying to apply these principles.

What are generally called "best practices" are often rather lowest-common-denominator techniques that have found their way into practice, rather than what might otherwise be the *best practices* that would be useful. Furthermore, the supposedly best practices can be manhandled or womanhandled by very good programmers, and bad programming languages can still be used wisely. Unfortunately, spaghetti code is seemingly always on the menu, and bloatware tends to win out over elegance. Overall, there are no easy answers. However, having sensible system and network architectures is generally a good starting point, as discussed in Chapter 4, where we specifically consider classes of system and network architectures that are consistent with the principles noted here, and that are highly likely to be effective in fulfilling the CHATS goals. In particular, we seek to approach inherently complex problems architecturally, structuring the solutions to those problems as conceptually simple compositions of relatively simple components, with emphasis on the predictable behavior of the resulting systems and networks — which is the essence of Chapter 3.

Chapter 3

Realistic Composability

Synopsis

One of the biggest obstacles to software development — and particularly system integration — is the difficulty of predictably composing subsystems out of modules, systems out of subsystems, and networks of systems out of systems and networking technology.

In this chapter, we outline some of the obstacles to achieving facile composability as well as some of the approaches that can contribute to the development of significantly greater composability in systems with critical requirements.

3.1 Introduction

The basic challenge confronting us is to be able to develop, configure, and operate systems and networks of systems with high levels of trustworthiness with respect to critical requirements for security, reliability, fault tolerance, survivability, performance, and other behavioral criteria, without too seriously sacrificing the desired functionality. As noted in Chapter 1, both compatibility and interoperability are important. Inherently high assurance that those systems will perform dependably as expected is also extremely desirable. These attributes can be greatly aided by taking pains to constrain architectures and the software development process.

To these ends, one of the most fundamental problems involves assuring the ability to compose subsystems to form dependable systems and to compose component systems to form dependable networks — without violating the desired requirements, and without diminishing the resulting trustworthiness. Composability problems are very old, relative to the youth of the computer field. They exist throughout the life cycle, involving composability (and noncomposability) of requirements, policies, specifications, protocols, hardware subsystems, and software components (with respect to their source code, compilers, object code, and runtime libraries), as well as arising in system and network reconfiguration, upgrades, and maintenance (for example). Analogous problems also arise with respect to the compositionality of assurance measures (including formal methods and component testing) and their evaluations, and even more so to the evolution of evaluations over time as systems change. Ultimately, the degree to which composability is attainable depends strongly on the system and network architectures, but is also influenced by many other factors.

Unfortunately, many seemingly sound compositions can actually compromise the desired overall requirements, as noted in Section 3.2.

Various approaches to decomposing systems into components are examined in Section 3.3, whereas how to enhance composability is considered in Section 3.4. Of additional interest is the concept of combining subsystems in ways that can actually *increase* the resulting trustworthiness. This is explored in Sections 3.5 and 3.6, along with the relevance of concepts of software engineering discipline, programming-language constructs, structural compatibility, execution interoperability, and development tools — all of which can considerably improve the likelihood of achieving seamless composability.

We include many references here and intentionally try to balance important early efforts that deserve not to be forgotten with more recent efforts that continue toward the ultimately desired research and development results.

3.2 Obstacles to Seamless Composability

A modular system is one that falls apart easily! E.L. (Ted) Glaser, 1965

Seamless composability implies that a composition will have the desired beneficial properties, with no uncontrollable or unpredictable side effects. That is, the composed system will do exactly what it is expected to do — *no more and no less.* (*More* and *less* can both create potentially serious problems.) In practice, many pitfalls are relevant to the composition of subsystems into systems — often involving unanticipated effects (colloquially, "side effects") that impede the ideal goal of unencumbered composition and interoperability among the subsystems:

- **Inadequate requirements.** If stated requirements do not explicitly demand that subsystems and other components must be developed in ways that would encourage compatibility and interoperability, composability is likely to be much more difficult to achieve. Furthermore, poorly defined requirements are likely to hinder composability.

- **Nonexistent or inappropriate specifications.** If system and subsystem specifications do not adequately define the relationships among interfaces, inputs, internal state information and state transitions, outputs, and exception conditions, and if those specifications are oblivious to critical relationships with related functionality, determining to what extent composability is possible becomes much more difficult. Composition of underconstrained specifications is an inherent problem, because the extent to which the components compose is ill-defined; supposed demonstrations of composability may actually be meaningless. Overly constrained specifications (for example, including unnecessarily low-level and possibly incompatible details) are also often an impediment to composability. Shared state information across components is also a particular source of potential problems.

- **Properties that exist beyond what is defined by stated individual subsystem interface specifications.** Assuming the presence of meaningful specifications, inadequacies of the specifications and inconsistencies between specifications and implementations are characteristic problems. In general, specifications are always inherently incomplete with respect to defining what should *not* happen, even when they are fairly good at defining what should happen. (Abstraction is a very important technique for simplifying specifications, but it suppresses detail that may include undesirable aspects of behavior and may therefore negatively affect compositional properties.) In addition, programming languages and compilers themselves provide very few if any guarantees that something beyond what is expected cannot occur. Examples include shared-buffer interactions and unanticipated information residues from one invocation of a subsystem to a subsequent or concurrent invocation of the same subsystem; buffer overflows and other cases of inadequate bounds checks and inadequate runtime validation; inadequate authentication; improper initialization and finalization; improper encapsulation, which can result in interference and other unexpected interactions; race conditions; covert channels; and intentionally planted Trojan horses. This list represents just the tip of a huge iceberg. All these problems can impair composability. As one example, various Windows operating systems are actually relatively modular (which is essential for orderly development), but the modules are not sufficiently encapsulated to prevent adverse effects resulting from composition. Drew Dean suggests that TCP over 802.11 and TCP over TCP are also interesting examples of composition problems.

- **Properties that manifest themselves only as a result of combinations of subsystems.** Examples include adverse *emergent* properties (i.e., disruptive or even constructive effects that are not evident in any of the individual subsystems but that arise only when the subsystems are combined); adverse feedback interactions between subsystems, such as infinite loops or dependence on functionality that is less trustworthy; emergent covert channels that do not exist in any of the subsystems in isolation; mutual incompatibilities in the interfaces — perhaps resulting from internal state interference; global failure modes resulting from local faults, as in the 1980 ARPANET collapse [320] and the 1990 AT&T long-distance collapse (e.g., see [260]); socalled "man-in-the-middle" attacks (which might alternatively be called untrustworthy interpositions), in which an interposer can simulate the actions of each component; and other failure modes that arise only in the overall system context. A fascinating noncomposability situation is noted in attempts to combine encryption with digital signatures [12]: signatures are composable with public-key cryptography, but *not* with symmetric cryptography, in which case security may break down. These impediments to composability can arise essentially everywhere throughout the development life cycle — for example, incompatibilities among different requirements and policies, undesirable interactions in specifications and implementations, and difficulties in reconfiguration and maintenance. Composability of cryptographic libraries useful for automated proofs have been considered by IBM Rüschlikon [27, 28].

- **Multivendor and multiteam incompatibilities.** In the interests of having heterogeneous that can mix and match alternative components, it would be desirable to use multiple system developers. However, incompatibilities among interface assumptions, the existence of proprietary internal and external interfaces, and extreme performance degradations resulting from the inability to optimize across components can result in the inability to compose the components.

- **Scalability issues.** Composability typically creates many issues of scalability. For example, performance may degrade badly or nonpredictably as multiple subsystems are conjoined.

Ideally, composability can result in a wide range of expected performance implications — for example, linear, multiplicative, or exponential in the number of composed subsystems. In practice, even further degradations can result — for example, from design or implementation flaws or indirect effects of the composition, such as unrecognized dependence on substantively slow interactions. Obviously, infinite loops and standstill deadlocks ("deadly embraces") are limiting cases of degradation, and often arise precisely as a result of composing subsystems.

- **Human issues.** The supposed "good guys" can accidentally have profoundly negative effects on composability, through poor system conception, inadequate requirements, lack of specifications that are comprehensive and accurate, bad software-engineering practice, misuse or bad choices of programming languages, badly managed development, and sloppy operational practice (for example). Insider "bad guys" can have various negative effects on the desired composability, such as installing Trojan horses during development, operation, and reconfiguration that impair interoperability and compromise security. Human activities can also directly impair enterprise interoperability [118]. Outsider "bad guys" are generally less likely to negatively affect composability externally, except as a result of penetrations (through which they become bad insiders), subversion of the development process, tampering, and denials of service.

In common usage, there is considerable confusion surrounding the relative roles of composability, intercompatibility, and interoperability (see Chapter 1). In that it is easy to conceive of examples in which composability implies neither intercompatibility nor interoperability, or in which neither intercompatibility nor interoperability implies composability, we avoid any attempts to taxonomize these three concepts. By avoiding the semantic distinctions, we focus primarily on seeking a strong sense of composability, recognizing that interoperability and intercompatibility may impose further constraints. From a practical point of view, what matters most is that the resulting composed systems and networks must satisfy their desired requirements. If that is the case, then we can simply say that the compositions satisfy whatever requirements exist for composability, interoperability, and intercompatibility.

3.3 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.")* Jim Horning [259]

Given a conceptual understanding of a set of system requirements, or even a detailed set of requirements, one of the most important architectural problems is to establish a workable structure of the system that can evolve into a successful implementation. The architectural decomposition of a network into subnetworks, a system into subsystems, or a subsystem into components, can benefit greatly from the principles enumerated in Chapter 2. In particular, modularity together with encapsulation, hierarchical layering, constructive uses of redundancy, and separation of concerns are examples of design principles that can pervasively affect the decomposability of a system design — and thereby the modular composability.

The work of Edsger Dijkstra (for example, [105, 107]) and David Parnas (for example, [281, 283, 284, 290, 295] has contributed significantly to the constructive structural decomposition of system architectures and system designs. In addition, Parnas [86, 282, 285, 287, 291, 292, 293, 294, 296, 297] provided definitive advances toward the formal specifications and analysis of real and complex systems, beginning in the early 1970s. Of particular importance is Parnas's enumeration of various notions of a *uses* b, and especially the concept of dependence [283] embodied in the relation a *depends on* b *for its correctness.* Appendix B elaborates on the *uses* relations.

Decomposition can take on several forms. **Horizontal decomposition** (modularization) is often useful at each design layer, identifying functionally distinct components at that layer. Horizontal decomposition can be achieved in various ways — for example, through coordination from higher layers, local message passing, or networked interconnections. In addition, the development process entails various temporal decompositions, such as abstraction refinement, in which the representation of a particular function, module, layer, or system interface undergoes successively increased specificity — for example, evolving from a requirements specification to a functional specification to an implementation. If any, additional functionality is added along the way, vulnerabilities may arise whenever development discipline is not maintained.

Vertical decomposition recognizes different layers of hierarchical abstraction and distinguishes them from one another. A very simple layering of abstractions (from the bottom up) might be hardware, operating system, middleware, application software, and users. Each of these conceptual layers can in turn be split into multiple layers, according to the needs of an architectural design, its implementation, and its assurance considerations.

Several important examples of vertical and horizontal system decomposition are found in Multics, the THE system, the Software Implemented Fault-Tolerant (SIFT) system, the Provably Secure Operating System (PSOS), the type-oriented protection of the Honeywell and Secure Computing Corporation lineage, multilevel secure (MLS) kernel-based architectures with trusted computing bases (TCBs), and the MLS database management system SeaView. These systems and others are considered in Chapter 4.

Ideally, it should be relatively easy to remove all unneeded software from a broadly supported general-purpose system, to achieve a minimal system configuration that is free of bloatware and its accompanying risks. (In some of the server-oriented architectures considered in Chapter 4, there is a fundamental need for highly trustworthy servers that are permitted to perform only a stark subset of the functionality of a general-purpose system, with everything else stripped out.) In practice, monolithic mass-market workstation software and conventional mainframe operating systems tend to defy, or at least greatly hinder, the subsetting of functionality. There are typically many unrecognized interdependencies — especially in the areas of device drivers and GUIs. (Somewhat intriguingly, real-time operating system developers seem to have done a much better job in realizing the benefits that can be obtained from stark subsetting, partly for reasons of optimizing performance, partly because of historical memory limitations, but perhaps mostly because of the importance of reducing per-unit hardware costs. However, their systems do not yet have adequate security for many critical applications.)

If a system has been designed to be readily composable out of its components, then it is also likely to be readily *decomposable* — either by removal of the unnecessary subsystems, or by the generation of the minimal system directly from its constituent parts. Thus, if composability is dealt with appropriately (e.g., in system design, programming language design, and compiler design), the decomposition problem can be solved as a by-product of composition. On the other hand, the decomposition problem is potentially very difficult for complex conceptual systems that are just being designed and for legacy software that was not designed to be either composable or decomposable.

And then we have a wonderful quote from Microsoft's Steve Ballmer, who said — in his 8 February 2002 deposition relating to the nine recalcitrant U.S. states — that it would be impossible to get the operating system to run properly and still meet the states' demands.

> *"That's the way good software gets designed. So if you pull out a piece, it won't run."*
> Steve Ballmer, Reuters, 4 March 2002.

(Modular, schmodular. That might be why many people consider "software engineering" to be an oxymoron. But what is missing from much mass-market software is not modularity, but rather clean abstraction and encapsulation.)

This is in contrast to a poignant e-mail quote from Cem Kaner, April 4, 2002: "The problem with installing these [...] patches is that, as lightly tested patches will, they can break one thing while fixing another. Last week I installed yet another Microsoft security patch for Win 2000, along with driver patches they recommended. As a result, my modem no longer works, my screen was screwed up until I reloaded the Dell driver, and my sound now works differently (and less well). I accepted patches for MS Office and Acrobat and now I get messages asking me to enable Word macros when I exit Word, not just when I open a document. (Given the widespread nature of Word macro viruses, I disable the feature.) It wasn't so long ago that it was common knowledge that patching systems reflects poor engineering and is risk prone. We should not be advocating a structure like this or making a standard of it."

3.4 Attaining Facile Composability

> *Ideally, we would like the development of complex hardware/software systems to be like snapping Lego pieces together! Instead, we have a situation in which each component piece can transmogrify its modular interface and its physical appearance — thereby continually disrupting the existing structure and hindering future composability. An appropriate analog would be if civil engineering were as undisciplined as software engineering.* PGN

Ideally, it should be possible to constrain hardware and software subsystems — and their compositions — so that the subsystems can be readily integrated together with predictable consequences. This goal is surprisingly difficult. However, several approaches can help improve the likelihood that composition will not create negative effects. (Note that Brad Cox achieved something like this in the mid-1980s with what he called software integrated circuits; see http://en.wikipedia.org/wiki/Software_component.)

In hardware, address relocation, segmentation, paging, multiprocessing, and coprocessors have helped. In software, virtual memory, virtual machines, distributed operating systems, modern software engineering and programming languages better enforcing the principles of good software-engineering practice, sound distributed programming, network-centered virtualized multiprocessing, and advancing compiler technology can contribute to increased composability — if they are properly used. In particular, virtual memory techniques have considerably increased the composability of both hardware and software. (It is lamentable that software engineering practice is so seldom good!)

- **Compatibility and interoperability.** As noted in Chapter 1, compatibility implies merely the ability to coexist within a common framework, whereas interoperability additionally implies the ability to work together without adverse side effects. Both are generally essential prerequisites for composability.

- **Web interoperability.** In recent years, considerable effort has been devoted toward establishing a common definition of a *Web portal* concept that would facilitate universal interoperability providing access to Web services. A recent article by Michael Alan Smith [355] proposes a hierarchical General Portal Model that attempts to unify 17 somewhat differing definitions from the literature. From the top, the layers address process interfaces (process identification, transformation), resource discovery (resource identification, resource location, resource binding), and network interfaces (security, network access). In this context, a portal implies an "infrastructure providing secure, customizable, personalizable, integrated access to dynamic content from a variety of sources, in a variety of formats, *wherever it is needed.*"

- **Consistency and completeness of the interface specifications.** It is desirable that there be no externally discernible functional behavior other than precisely what is specified, implying bilateral consistency of behavior with respect to the functional specifications. That is, the subsystem must do what it is supposed to do, *and nothing else beyond what is specified.* However, it is important to note that specifications are inherently incomplete, and that many system failures (in security, reliability, performance, and so on) result from events that occur outside of the scope of the specifications and thus are undetectable by any analyses based on those specifications.

- **Independence of specification abstractions.** As noted above, abstraction can be an enormous aid to composability of specifications, as well as to assurance proofs. However, it is essential that the details not explicitly represented by each abstraction be independent of the details of other abstractions. Otherwise, composability will most likely be impaired. One elegant example of provable composability is seen in the orthogonality theorem of Chander, Dean, and Mitchell [72], which provides soundness and completeness proofs for a trust management kernel with a clean separation between authorization and structured distributed naming.

- **Timing and synchronization issues.** In general, Lamport-style safety properties (i.e., nothing bad happens) compose better than liveness properties (something good eventually happens with certainty) [197], but this boundary is blurred by the inclusion of timing constraints, which are technically safety properties, but generally not composable. It is also blurred by the existence of properties that are neither safety nor liveness — such as information flow. Furthermore, time (whether real time or relative time) is typically common to different abstractions, which is a reason that synchronization and timing constraints can present serious impediments to facile composition. For example, see Kopetz [195] on composability in the Time-Triggered Architecture.

- **Statelessness, total state visibility, object-oriented paradigms, and information hiding.** If a subsystem is stateless (that is, it does not remember any of its own state information from one incarnation to the next), then it may be less likely to have adverse interactions when that subsystem is composed with other subsystems — although there are always issues such as noncommutativity of operations and interference during concurrent execution. In addition, nontrivial recovery, as in selective rollback, may be unnecessary. However, statelessness is often not a desirable goal — although stack disciplines can effectively separate the state information from the subsystem itself. Assuming that a subsystem is stateful (that is, it retains at least some of its own state information from one incarnation to the next), there is a choice between the classical notion of information hiding and external visibility of state information. External state visibility of each subsystem at least makes state information explicit, and also tends to make explicit any residues that might impair compositionality. On the other hand, because information hiding typically masks internal state information, it can hinder facile composability. For example, internal state information can be in conflict across the implementation of different subsystems even if invisible in the specifications — and particularly so if the subsystems share any concrete state attributes that are not separately specified and assured (as in the case of virtual machines and virtual memory). (For example, pointers, loosely bound aliases, and other indirect references tend to create problems.) Thus, the separation of common stateful entities can greatly facilitate composition. Nevertheless, information hiding is very desirable for other reasons, including isolation, security, system integrity, and tamperproofing. One interesting historical approach is found in the formal specifications of PSOS, in which certain state information is hidden but from which the state information that is explicitly visible at the module interface is derived. Because the hidden state information cannot be accessed outside of the module (information hiding), it cannot be referenced in any other module specification. As a result, there can be no module state residues or other state information that can be accessible to other modules or subsequent invocations of the same module beyond what is explicitly declared as visible. This greatly increases the composability of modules and the analysis of potential interactions. It also rules out certain characteristic flaws simply because it is impossible to include them in the specifications! This approach is discussed further in Section 6.4.

  The object-oriented paradigm (especially with meticulously strong typing) can also contribute significantly to composability, based on abstraction, encapsulation, subtype inheritance with respect to specifications, and polymorphism. Note that the seeming inheritance of implementations without strict inheritance of specification subclasses runs counter to composability. Every subclass instance must meet the specifications of all its superclasses, or else all verifications of uses of the superclasses are unsupported. In any case, any internal state (residual or otherwise) of a subsystem should be demonstrably independent of the internal states of other subsystems with which it is to be composed. In essence, the required analysis is somewhat similar to that used in multilevel-security flow analysis and noninterference, as in the work of Owicki–Gries and Goguen–Meseguer [139, 140]. A variant of ML, Objective Caml is an outstanding example of a programming language that contributes very substantially to increasing composability; this results from Objective Caml's module system and approach to object orientation, in addition to the type system, exception handling, and automatic memory management provided by ML. (See http://caml.inria.fr and its associated FAQ pages http://caml.inria.fr/FAQ/general-eng.html.) See also Bertrand Meyer's Eiffel programming by contract (http://www.inf.ethz.ch/ meyer/). There are also some similarities with programming-language analyses done by compilers and debugging tools. However, although such an independence condition would be theoretically desirable, it is also not sufficient in practice — because of the inherent incompleteness of the specifications and the presence of emergent properties. Modula 3 also has a modular system conducive to composability.

- **Policy composition.** Serious problems can result when different policies do not compose properly — especially if that lack of composability is not discovered until much later in development. Furthermore, attempting to compose policies often results in emergent properties that are not evident from the constituent policies. For example, see recent work by Virgil Gligor et al. with respect to the composability of separation-of-duty policies [135] and application-specific security policies [134]. Gligor notes (among other things) that policy composability does not necessarily imply the usefulness of the resulting policies, and that existing compositionality criteria are not always realistic. Denial of service is a particularly thorny policy; besides, policies that do not address denials of service are inherently incomplete. Of considerable interest is recent work by Heiko Mantel relating to the general composability of secure systems [220] and on flow properties that are preserved under refinement [219]. Also relevant here are earlier papers on the difficulties of composing policies for multilevel security [368] and restrictiveness [222, 223, 225] (the nondeterministic generalization of noninterference), as is other work on policy composition (e.g., [3, 36, 37, 147, 162, 218, 224, 226, 230, 238, 319, 325, 363, 388, 389]).

  Jim Horning notes that policy composition is an instance of the general problem of building axiomatic systems. Consistency is difficult to demonstrate in practice (and impossible to demonstrate in general), and surprises emerge even from the composition of a few simple axioms. For example, the axiomatization of rational numbers in terms of 0, 1, +, –, *, and / was incorrect in the published versions of the Larch Shared Language Handbook, despite its scrutiny by numerous colleagues, and despite the fact that this is an extensively studied and axiomatized domain. Every axiom seems to make sense in isolation, but their combination leads to a contradiction or otherwise unacceptable conclusion.

- **Proof composition.** An extraordinary book on compositionality of proofs [94] is worth careful reading for anyone interested in formal verification and high assurance of systems. Composability of proofs is also an attribute of the Berkeley static analysis discussed in Appendix A.

- **Certification composition.** Rushby [322] has characterized some of the main issues relating to the modular certification of aircraft derived from separate certification of its components, based on an extension of a formal verification approach. The crucial elements involve separation of assumptions and guarantees (based on "assume-guarantee reasoning") into normal and abnormal cases.

- **Protocol composition.** There is also ongoing work on protocol composability by Nancy Lynch at MIT, Dushko Pavlovic at Kestrel, John Mitchell at Stanford, and others — for example, see [93]. Several of the cited works include further references for anyone aspiring to be the Compleat Composability Maven. An interesting research challenge might be to consider a particular collection of protocols (e.g., for authentication, encryption, and integrity preservation) and prove that they are mutually composable, subject to certain constraints; the proofs could also be extended to demonstrating that their modular implementations would be composable.

- **Other past research on composition.** There was a significant flurry of activity on system composition in the early 1990s [7, 55, 203, 204], including some works specifically addressing principles of secure design [56], specification [160], and design and evaluation [365].

- **Dependency analysis.** In many systems, there are unrecognized interdependencies among different components that hinder composability. Similar comments are relevant to contradictory or otherwise incompatible interdependencies among policies, models, separately compiled software, and even proofs. Identifying such dependencies and removing them or otherwise neutralizing them would be a considerable aid to composability, The role of dependency analysis as an assurance technique is considered in Section 6.2.

- **Constrained and guarded dependency strategies.** Deterministic linearization or other suitable prioritization of intersubsystem dependencies can avoid many adverse dependency problems, such as often result from misguided locking strategies and search strategies, upgrade compatibility mismatches, and unanticipated distributed interactions. For example, in Dijkstra's THE system paper [106], the use of a linearly ordered hierarchical locking structure guaranteed that no deadly embraces could occur *between* two different layers of abstraction (although in subsequent years a deadly embrace was occasionally discovered *within* a particular layer). As another example, Ken Biba's multilevel integrity [43] (MLI) requires in essence that no computational entity (e.g., user, program, process, or data) may depend on any other entities that are deemed less trustworthy (i.e., that are potentially less highly trusted) with respect to integrity. (MLI is considered further in Section 4.3.) In the broadened sense of dependence considered here, the strict lattice ordering of multilevel integrity attributes implied by Biba may be relaxed if any relative untrustworthiness can be masked by creative system architecture or otherwise transcended. This notion of guarded acceptability of dependence on less trustworthy components is systematized in [264] (where it is referred to as *generalized dependence*). Here referred to as **guarded dependence**, this concept is explored further in the next two sections. See Abadi et al. [1] for a recent formalization of dependency.

- **Functional consistency among layers of abstraction.** The 1977 Robinson–Levitt paper [316] on hierarchical formal specifications introduced the concept of formal mappings between different layers of functional specifications that represent abstract implementations of each layer as a function of the lower layers. Formal proofs at one layer can be derived by using the mapping functions together with the formal specifications at appropriate layers. The relatively unsung Robinson–Levitt mapping analysis is actually quite far-reaching, and can be used directly to relate properties of a composed system to individual properties of its subsystems. As noted above with respect to correctness and completeness of interface specifications, this approach is of course limited by any incompleteness in the functional specifications and mapping functions. The Robinson–Levitt approach was part of the SRI Hierarchical Development Methodology (HDM) [318] used in the Provably Secure Operating System [120, 268, 269] project in the 1970s. A comparable facility exists in the interpretation mechanism of PVS, the current ongoing SRI formal verification environment, as noted further in Section 6.4. An informal application of explicit interlayer relationships is found in the analysis of the interlayer dependencies in the Honeywell/Secure Computing Corporation (SCC) LOgical Coprocessor Kernel (LOCK) [335].

- **Operating system and programming language approaches.** Program modularity, recursive and nested procedure-call protocols, clean stack disciplines, and the absence of unintended residues can all greatly enhance composability. Virtualized multiprocessing (e.g., [175, 176, 177, 178]) has considerable possibilities in enabling extremely efficient distributed processing by abstracting out many of the usual pitfalls, especially when distributed across networked systems. There is an important role for sound programming languages that naturally enforce modular separation with abstraction and encapsulation, compilers that efficiently enforce the programming-language modularity and strong typing, systems that provide efficient interprocedure and interprocess control flow, and optimizing compilers that do not throw out the baby with the bathwater (e.g., by prematurely binding entities that need to remain separated until later, creating less easily analyzed object code, seriously impeding debugging, or compromising security separations provided by architectural encapsulations and programming languages). (However, well-implemented aggressive optimizers are less likely to violate security than programmers are.) As one example, SPARK [30] (the SPADE Ada Kernel, based on the Southampton Program Analysis Development Environment) provides a language-based approach to improving security and safety. (A review of [30] is included in the Risks Digest 23(01): http://catless.ncl.ac.uk/Risks/23.01.html.) Correctness-preserving transformations that survive compilation and optimization are another approach with significant promise. In particular, optimizing compilers must be fairly farsighted not to compromise the integrity of source code in the context of its system execution, although careful modularity with abstraction and encapsulation can diminish some of those possible effects. An alternative approach to assuring the soundness of the optimization is the translation validation approach considered at NYU [391], in which a validation tool confirms that the object code produced by the optimizer is a correct translation of the source code.

- **Principled designs, implementations, and use.** The Saltzer-Schroeder principles [345] and the subsequent extensions enumerated in Section 2.3 are often cited as extremely beneficial to the attainment of security. Techniques particularly relevant to composability include abstraction, hierarchical layering, encapsulation, the object-oriented paradigm, design diversity, composability, pervasive authentication and access control as well as administrative and operational controllability, pervasive accountability and recovery, separation of policy and mechanism, assignment of least privilege, separation of concerns, separation of roles, separation of duties, and separation of domains.

- **Holistic architectures.** Historically, certain institutions — for example, SRI's Computer Science Laboratory and the University of Newcastle upon Tyne — have long stressed the importance of system architectures that *a priori* address the notion of dependability, which encompasses high-assurance approaches to security, reliability, availability, survivability, human safety, interoperability, maintainability, composability, and so on. See Chapter 4.

- **Research in predictable composition.** Sascha Romanovsky at Newcastle noted the relevance of ongoing work by Ivica Crnkovic (http://www.idt.mdh.se/ icc/) and Magnus Larsson in Sweden [88]. This work is extremely interesting and bears close watching.

The above approaches are in a sense all part of what should be commonly known as good software-engineering practice. Unfortunately, system architecture and programming development seldom observe good software-engineering practice. However, the constructive aspects of software engineering — including establishment of requirements, careful specifications, modularity and encapsulation, clean hierarchical and vertical abstraction, separation of policy and mechanism, object orientation, strong typing, adherence to the basic security principles (e.g.,separation of privileges, allocation of least privilege, least common mechanism, assumptions of open source rather than reliance on security by obscurity), suitable choice of programming language and development tools, and (above all) sensible programming practice — can all make positive contributions to composability.

The potential importance of formal methods is largely underappreciated, including formal statements of requirements and specifications, and formal demonstrations (e.g., rigorous proofs and model checking) of consistency of specifications with requirements, consistency of source code with specifications, correctness of a compiler, and so on. The formal methods community has for many years dealt with consistency between specifications and requirements, and with consistency between code and specifications, although that work is seldom applied to real systems. The specifications tend to idealize behavior by concentrating on only the obviously relevant behavioral properties. Formal approaches can provide enormous forcing functions on composability and system correctness, even if applied only in limited ways — such as model checking for certain properties relating to composability or security. They can also be extremely valuable in efforts to attain high assurance. However, because of their labor-intensive nature, they should generally be applied particularly where they can be most effective. (See Chapter 6.) Once again, architectures that minimize the extent of necessary trustworthiness are important.

The set of assumptions as to what threats must be defended against is itself almost always inherently incomplete, with respect to what might actually happen. Nominal security requirements often ignore reliability and survivability issues (for example, see [264], which seeks to address relevant requirements within a common and more comprehensive architectural framework). Even detailed security requirements often tend to ignore the effects of buffer overflows, residues, and — even more obscurely — emanations such as those exploitable by Paul Kocher's differential power analysis [192, 193] (whereby cryptographic keys can be derived from the behavior of hardware devices such as smart cards) and external interference that can result in internal state changes (or even the ability to derive cryptographic keys, as in Dan Boneh's RSA fault injection attack [54] — which resulted in a faulted version that when subtracted from the correct version allowed a linear search instead of an exponential search for the private key!). Attempting to enumerate everything that is *not* supposed to happen is almost always futile, although relatively comprehensive canonical checklists of potential threats and characteristic flaws to be avoided can be very useful to system architects and software developers. Various partial vulnerability and threat taxonomies exist (e.g., [201, 260, 271]), although a major effort would be worthwhile to define broad equivalence classes that at least give extensive coverage, and for which effective countermeasures would be available or incorporated into new system and network architectures. It is important in considering composability that all meaningful requirements and functional behaviors be adequately specified and assured.

3.5 Paradigmatic Mechanisms for Enhancing Trustworthiness

> *You can't make a silk purse out of a sow's ear.*
> But in a sense, maybe we can — in certain cases!

It is clear that the ideal goals of unencumbered composability and easy interoperability are rather abstract and potentially unrealistic in many practical applications. Indeed, much of the research on properties that compose in some sense (e.g., strict lattice-based multilevel security) is extremely narrow and not generally applicable to common real-world situations. Consequently, we seek a more realistic notion that enables us to characterize the consequences of compositions, essentially seeking to anticipate what would otherwise be unanticipated. That is, we seek a discipline of composition.

On one hand, we would like to be able to compose subsystems in such a way that the resulting system does not lose any of the positive properties of its subsystems — in some sense, a weak compositional monotonicity property in which trustworthiness cannot decrease with respect to certain attributes. (We refer to this as **nondecreasing-trustworthiness monotonicity**.) This is indeed theoretically possible if there is suitable independence or isolation among the subsystems. In the literature of multilevel security, we are familiar with an architectural abstraction hierarchy beginning with a security kernel that enforces a basic multilevel separation property, then trustworthy extensions that are privileged in certain respects, then application software that need not be trusted with respect to multilevel security, and finally user code that cannot compromise the multilevel security that is enforced by the underlying mechanisms. However, this hierarchy assumes that the kernel is absolutely nonsubvertible and nonbypassable. In the real world of conventional operating systems, such an assumption is totally unrealistic — because the underlying operating system is typically easily subverted.

On the other hand, a fundamental hope in designing and implementing systems is that it should be possible to build systems with greater trustworthiness out of less trustworthy concepts — that is, making the proverbial silk purse out of the sow's ear, as noted above in the discussion on guarded dependence in Section 3.4. This also suggests a stronger kind of monotonicity property in which trustworthiness can actually increase with respect to certain attributes under composition and layered abstraction. (We refer to this as **cumulative-trustworthiness monotonicity**.) To this end, it is in some cases possible to relax certain assumptions of noncompromisibility of the underlying mechanisms — assumptions that are absolutely essential to the nondecreasing-trustworthiness monotonicity typified by multilevel security noted in the preceding paragraph. On the *other* other hand, desire for some sort of compositional monotonicity must be tempered by the existence of emergent properties that cannot be fully characterized in terms of lower-layer properties. That is, the properties at one layer must be superseded by related but different properties at higher layers.

What is perhaps most important in this context is the ability to make the dependencies explicit rather than allowing them to be unidentified and latent.

Fundamentally, trustworthiness is multidimensional. Increasing trustworthiness with respect to one set of attributes does not necessarily imply increasing trustworthiness with respect to other attributes. For example, increasing fault tolerance may decrease security and performance; increasing security may decrease reliability, interoperability, and performance. Furthermore, emergent properties must also be considered — particularly those related to trustworthiness. Once again, we must be very explicit as to the properties under consideration, and cognizant of how those properties relate to other system properties.

Approaches to increasing trustworthiness are explored next. The following list is the outgrowth of earlier work by Neumann in *Practical Architectures for Survivable Systems and Networks* [264], which enumerates paradigmatic mechanisms by which trustworthiness and the ensuing assurance can be enhanced by horizontal compositions at the same layer of abstraction and by vertical compositions from one layer of abstraction to the next. Each of these paradigms for guarded dependence demonstrates techniques whereby trustworthiness can be enhanced above what can be expected of the constituent subsystems or transmission media. (References given in the following enumeration are suggestive, and by no means exhaustive.)

1. **Error-correcting codes.** Hamming's early paper on single-error-correcting codes [158] inspired a large body of work on error-correcting codes, with several books providing useful overviews (for example, [9, 302, 354]) of an extensive literature. Most of the advances are based on solid mathematics (abstract algebra) — as is also the case with public-key cryptographic algorithms (e.g., abstract algebra and number theory). The constructive use of redundancy can enable correct erroneous communications despite certain tolerable patterns of errors (e.g., not only single errors, but also random multiple errors, bursts of errors, or otherwise correlated patterns, as well as codes that are optimized for asymmetric errors such as 1-to-0 bit-dropping errors only or erasure errors), in block communications or even in variable-length or sequential encoding schemes, as long as any required redundancy does not cause the available channel capacity to be exceeded (following the guidance of Shannon's information theory). In addition, error-correcting coding can also be used in arithmetic operations (e.g., [274, 310]). Gilbert et al. [131] also considered the problem of detecting intentional deception. With suitable choices of redundancy and mathematically based code construction, error-detecting and error-correcting codes can permit arbitrarily reliable communications over unreliable communication media.

2. **Fault-tolerance mechanisms.** Traditional fault-tolerance algorithms and system concepts can tolerate certain specific types of hardware and software failures as a result of constructive use of redundancy [16, 87, 183, 212, 232, 247, 270, 378]. There is an extensive literature on fault-tolerance algorithms that permit systems to withstand arbitrary failures up to the maximum intended fault-tolerance design coverage, with various modes of operation such as fail-safe, fail-soft, fail-fast, and fail-secure. Indeed, many of the fault-tolerance concepts have been around for many years, as for example the 1973 report [270] that discusses the advantages of distributing appropriate techniques according to hierarchical layers of abstraction

and different functionality. However, failures beyond that coverage may result in unspecified failure modes. This in turn can be addressed by progressively invoking different fault tolerance techniques, for diagnosis, rollback, forward recovery, repair, reconfiguration, and so on. In terms of communications and processing, error-detecting codes and other forms of error detection combined with possible retransmittal, instruction retry in hardware, or other remediation, can be effective whenever it is not already too late. The early work of John von Neumann [370] and of Ed Moore and Claude Shannon [242] showed how reliable subsystems in general (von Neumann) and reliable relay circuits in particular (Moore–Shannon) can be built out of unreliable components — as long as the probability of failure of each component is not precisely one-half and as long as those probabilities are independent from one another. With suitable configurations of components (e.g., "crummy relays" in the case of the Moore–Shannon paper), high reliability can be achieved out of low-reliability components. Also relevant is the 1960 paper of Paul Baran [29] on making reliable communications despite unreliable network nodes, which was influential in the early days of the ARPANET. For a recent highly relevant work, see the Guruswami-Sudan approach to achieving a significant improvement in decoding techniques for Reed-Solomon codes [152] and a subsequent system-theoretic formulation by Kuijper and Polderman [196].

3. **Byzantine fault tolerance.** Byzantine faults are those in which no assumptions or constraints are placed on the nature of the faults locally. In contrast to conventional fault tolerance, Byzantine fault tolerance architecturally enables a system to be able to withstand Byzantine fault modes [198, 331, 337], providing successful operation despite the arbitrary and completely unpredictable behavior (maliciously or accidentally) of up to some ratio of its component subsystems (for example, k out of $3k + 1$ in various cases). Thus, with no limitations on the failure modes of individual component subsystems, Byzantine systems can perform correctly even if the up to k bad subsystems fail in optimally contrived and malicious ways. Examples in the literature include Byzantine clocks and Byzantine network-layer protocols [299].

4. **Redundancy-based total-system reliability.** SRI's Software-Implemented Fault Tolerance (SIFT) fly-by-wire avionics system from the 1970s is an early example of achieving a total-system (including application software) probability of failure of 10^{-10} per hour out of seven off-the-shelf avionics processors with probability of failure of 10^{-5} per hour [60, 63, 232, 247, 248, 377, 378]. SIFT is discussed further in Section 4.3.

5. **Self-synchronization.** Self-synchronizing techniques can result in rapid resynchronization following nontolerated errors that cause loss of synchronization, including intrinsic resynchronizability of sequentially streamed codes. Common approaches involve adding explicit framing bits. Also found in the early literature are redundant serial codes with implicit synchronization properties that are decodable only if in the correct block synchronization — as in the case of comma-free codes (block codes that can have only one correct framing boundary when strung together), and even error-correcting comma-free codes. A rather different approach uses inherent self-synchronizing properties of finite-state machines that are used to generate variable-length and sequential codes [252, 253, 254], enabling eventual resynchronization without having to add any redundancy to the codes. This approach applies to

variable-length Huffman codes [167] (as in [252, 253]) as well as Huffman-style information-lossless sequential machines [168] (as in [254]). In both of these schemes, it is typically possible to recover from arbitrarily horrible errors, after a period of time that depends on the resynchronizing properties of the generating sequential machine. Yet another example of self-stabilization is given by Dolev [110]. Cipher-block chaining (CBC) cryptographic modes are another example in which synchronization can be a serious problem.

6. **Robust synchronization algorithms and atomic transactions.** Various approaches exist for robust synchronization, including hierarchically prioritized locking strategies as in the T.H. Eindhoven THE system [106], two-phase commitments [367], stable storage abstractions [200], nonblocking atomic commitments [312], and fulfillment transactions [231] such as fair-exchange protocols guaranteeing that payment is made if and only if goods have been delivered.

7. **Alternative-computation architectural structures.** When failure in a computation can be detected, satisfactory but nonequivalent results can be achieved (with possibly degraded performance), despite failures of hardware and software components and failure modes that exceed planned fault coverage. For example, the Newcastle Recovery Blocks approach [15, 16, 166] provided recursively for explicit alternative procedures in case the primary procedures failed their acceptance tests.

8. **Alternative-routing schemes.** The early ARPANET routing protocols (e.g., [8]) introduced the notion of dynamic reconfiguration in packet-switched networks, with good performance and eventual successful communications despite major outages among intermediate nodes and disturbances in the communications media. Much earlier work at Bell Laboratories on nonblocking telephone switching networks was an intellectual precursor of this concept (although dynamic routing did not appear in telephone networks until the 1980s), and also led to the 1960s work at SRI on the butterfly design for fast Fourier transforms and other applications.

9. **Cryptographic secrecy.** Encryption can be applied in many ways — for example, to an open transmission medium [340] or to specific applications such as e-mail [390], or to a storage medium [373]. It can result in content that is arbitrarily difficult to interpret, even if the communications are intercepted or the stored data acquired. Note that cryptography and cryptographic protocols by themselves do not provide complete solutions, and are indeed subject to numerous attacks [13, 184, 192, 193, 258, 341] including subversions of the underlying operating systems.

10. **Cryptographic integrity checks.** Secret- and public-key encryption can both be used for cryptographic checksums that have a very high probability of detecting alterations to software, data, messages, and other content [171, 340], assuming no subversions of the underlying mechanisms (e.g., operating systems).

11. **Cryptographic authentication.** Public-key and secret-key encryption can both be used to verify the authenticity of the alleged identity of a user, subsystem, system, or other entity, and can greatly enhance overall security and integrity [157, 171, 340], once again assuming no subversions of the underlying mechanisms (e.g., operating systems).

12. **Fair public-key and secret-sharing cryptographic schemes and robust crypto.** Examples include [51, 236]. Various multi-key crypto schemes require different parties to cooperate via the simultaneous presentation of multiple keys — allowing cryptographically based operations to require the presence of multiple authorities for encryption, sealing, verification of authenticity, access controls, and so on. These might be called n-out-of-n schemes, where all of the n entities must participate. Closely related are multiperson access-control schemes that do not require cryptography, and two-person business procedures. Multi-agent schemes are intended to increase the trustworthiness and integrity of the resulting action, although there can be additional risks involved as in the case of potential misuses of key escrow [6]. See also recent work on self-healing key distribution with revocation [362] and multicast packet authentication [280].

13. **Threshold multi-key-cryptography schemes.** A generalization of the n-out-of-n multi-agent schemes requires the presence of a sufficient proportion of trustworthy entities — perhaps in which at least k out of n keys are required. This is applicable to conventional symmetric-key cryptography, public-key cryptography, authentication, and escrowed retrieval (sometimes euphemistically called "key recovery"). Examples include a Byzantine digital-signature system [102]; a Byzantine key-escrow system [313] that can function successfully despite the presence of some parties that may be untrustworthy or unavailable; a signature scheme that can function correctly despite the presence of malicious verifiers [300]; and Byzantine-style authentication protocols that can work properly despite the presence of some untrustworthy user workstations, compromised authentication servers, and other questionable components (see Chapter 7 of [264]).

14. **Security kernels and Trusted Computing Bases.** The so-called "trusted" computing bases (TCBs) should ideally be *trustworthy* computing bases. Constructive use of kernels and TCBs in multilevel-secure (MLS) systems can lead to nonsubvertible MLS application properties, such as the MLS database security in SeaView [100, 213, 214], which demonstrated how a multilevel-secure database management system can be implemented on top of a multilevel-secure kernel — with absolutely no requirement for multilevel-security trustworthiness in the Oracle database management system. (This is the notion of balanced assurance, which requires composability of policies and of components.) Another approach was the tagged capabilities of PSOS [120, 268, 269] (see Section 4.3), in which the hardware has only two instructions creating capabilities — creating a new capability for a new object of a particular type, and creating a restricted copy with access privileges that would be at most as powerful, but never more powerful. This design rather simply avoided the ability to manipulate capabilities in hardware and software. Distributed systems need special care, especially MLS systems (e.g., [108]).

15. **Architecturally reduced dependence on trustworthiness.** Closely related to kernelized systems in principle, but radically different in their practical implications, are architectural approaches that starkly reduce the extent to which subsystems must be trusted, or the extent to which all phases of the development process must be trusted. Instead, the focus is on certain critical properties of selected subsystems or critical stages of the development process. In many cases, trustworthiness can be judiciously isolated within centralized systems or among

distributed subsystems. In this way, the perimeters around what must be trustworthy can be reduced, which in turn reduces what is sometimes referred to as the attack surface.

Several quite different examples are worth mentioning to illustrate this concept:

- **Layered protection.** Kernels (noted above), rings of protection, and properly implemented capability-based addressing can protect themselves against compromise from above, as in the Multics operating system [91, 150, 277, 333, 344] and various capability-based architectures [117, 143, 186, 120, 268, 269].
- **MLS enforcement.** Multilevel-secure systems and networks can be designed and implemented in which critical security properties such as MLS are enforced in selected servers, but in which there is no MLS dependence in the end-user systems [264, 308].
- **PCC.** Proof-carrying code [250] can enable the detection of unexpected alterations to systems or data and thus hinder the tampering of data and programs, and resulting contamination — irrespective of where in the development process malicious code is introduced prior to the establishment of the proof obligations.
- **Proof checking.** Proof-checkers can provide assurance that theorem provers have arrived at correct proofs, without having to trust the provers. Proof-checkers tend to be orders of magnitude simpler to develop, to assure, and to use than theorem provers.
- **Independent accountability.** There is enormous contention regarding the integrity of closed-source proprietary electronic systems for casting ballots, recording votes, and determining the results of elections, particularly those systems that are self-auditing with no external assurance. These systems reflect the need for a collection of critical requirements for security (e.g., system integrity, vote integrity, vote confidentiality, voter privacy, anonymity, accountability, nondenial of service, and overall system verifiability) as well as reliability and and other -ilities. Unfortunately, existing touch-screen direct-recording self-auditing electronic voting systems provide no assurances whatsoever that the vote that is cast is identical to the vote that is subsequently recorded and counted, and no meaningful recount is possible because there is no truly independent audit trail. Rebecca Mercuri's PhD thesis [233, 234] suggests the incorporation of a voter-verified electronically readable independent hard-copy image of the ballot as cast. This relatively simple mechanism almost by itself can surmount numerous potential weak links in the electronic stages of the election process, and could thereby enable detection and prevention of many kinds of internal fraud. This is a seemingly rare example of where the highly distributed weak-link nature of security and reliability can be overcome by a relatively simple conceptual mechanism. An entirely different approach has been proposed by David Chaum [73] that has a similar result — providing a small mechanism relative to the overall voting process that provides for each voter's ability to verify that a private ballot was correctly recorded, despite the potential untrustworthiness of any front-end system for vote casting. (Chaum's approach is applicable to a variety of voting-machine types.) Mercuri's and Chaum's methods both allow far greater trustworthiness of the overall voting systems despite potential untrustworthiness of the voting machines themselves.

16. **Mutual suspicion.** The ability to operate properly despite a mutual lack of trust among various entities was explored in 1972 in Mike Schroeder's doctoral thesis [343]. There seems to have been relatively little work along those lines since. Unfortunately, in practice, implementations typically tend to implicitly assume that some or all of the participating entities are trusted, irrespective of whether they are actually trustworthy.

17. **Interposition of trustworthy intermediation.** In principle, interposing cross-domain protection mechanisms such as firewalls (e.g., [80]), guards (which are generally much simpler than firewalls — perhaps only preventing content with undesirable keywords from being disseminated; for example, see [40, 66, 307]) and proxies (which also act as trusted intermediaries) can supposedly mediate between regions of potentially unequal trustworthiness — for example, ensuring that sensitive information does not leak out and that Trojan horses and other harmful effects do not sneak in, despite the presence of untrustworthy subsystems or mutually suspicious adversaries. For example, intermediation of network connectivity can increase the trustworthiness of internal secrecy (controlling the outbound direction) and internal integrity (controlling the inbound direction). However, care must be taken not to allow unrestricted riskful traffic, such as Java- and JavaScript-enabled Web content, PostScript, ActiveX, and other executable content that might execute if there are flaws in the underlying systems or in the virtual-machine environments.

18. **Type enforcement, object-oriented domain enforcement, and advanced access-control techniques.** Architecturally integrated access controls can effectively mediate or otherwise modify the intent of certain attempted operations, depending on the execution context [120, 268, 269, 343] — for example, the confined environment of the Java Virtual Machine [146, 148] and related work on formal specification [95, 142] for the analysis of the security of such environments. Such enforcement can be implemented in a combination of hardware and system software (as in the strong type enforcement of PSOS and Secure Computing Corporation systems), programming languages, and compilers. Other forms of static analysis are of course also relevant, particularly when embedded in the compilation process (including language pre- and post-processors) and are valuable in enhancing the trustworthiness of architectures, implementations, and system operation.

19. **Integrated internal checks.** A combination of static (e.g., design-time and compile-time) and dynamic (runtime) analysis can prevent or mediate execution in questionable circumstances — for example, embedded in programming languages and compilers within the development process, and in resulting operating-system software and application programs, as in the cases of argument validation, bounds checks, strong typing and rigorous type checking, consistency checks, redundancy checks, and independent cross-checks. Static and dynamic checks can be used to significantly increase the trustworthiness of a subsystem or system, with respect to security, reliability, and performance. Bill Arbaugh's trustworthy bootload protection [18, 19] is an example of a bootload-time check.

20. **External runtime checks.** Addition of wrappers (e.g., [381]) (without modifying the source or object code of the wrapped module) can in principle enhance survivability, security, and reliability, and otherwise compensate for deficient components — such as adding a "trusted path" to an inherently untrustworthy system, enabling monitoring of otherwise unmonitorable

functionality, or providing compatibility that was not of wrapped legacy programs with other programs. However, the utility of the wrapper approach may be subverted if the wrapper does not completely encapsulate the underlying mechanisms (e.g., operating systems).

21. **Real-time analysis.** Anomaly and misuse detection to diagnose real-time threats (e.g., from insiders, outsiders, internal malfunctions, and external environmental failures) can provide rapid analyses of actual failures and potential misuses in progress. As one example of such a system for anomaly and misuse detection, the EMERALD system [208, 207, 272, 304, 306] represents the most recent results of more than two decades of research at SRI. (See http://www.csl.sri.com/intrusion for extensive background.)

22. **Real-time response.** Given the results of real-time analysis as noted above, it is possible to trigger automated or semiautomated rapid responses — including dynamic alterations of system and network configurations, carefully controlled automated software upgrades in response to detected flaws, and enforced alterations in certain user processes, based on evaluations of the perceived real-time events. (Note that the alternative-computation architectures of technique 7 and the alternative-routing schemes of technique 8 have a similar flavor, except that the alternatives tend to be more closely integrated into the architecture, rather than dynamically variable.)

This enumeration is undoubtedly not exhaustive, and is intended to be representative of a wide variety of trustworthiness-enhancing types of mechanisms. Furthermore, these techniques do not necessarily compose with one another, and may in fact interfere with one another — especially if used unwisely. On the other hand, many efforts to attain trustworthy system for security and reliability need to rely on a combination of the above techniques — as is the case with IBM's concept of autonomic systems that can continue to operate largely without system administration. (For example, see IBM's Architectural Blueprint for Autonomic Computing, http://www-3.ibm.com/autonomic/index.shtml.)

It is clear that reliability enhancement is often based on solid theoretical bases; furthermore, that enhancement is quite tangible, but only if we assume that the underlying infrastructures are themselves not compromisible — for example, as a result of security violations or uncovered hardware malfunctions. However, in cases of mechanisms for would-be security enhancement, the dependence on the assumption of noncompromisibility of the underlying infrastructures is much more obviously evident; if we are trying to create something more secure on top of something that might be totally compromisible, we are indeed trying to build sandcastles in the wet sand below the high-water mark. Thus, security enhancement may critically depend on certain measures of noncompromisibility in the underlying hardware and software on which the implementation of the enhancement mechanisms depend.

So far, little has been said here about the relevance of these techniques to open-source software. In principle, all these techniques could be applied to closed-source proprietary software as well as open-source software. However, in practice, relatively few of these techniques have found their ways into commercial products — error-correcting codes, atomic transactions, some fault tolerance, alternative routing, cryptography, and some dynamic checking are obvious examples. The opportunities are perhaps greater for appropriate techniques to be incorporated into open-source systems, although the incentives may be lacking thus far. (The relevance of open-source paradigms is considered in Section 4.5.)

Although we suggest that the above techniques can actually enhance trustworthiness through composition, there are still issues to be resolved as to the embedding of these techniques into systems in the large — for example, whether one of these trustworthiness-enhancing mechanisms might actually compose with another such mechanism. Even more important is the question of whether the underlying infrastructures can be compromised from above, within, or below — in which we have just another example of building sandcastles in the wet sand below the high-tide level.

3.6 Enhancing Trustworthiness in Real Systems

> *Bad software lives forever. Good software gets updated until it goes bad, in which form it lives forever.* Casey Schaufler

Several conclusions can be drawn from consideration of the paradigmatic approaches for enhancing trustworthiness enumerated in Section 3.5.

- **Security versus reliability.** Enhancing reliability is in many ways quite different from enhancing security, although there are also commonalities. Quantitative probabilistic assessments are straightforward and meaningful with respect to reliability in the small, although somewhat less definitive in the large — that is, when applied to entire systems. Reliability enhancements can typically make reasonably realistic assumptions about the stability of the underlying infrastructures, and derive reasonably accurate measures of the resulting reliability — as well as satisfying real-time checks that the assumptions remain valid. However, in many cases, the assumptions are inaccurate.

  On the other hand, quantitative assessments of security are usually highly suspect. The assumptions on which they are based are generally not probabilistic in nature; each of those assumptions can be vitiated by a wide variety of circumstances — including insider misuse, penetrations that exploit design flaws and code bugs, and other forms of subversion, as well as nontolerated failures in hardware and software, power failures, interference, acts of God, and squirrelcides. As a result, measures of security derived from questionable assumptions are extremely questionable. Worse yet, measures derived from apparently sensible assumptions are still questionable if any of those assumptions is violated, and the assumption that real-time checks could ensure the continued validity of those assumptions is itself questionable.

  Thus, in our efforts to enhance trustworthiness, there are some consequential differences between reliability and security that must be taken into account. We believe that some of the approaches outlined in Section 3.5 can be extremely effective when used together. In particular, a combination of architectural techniques (1 to 18) and real-time measures to ensure continued validity of reliability and security assumptions (as in techniques 18 through 22) can be most effective. Above all, security and reliability must both be considered together architecturally. Even if composability arguments might allow them to be implemented with some separability, there are tradeoffs that must be addressed in architecture and system development.

- **Achieving both security and reliability.** With respect to attaining both security and reliability at the same time, very few of the paradigmatic techniques are capable of addressing both classes of requirements at the same time. However, achieving both security and reliability is a nontrivial exercise in composition.

  Techniques 1 through 8 are aimed primarily at hardware and software reliability, although these techniques do provide some resistance to active threats. Technique 3 (Byzantine agreement) is applicable to certain modes of unreliability and malicious attacks, typically if not more than k out of $3k + 1$ subsystems are compromised. (However, note paradoxically that if we could somehow verify dynamically that at most k subsystems had been compromised, the Byzantine protocol itself would be gratuitous! But perversely, if k of the subsystems can be compromised — from below or from within, or even from outside — then it seems highly likely that more than k subsystems — or indeed all n — could be compromised, thus completely undermining the effectiveness of the Byzantine protocol.) Techniques 9 through 18 are aimed primarily at security. Technique 10 is overkill if used merely for reliability, for which hashing or cyclic redundancy checks may be adequate. In their stated forms, only techniques 18 through 22 have any significant potential for addressing both classes of requirements simultaneously.

  It is possible to develop hybrid techniques that combine several of these paradigmatic techniques in combination, such as the integrated use of encryption and error correction. In such cases, composability is once again a critical issue. For example, although cryptography and error-correcting coding might seem to be commutative in a perfect mathematical world, they are not commutative in the real world. If a message were encoded for error correction *before* being encrypted, then decryption would have to precede error correction, and any errors in the transmission could then result in errors in decrypted text that would produce totally erroneous error correction. Even more important is the reality that error correction before encryption adds redundancy, and thereby increases the opportunities for cryptographic attacks. On the other hand, if a message were encoded for error correction *after* being encrypted, then decryption would follow error correction, and the decrypted message would be correct if the transmission errors did not exceed the coverage of the error correction; however, if the errors exceeded the coverage, the error-corrected encrypted text would be in error, and would result in disruptions to the decryption — potentially long-lasting in the case of certain cipher block chaining. Unfortunately, the use of sequential (re)synchronization techniques (as in techniques 5 and 6 above) could further muddy the waters. In practice, it seems advisable to compress first (to reduce the redundancy), then encrypt, and then provide error correction!

  In fact, the general situation with compositions involving cryptography is quite complex. For example, the composition of two sound cryptographic techniques can make the resulting system susceptible to cracking. Similarly, the composition of a cryptographic algorithm with a random-number generation scheme seems to be a fertile area for bad implementations. Furthermore, embedding good cryptographic implementations into weak systems is clearly riskful.

- **Survivability.** With respect to attaining high availability and survivability in the face of realistic adversities, security and reliability are both necessary requirements. Neumann's ARL report [264] on architectures for survivability is extremely relevant to the development and operation of systems and networks that must be highly available and dependably survivable. The reader interested in attaining survivable systems and networks is encouraged to study that work, rather than our having to repeat much of it here.

- **Multilevel security and multilevel integrity.** With respect just to the various aspects of security, confidentiality is quite different from system integrity and data integrity. This is illustrated nicely by the concepts of mandatory security and integrity, considered in greater detail in Chapter 4. Bell & LaPadula's multilevel security [35] (basically a set of confidentiality properties) and Biba's multilevel integrity [43] are formal duals (e.g., see [264]). Simply stated, the former demands that there be no adverse information flow for confidentiality, whereas the latter demands that there be no adverse control flow for integrity. The former implies that information never leaks to anything that is less trusted for confidentiality (to a lower secrecy level), whereas the latter implies that no computational entity ever depends on anything that is less trusted for integrity (at a lower integrity level). There are various ways of viewing the formal dual. Intuitively, a similar dual exists between conventional (discretionary) confidentiality and conventional forms of integrity. For example, the spread of integrity-defeating e-mail viruses is similar to the potential for unrestricted dissemination of sensitive information. In the case of confidentiality, once the cat is out of the bag, the information may have already escaped in unknown ways — and once it is outside of the local system purview, all bets may be off. It may be possible to identify suspicious user authentication and suspicious augmentation of privileges, but it may be too late to stop the undesired loss of confidentiality. In the case of integrity, the failure of a given integrity check (e.g., a cryptographic checksum on software or on data) is immediate grounds for suspicion of tampering, accidental corruption, or other contamination. By incorporating and enforcing integrity checks on subsystems, systems, data, and especially applications, damage can potentially be prevented, blocked, or otherwise confined before it can occur — once again assuming that those mechanisms are not compromised. An example of an end-to-end approach to security and integrity among collaborating entities is found in the SRI Enclaves work of Li Gong [144], subsequently reimplemented in Java [191], and then extended to provide a Byzantine form of intrusion-tolerant robustness [111].

- **Layered applicability.** Each of the above approaches is potentially applicable at various different layers of abstraction, upward from hardware to microcode to kernels to operating systems to middleware to application packages to user software. Each concept may have different interpretations at each hierarchical layer, with different types of objects, different access control rules and different notions of type enforcement, different protective measures for reliability, confidentiality, data integrity, system integrity, availability, ultimately culminating in properties such as survivability and human safety with respect to the overall systems and networks. However, each approach will have its own associated costs at each layer of application, and undisciplined use is likely to result in multiplicative escalation of overhead. Where security and reliability are critical, it may be worth the effort. Incidentally, although much effort is typically devoted to operating systems, specification and evaluation of properties of application software gets short shrift. The Robinson–Levitt type of analysis noted above is applicable to characterizing the specifications and properties of each hierarchical layer, and relating them one to another.

- **Importance of architectural approaches.** Approaches that appear to have great promise must be considered within the framework of the overall system/network architecture. Efforts to concentrate security and reliability in application software can often be undermined by operating-system compromises from below. Efforts to use operating-system enforcement may be compromised from outside and from within — or even from below, in the hardware. The sense of integrity derived from proofs of source-code satisfying requirements and proofs of compiler correctness can be circumvented by Trojan horses inserted into the object code of the compiler, as so elegantly demonstrated by Ken Thompson [364], and revisited more formally by Wolfgang Goerigk [138]. Proof-carrying code can be subverted by compromising the proof-checker (for example). In turn, each of the above techniques for potentially increasing trustworthiness must be looked on as a potential increase in complexity, and therefore as an opportunity for new design flaws, new implementation bugs, new operational hazards, and new modes of failure. Indeed, some of the techniques themselves can potentially create platforms that act as high-value weak-link targets for adversaries, or platforms from which new types of attacks can be launched. Thus, poorly designed defensive measures can actually result in increasing the relative vulnerability of the system.

- **Risks of automated responses.** One of the potentially most riskful techniques in Section 3.5 is the final technique (22), particularly as it relates to automated responses that involve reconfigurations or remotely inserted upgrades. Ideally, automated responses to perceived security and reliability threats could provide the adaptive ability to surmount, or at least recover from, arbitrarily bad events. However, slightly misconceived responses are very likely to result in increased rather than decreased chaos. Furthermore, maliciously induced automated upgrades provide a fertile opportunity for destructive denials of service. Because trying to control the actions of possible responses itself entails high risks, this approach might best be used with human intervention in all but the most clear-cut cases. However, in that people are demonstrably a major contributor — if not the most prevalent cause — of security violations and unreliability, human intervention should always be grounds for suspicion.

  Overall, it seems likely that generic approaches for automated response may not compose easily with other techniques; they run the risk of overwriting critical data or interfering with critical software, failing to interoperate because of versioning problems, or overreacting when only a simple remediation is required. For example, shutting down the ARPANET to MILNET connection in an attempt to block the 1988 Internet Worm resulted in the MILNET not receiving the information provided by the Worm's creator on how to defuse it. Similar self-inflicted denials of service have also occurred on various occasions since then. Furthermore, platform-dependent effects must be considered, such as the need to reboot in a Microsoft environment because of the interactions with DLLs — as opposed to the greater flexibility of shared libraries and resourcing in Unix/Linux systems. Consequently, efforts to incorporate such approaches are likely to need special care.

- **Humility.** Above all, considerable humility is required in any efforts to design, implement, and operate systems and networks with stringent requirements for security, reliability, and guaranteed performance levels. Anyone who believes that a perfect or even completely sufficient solution has been achieved is suspect. Indeed, it may be advisable to include in any effort a few resident skeptics and borderline paranoids with obsessions regarding the possibilities of errors and user malice. Above all, great discipline is needed throughout design, development, operation, maintenance, and management.

In general, the above discussion illustrates that composition — of different specifications, policies, subsystems, techniques, and so on — must be done with great care. We have begun to characterize some of the pitfalls and some of the approaches that might result in greater compositional predictability. However, the problem is deceptively open ended.

3.7 Challenges

The components that are cheapest, lightest, and most reliable are the ones that are not there. Gordon Bell

Efforts to achieve much greater composability present many opportunities for future work.

- Exploring the theoretical limits and practical implications of policy composition.
- Establishing realistic policy composability criteria, and formalizing them to make them amenable to formal analysis.
- Enhancing programming languages, and constraining program writing and compiler usage with effective preprocessors and analyzers.
- Developing a suite of software engineering tools that would greatly increase composability.
- Establishing some illustrative applications demonstrating effective system compositions, such as an implementation of the Enclaves Java code that would compose nicely with a wide range of cryptography and networking protocols.
- Developing easily subsettable secure operating systems that could be tailored effectively to special-purpose applications such as critical servers and real-time control. Better yet would be the ability to develop minimal systems that could be configured by composing just the needed components. (See Chapter 4 for an elaboration of that approach. As observed in Section 3.3, real-time system developers have increasingly been working toward this goal, although the resulting systems are still lacking with respect to security.)

3.8 Summary

This chapter outlines various techniques for enhancing compositionality, and for enhancing the resulting trustworthiness that can be achieved by various forms of compositions. Interoperable composability is a pervasive problem whose successful achievement depends on many factors throughout the entire life cycle. It clearly requires much more consistent future efforts in system design, language design, system development, system configuration, and system administration. It requires a highly disciplined development process that intelligently uses sound principles, and it cries out for the use of good software engineering practice. Sound system architecture can also play a huge role. Designing for composability throughout can also have significant payoffs in simplifying system integration, maintenance, and operations. However, seamless composability may be too much to expect in the short term. In the absence of a major cultural revolution in the software development communities, perhaps we must begin by establishing techniques and processes that can provide composability sufficient to meet the most fundamental trustworthiness requirements.

Overall, we believe that the approaches outlined here can have significant potential benefits, in commercial software developments as well as in open-source software — in which the specifications and code are available for scrutiny and evolution and in which collaborations among different developers can benefit directly from the resulting composability. Ultimately, however, there is no substitute for intelligent, experienced, farsighted developers who anticipate the pitfalls and are able to surmount them.

Chapter 4

Principled Composable Trustworthy Architectures

Synopsis

> *Virtue is praised, but is left to starve.* Juvenal, *Satires,* i.74. (Note: The original Latin is *Probitas laudatur et alget;* “probitas” (probity) is literally rendered as “adherence to the highest principles and ideals”.)

Many system developments have stumbled from the outset because of the lack of a well-defined set of requirements, the lack of a well-conceived and well-defined flexible composable architecture that is well suited to satisfy the hoped-for requirements, the lack of adherence to principles, and the lack of a development approach that could evolve along with changing technologies and increased understanding of the intended system uses.

In this chapter, we draw on the principles of Chapter 2 and the desire for predictable composability discussed in Chapter 3. we consider attributes of highly principled composable architectures suitable for system and network developments, appropriately addressing composability, trustworthiness, and assurance within the context of the CHATS program goals.

4.1 Introduction

> *It ain’t gonna be trustworthy if it don’t have a sensible architecture.*
> (With kudos to Yogi Berra’s good sense of large systems)

The following goals are appropriate for truswtworthy architectures.

- **Predictable composability.** It is highly desirable that systems and networks be conveniently developed out of subsystems and subnetworks that by themselves have certain desirable properties and that when combined can contribute constructively and predictably to the satisfaction of the overall requirements. As noted in Section 3.2, composability is a concept that is meaningful with respect to requirements, policies, specifications, designs, protocols, and implementations, among others. Seamless composability implies that a composition will have the desired beneficial properties, with no uncontrollable or unpredictable side effects. That is, the composed system should do exactly what it is expected to — no more, and no less. (An early software-engineered system approach to networked systems is found in [256], which may be of some historical relevance to this report, but probably little practical value today.)

- **Trustworthiness.** In the present context, the concept of trustworthiness is meaningful only with respect to a set of critical requirements whose continued satisfaction is necessary for success, under anticipated operating conditions. Trustworthiness might typically encompass attributes of security, reliability, survivability, real-time performance, and other vital attributes. In critical systems, failure to meet the trustworthiness requirements can result in serious consequences. Linguistically, trustworthiness means worthy of being trusted to do what it is supposed to do (and nothing else), with some level of assurance.

- **Assurance.** In the present context, assurance provides some measures of confidence that the requirements (and in particular, the trustworthiness requirements) will be satisfied by an architecture and its implementation. The properties of an overall system should ideally be largely derivable from the properties of the subsystems, in correspondence with the nature of the compositions. Measures of assurance may range (for example) from hand-waving (proof by emphatic assertion), through testing and pervasive red-teaming, to the extensive use of formal methods for verification or model checking of properties of requirements, policies, architectural structures, module and system specifications, protocols, implementation, and real-time configuration management. Reasoning about system upgrades and code patches is also relevant. (See an earlier report [259] on what formal methods can do for secure system architecture, some conclusions from which are reprised here.)

Thus, we seek principled composable architectures that can satisfy the trustworthiness goals, with some meaningful assurance that the resulting systems will behave as expected.

4.2 Realistic Application of Principles

A system is not likely to be trustworthy if its development and operation are not based on well-defined expectations and sound principles.

We next examine combinations of the principles discussed in Chapter 2 that can be most effective in establishing robust architectures and trustworthy implementations, and consider some priorities among the different principles.

From the perspective of achieving a sound overall system architecture, the principle of minimizing what must be trustworthy (Section 2.3) should certainly be considered as a potential driving force. Security issues are inherently widespread, especially in distributed systems. We are confronted with potential questions of trustworthiness relating to processing, multiple processors, primary memory and secondary storage, backup and recovery mechanisms, communications within and across different systems, power supplies, local operating environments, and network communication facilities — including the public carriers, private networks, wireless, optical, and so on. Whereas different media have differing vulnerabilities and threats, a trustworthy architecture must recognize those differences and accommodate them.

As noted in Section 2.6, systems and networks should be able to reboot or reconstruct, reconfigure, and revalidate their soundness following arbitrary outages without violating the trustworthiness requirements — and, insofar as possible, without human intervention. For example, automated and semiautomated recovery have long been a goal of telephone network switches. In the early Electronic Switching Systems, an elaborate diagnostic dictionary enabled rapid human-aided recovery; the goal of automated recovery has been realistically approached primarily only in the previous two decades. The Plan 9 directory structure provides an interesting example of the ability to restore a local file system to its exact state as of any particular specified time — essentially a virtualized rollback to any desired file-system state. In addition, two recent efforts are particularly noteworthy: the IBM Enterprise Workload Manager, and the Recovery-Oriented Computing (ROC) project of David Patterson and John Hennessy (as an outgrowth of their earlier work on computer architectures — e.g., [161, 298]).

There are of course serious risks that the desired autonomous operation may fail to restore a sound local system, distributed system, or network state — perhaps because the design had not anticipated the particular failure mode that had resulted in a configuration that was beyond repair. Developing systems for autonomous operation thus seriously raises the ante on the critical importance of system architecture, development methodology, and operational practice.

This if course works for malware as well! For example, Brian Randell forwarded an observation from Peter Ryan, who noted that the Lazarus virus places two small files into the memory of any machine that it infects. If either one of these files is manually deleted, its partner will resurrect the missing file (ergo, the symbolism of rising from the dead). Ryan added, "Now there's fault-tolerance and resilience through redundancy and self-healing (and autonomic design!?)!"

A system in which essentially everything needs to be trusted (whether it is trustworthy or not) is inherently less likely to satisfy stringent requirements; it is also more difficult to analyze, and less likely to have any significant assurance. With respect to security requirements, we see in Section 4.3 that trustworthiness concerns for integrity, confidentiality, guaranteed availability, and so on, may differ from one subsystem to another, and even within different functions in the same subsystem. Similarly, with respect to reliability and survivability requirements, the trustworthiness concerns may vary. Furthermore, the trustworthiness requirements typically will differ from one layer of abstraction to another (e.g., [270]), depending on the objects of interest. Trustworthiness is therefore not a monolithic concept, and is generally context dependent in its details — although there are many common principles and techniques.

Many of the principles enumerated in Chapter 2 fit together fairly nicely with the principle of minimizing the need for trustworthiness. For example, if they are sensibly invoked, abstraction, encapsulation, layered protection, robust dependencies, separation of policy and mechanism, separation of privileges, allocation of least privilege, least common mechanism, sound authentication, and sound authorization all can contribute to reducing what must be trusted and to increasing the trustworthiness of the overall system or network. However, as noted in Section 2.6, we must beware of mutually contradictory applications of those principles, or limitations in their applicability. For example, the Saltzer–Schroeder principle of least common mechanism is a valuable guiding concept; however, when combined with strong typing and polymorphism that are properly conceived and properly implemented, this principle may be worth downplaying in the case of provably trustworthy shared mechanisms — except for the creation of a weak link with respect to untrustworthy insiders. For example, sharing of authentication information and use of single signon both create new risks. Thus, this Saltzer-Schroeder principle could be reworded to imply avoidance of untrustworthy or speculative common mechanisms. Similarly, use of an object-oriented programming language may backfire if programmers are not extremely competent; it may also slow down development and debugging, and complicate maintenance. As a further example, separation of privilege may lead to a more trustworthy design and implementation, but may add operational complexity — and indeed often leads to the uniform operational allocation of maximum privilege just to overcome that complexity. As noted in Section 2.3, the concept of a single sign-on certainly can contribute to ease of use, but can actually be a colossal security disaster waiting to happen, as a serious violation of the principles of separation of privilege and least common mechanism (because it makes everything accessible essentially equivalent to a single mechanism!). In general, poorly invoked security design principles may seriously impede the user-critical principle of psychological acceptability (e.g., ease of use). (See Chapter 2 for discussion of further pitfalls.)

From an assurance perspective, many of the arguments relating to trustworthiness are based on models in which inductive proofs are applicable. One important case is that of finite-state machines in which the initial state is assumed to be secure (or, more precisely, consistent with the specifications) and in which all subsequent transitions are security preserving. This is very nice theoretically. However, there are several practical challenges. First of all, determining the soundness of an arbitrary initial state is not easy, and some of the assumptions may not be explicit or even verifiable. Second, it may be difficult to force the presence of an known secure state, especially after a malfunction or attack that has not previously been analyzed — and even more difficult in highly distributed environments. Third, the transitions may not be executed correctly, particularly in the presence of hardware faults, software flaws, and environmental hazards. Fourth, system reboots, software upgrades, maintenance, installation of new system versions, incompatible retrievals from backup, and surreptitious insertion of Trojan horses are examples of events that can invalidate the integrity of the finite-state model assumptions. Indeed, software upgrades — and, in particular, automated remote upgrades — must be looked on as serious threats. Under malfunctions, attacks, and environmental threats, the desired assurance is always likely to be limited by realistic considerations. In particular, adversaries have a significant advantage in being able to identify just those assumptions that can be maliciously compromised — from above, from within, and from below. Above all, many failures to comply with the assumptions of the finite-state model result from a failure to adequately comprehend the assumptions and limitations of the would-be assurance measures. Therefore, it is important that these considerations be addressed within the architecture as well as throughout the development cycle and operation, both in anticipating the pitfalls and in detecting limitations of the inherently incomplete assurance processes.

4.3 Principled Architecture

> *There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.*
> Sir Charles Anthony Robert Hoare

Tony Hoare's comment is obviously somewhat facetious, especially when confronted with complex requirements; he has chosen two extremes, whereas the kind of realistic system designs for inherently complex system requirements that we consider in this report obviously must lie somewhere in between. Nevertheless, Hoare's two extremes are both prevalent — the former in theory and the latter in practice.

We next seek to wisely apply the principles to the establishment of robust architectures capable of satisfying such complex requirements. Our conceptual approach is outlined roughly as follows, in a rather idealized form.

1. For each representatively comprehensive and realistic range of related system requirements, and with some working understanding of the relative importance of the desired principles relevant to those requirements, establish a spanning set of predictably composable trustworthy components from which systems of varying complexity, varying trustworthiness, and varying assurance can be developed, configured, administered, and maintained. Where generality is not naturally achievable — for example, within a particularly narrow range of requirements and applicable principles — separate architectural families should be considered instead of trying to lump everything into a common family. That is, we seek to establish some mainline families of architectures capable of attaining high security, reliability, survivability, and other critical attributes, as desired, but also to allow the general architectural framework to be adapted to special-purpose dedicated uses.

2. For a particular set of requirements within any particular architecture or family of architectures, seek to minimize what functionality must be trustworthy with respect to each of various criticalities (reliability, integrity, nondenial of service, guaranteed real-time performance, etc.).

3. Determine a minimal subset of components necessary for each specific set of requirements, and analyze it for consistency with the given requirements. (We use "minimal" to imply "minimum-like" but not necessarily the absolute minimum.) This is the notion of *stark subsetting* introduced in Section 3.3 — that is, avoiding or eliminating unneeded functionality and (hopefully) unneeded complexity and bloatware.

4. Examine the extent to which the chosen principles are satisfied, and consider the consequences. As appropriate, recycle through the previous steps for various families of architectures, further splitting families as suggested in the first step, reexamining the attainable trustworthiness as in the second step, and refining the minimal subset as in the third step, with corresponding refinements of the priorities for the principles and the architectures themselves.

5. In parallel, evaluate the extent to which the desired trustworthiness might be achieved. If high assurance is required, the requisite approaches and evaluations should be applied throughout each iteration through the development stages, and at various layers of abstraction, as appropriate. However, whereas formal analysis can be especially valuable in the development of high-assurance critical systems (and then particularly in the early development stages), it is not likely to be fruitful in the absence of a principled development. Thus, it is often unwise to attempt to apply formalisms to badly conceived designs and developments. That would be throwing good money after bad, unless it adds significantly to the awareness of how bad the architecture might be — which can usually be realized much more economically. (Formal

analysis of the software implementation in such cases is generally much less rewarding than analyses of requirements and architectures, especially if it is the design that is flawed.)

The notion of stark subsetting relates to the paired notions of composability and decomposability discussed in Section 3.3. The primary motivation for stark subsetting is to achieve minimization of the need for trustworthiness — and, perhaps more important, minimization of the need for unjustifiable (unassured) trust. Stark subsetting can also dramatically simplify the effort involved in development, analysis, evaluation, maintenance, and operation.

For a meaningfully complete stark subset to exist for a particular set of requirements, it is desirable that the stark subset originate from a set of composable components. As we note in Section 3.3, "If a system has been designed to be readily composable out of its components, then it is also likely to be readily *decomposable* — either by removal of the unnecessary subsystems, or by the generation of the minimal system directly from its constituent parts. Thus, if composability is attainable, the decomposition problem can be considered as a by-product of composition ..."

One of the architectural challenges is to attempt to capture the fundamental property of the multilevel-integrity concept, namely, that an application must not be able to compromise the integrity of the underlying mechanisms. However, there is an inherent difficulty in the above five-step formulation, namely, that satisfaction of overall system properties such as survivability and human safety depends on application software and users, not just on the integrity of the operating systems. Therefore, it is not enough to be concerned only with the architecture of the underlying infrastructure; it is also necessary to consider the entire system. (The Clark–Wilson application integrity model [82] is an example that requires such an analysis.)

Primarily for discussion purposes, we next consider two extreme subspaces in a highly multidimensional space of architectures, each with its own ranges of trustworthiness and corresponding ranges of trustedness, and with associated ranges of assurance, composability, evolvability, principle adherence, and so on. (Note that these dimensions are not necessarily orthogonal, although that is unimportant here.) Of course, there are many interesting subspaces somewhere in between these two extremes, although it is not useful to attempt to itemize them here. Within each subspace in the overall multidimensional space, there are wide variations in what properties are relevant within the concept of trustworthiness, whether the implied trust (if any) is explicit or implicit, in what kinds of assurance might be provided, and so on.

The two illustrative extremes are as follows:

- **Maximal trust: More or less uniform trust, irrespective of actual trustworthiness.** In conventional systems in which security and reliability are not deemed fundamental or are only casually addressed by the architecture, potentially every component can depend on every other component (including all application code, user programs, compilers, libraries, and system test code), and therefore must be trusted whether it is trustworthy or not, even if its failure or subversion can result in total compromise of the system, its applications, and other networked systems. (The notions of *compromise from outside, compromise from within,* and *compromise from below* in Section 2.3 encompass systemic as well as human causes.) An architecture of this kind is likely to be relatively unstructured and unprincipled. Maximal trust typically implies few if any explicit assumptions in the architecture about possibly untrustworthy components, and the entire system is implicitly assumed to behave as expected. (In many cases, trust is blind, or even worse, uniformly blind.) Although this extreme maximal-trust subspace might seem somewhat artificial, it is in reality representative of surprisingly many software products. However, it exhibits a relatively clear-cut contrast with the principle-inspired notion of minimal trust, considered next.

- **Minimal trust: Selectively wisely placed trust, based on relative trustworthiness where needed.** In an architecture that is motivated by the principle of minimization of what must be trustworthy, the architecture can ensure that greater trust need be placed primarily in certain trustworthy components, with respect to whichever requirements must be trustworthy. An overly simple example is given by a trustworthy guard that sits between two (or among more than two) systems with potentially unequal trust or with mutual suspicion (that is, bilaterally questionable trustworthiness); the guard mediates all intersystem traffic — for example, attempting to ensure no outbound leakage of sensitive information and no inbound insertion of Trojan horses; however, the trustworthy guard may typically assume little or nothing about the trustworthiness of any of the systems that it is guarding. (Some of the early guards were based rather simply on keyword scanning.) A firewall provides another instance of minimal trust, where the assumption is made that the systems on the inside cannot be compromised by the systems on the outside, even if the interior systems are not trustworthy; the firewall is trusted to satisfy only certain requirements. Wrappers provide yet another example. (Suitable use of real-time monitoring and anomaly/misuse detection may be desirable, although some of that functionality itself needs to be trustworthy with respect to system integrity, data confidentiality, and nondenials of service.) More generally, overall system architectures considered below involve reliance on certain trustworthy servers, with less trustworthiness required elsewhere.

Minimal trust is generally compatible with the notions of judicious modularity and stark subsetting. On the other hand, maximal trust is usually a consequence of badly designed systems — in which it is very difficult to achieve trustworthy subsets, let alone to remove large amounts of bloatware that in a well-designed system would conceptually not have to be trustworthy. (Compare this with the quote from Steve Ballmer given in Section 3.3.)

What might at first seem to be a hybrid minimax approach to trust and trustworthiness is given by Byzantine agreement, discussed in this context in Section 3.5: even if at most k out of n subsystems may misbehave arbitrarily badly, the overall system still behaves correctly (for suitable k and n). Byzantine agreement makes a negative assumption that some portion of the components may be *completely untrustworthy* — that is, arbitrarily bad, maliciously or otherwise — and a positive assumption that the remaining components must be *completely trustworthy*. However, Byzantine agreement is in a strict mathematical sense an example of *minimum* (rather than *minimal*) trust: in the case of Byzantine clocks, the basic algorithm [198, 331, 337] provably minimizes the number $2k + 1$ of trustworthy clock subsystems for any given number k of arbitrarily untrustworthy clock subsystems, with the resulting $3k + 1$ subsystems forming a trustworthy clock system. (Note that the assumption that at most k of the clocks may be arbitrarily untrustworthy is explicit, although the nature of the untrustworthiness can be completely unspecified.)

Purely for purposes of discussion, we next consider two extreme alternatives with respect to homogeneity versus heterogeneity of architecture, centralization and decentralization of physical configurations, logical control, trust, and trustworthiness. There are many combinations of these aspects of centralization versus decentralization, but for descriptive simplicity we highlight only two extreme cases. For example, we temporarily ignore the fact that centralized control could be exerted over highly distributed systems — primarily because that is generally very unrealistic in the face of events such as unexpected outages and denial-of-service attacks. Similarly, centralized systems could have distributed control.

- **Centralized homogeneous systems, centralized control, centralized trust, and centralized trustworthiness.** Centralized trustworthiness is largely a relic of the stand-alone batch-processing mainframe systems of the past and the centralized time-sharing systems that began emerging in the 1960s. It can be very effective in nonnetworked systems, but is impractical in any highly distributed processing environment. However, it is used effectively in stand-alone classified systems and others that must be highly secure, physically isolated from other systems and networks, and physically protectable; in such environments, administrative control is generally centralized.

- **Decentralized heterogeneous systems, distributed control, distributed trust, and distributed trustworthiness.** In general, today's systems and networks tend to be increasingly heterogeneous in the diversity of their constituent subsystems and networked components, increasingly distributed in the dispersion of their physical locations, of their users, and their maintainers, and thus necessarily distributed in their logical control. Architectures that dependably structure the relative trustworthiness of the subsystems are therefore of considerable interest.

Note that a collection of centralized subsystems may be coordinated into a decentralized system, so the boundaries of our simplified descriptive dichotomy are not always sharp. However, the trustworthiness issues in heterogeneous systems and networks (particularly with respect to security, reliability, and survivability) are significantly more critical than in homogeneous systems and networks, even though the generic problems seem to be very similar. In fact, the vulnerabilities, threats, and risks are greatly intensified in the presence of highly diverse heterogeneity.

With respect to the principled approach of minimizing what must be trustworthy, the next set of bulleted items provides some motivating concepts for structuring robust systems with noncentralized trustworthiness, irrespective of whether the actual systems have centralized or decentralized control. (For example, fault tolerance and multilevel security are meaningful in centralized as well as distributed systems.)

- **Layered trust, with layered trustworthiness.** Trustworthiness can be layered in system architectures in a variety of ways, and has been in several historically important systems. This approach was used for both security and reliability in the 1960s in the **Multics** domain-based ring structure [91, 150, 277, 333], in which faults, errors, or failures of Ring 1 would not affect the operation of Ring 0, and thus would not crash the system although they could crash the executing process; similarly, problems in Ring 2 would not affect the operation of Ring 1, and thus would not crash the executing process, although they could abort the executing command. A similar approach was used in the 1970s in the design of the hierarchically structured capability-based strongly typed object-oriented Provably Secure Operating System (**PSOS**) [120, 268, 269], in which the lowest layers (to be implemented in hardware) could be executed directly from higher layers without compromise of the access control and strong typing of the capability mechanisms. (A **capability** is in essence a potentially portable and nonforgeable token whose possession permits some sort of specified access to a particular object or class of objects. Capability-based systems have a long history, and provide considerable potential as an alternative architecture. However, a pithy quote by Butler Lampson is perhaps telling: "Capability systems are the way of the future, and always will be.")

Layered trustworthiness was also found in the **THE** operating system (see Dijkstra's 1968 paper [106]), with respect to hierarchical locking strategies, as noted in Section 3.5. It was also used to ensure human safety of real-time flight-control in the 1970s in SRI's fly-by-wire Software Implemented Fault Tolerant (**SIFT**) [232, 247, 378]) system, with a hierarchy including real-time scheduling, a broadcast protocol, and majority voting. Furthermore, layered trustworthiness is a fundamental basis of implementations of multilevel security (**MLS**) in a typical system with an MLS kernel, an MLS-trusted computing base, and MLS-untrusted applications, including the Multics kernel retrofit [344] and a higher-layer abstract type manager approach suggested as an alternative to the basic PSOS architecture. Ideally, no application software should be able to compromise the enforcement of the MLS information flow constraints — from above or from outside. In each of these cases, the trust that is either explicitly or implicitly associated with each layer can also be layered. In systems based on protection rings or MLS kernels, relatively greater trust is given to lower layers — which, because of the constructive nature of the architecture, must be relatively more trustworthy than higher layers.

In a different sense of hierarchy, trust can also be layered with respect to policies. For example, the basic multilevel security policy [35] provides for lattice-ordered levels of trust for confidentiality (e.g., a linear ordering of Top-Secret, Secret, Confidential, and Unclassified, with associated nonlinear compartments), whereas a Biba-like multilevel integrity (**MLI**) policy [43] provides a similar partial ordering for integrity. Briefly, under an MLS policy, information may not flow from one entity to another entity that has a lower (or lattice-sense incomparable) security level; in MLI, no entity may depend on another entity that has a lower (or lattice-sense incomparable) integrity level (that is, is considered less trustworthy). Of course, when we attempt to compose policies such as MLS and MLI with **MLA** (multilevel availability) and **MLX** (multilevel survivability), composability problems and general operational confusion may arise. (MLA and MLX are suggested in [264].)

MLS is of course a fundamental approach to avoiding adverse information flows, irrespective of the complexities of its implementation and operation. On the other hand, MLI, MLA, and MLX appear to be of limited usefulness in the real world — although they are of considerable interest as examples of the principled notion of trying to avoid dependence on anything less trustworthy (with respect to integrity, availability, and survivability, respectively), and enabling explicit analyses of soundness wherever such dependence cannot be avoided. A possibly more useful alternative approach is the notion of policy factoring that arises in Secure Computing Corporation's notion of type enforcement, in which separate type-related policies are designed to be seamlessly composable because of the essential disjointness of the types. (This is actually a logical outgrowth of the pervasive use of strongly typed objects throughout the hierarchical layers of PSOS [120, 268, 269].) Incidentally, another alternative use of compartmented MLS would be a single-level system with compartments. (In 1982, Lipner [209] provided a discussion of how MLS and MLI concepts might be used in commercial practice, even in compartmented single-level environments.)

The SeaView database system [100, 213, 214] discussed in Section 3.5 is another example of architectural layering in which the entire database management system is effectively multilevel secure without the off-the-shelf DBMS itself having to be trusted for multilevel security.

- **Partitioned trust, with partitioned trustworthiness.** Closely associated with the concept of layered trustworthiness is the notion of partitioned trustworthiness — on which layered trustworthiness can be built, or with which higher-layer functionality can be isolated from other functionality at the same layer. Multilevel-security kernels provide such a basic partitioning. Even more fundamental is Rushby's separation kernel [323, 324], which provides a basis for nonsubvertible isolation that can be perpetuated throughout higher layers. (See also [321], in which Rushby applies the partitioning concept to avionics architectures.) This is in essence the basis for virtual machine monitors. For example, NSA's NetTop combines SCC's SELinuxindexSELinux (multilevel security combined with strong typing) and VMWare (which provides virtual machine monitors).

  Two other forms of logical and physical separation are also worth noting — Multiple Independent Levels of Security (**MILS**) and Multiple Single-Level (**MSL**) systems, whose intercommunications are carefully controlled (discussed in the following bullet). MILS systems are expected to provide truly independent partitions that could function at different security levels, with no information flow across multilevel security boundaries except for perhaps some exceptions that are carefully controlled by the underlying virtual machine monitors. MSL systems are expected to function with each partition operating strictly within a single level, with no exceptions. These constrained modes provide restricted alternatives to a strict multilevel-security architecture, although they may typically be significantly less flexible and much less useful in general applications.

  An even stronger isolation is provided by the physical airgap approach (also referred to as sneaker-net), where there is absolutely no direct electronic connection between hardware components (ignoring electromagnetic interference and emanations). However, that approach seriously impedes interoperability; although physical separation may be exactly what is desired for extremely sensitive multilevel security compartments, it is antithetical to widespread information sharing and increasingly impractical except in extremely critical embedded system applications. Besides, sneaker-nets notoriously seem to be subverted by people carrying electronic media from one partition to another (including games bearing malicious code!).

- **Emphasis on trustworthy servers and constrained interfaces (TS&CI).** The general architecture family begun by John Rushby and Brian Randell [327] in the Newcastle multilevel-secure Distributed Secure System (see also [329] for a summary and [328] for the Newcastle report) relies heavily on trustworthy servers. This type of controlled access in a multiple-single-level (MSL) architecture was extended by Proctor and Neumann [273, 308], permitting user-covert-channel-free multilevel-secure access to information with no MLS trustworthiness required for end-user systems, and allowing single-user single-level end-user systems that need not be trusted for MLS separation or for multiuser multiplexing. Instead, architectures of this family rely heavily on sharable trustworthy servers (e.g., file servers, network servers, crypto servers, and authentication servers) that can operate as multilevel secure subsystems and enforce multilevel security; they also have highly constrained interfaces to those servers that are controlled subject to the constraints imposed by the architecture. (In [259], that architecture type is characterized by the acronym RISSC, for Reduced Interfaces for Secure System Components; however, in this report, we avoid the use of the RISSC acronym because of its pronounced confusions with *RISC* and *risk*, and instead use **TS&CI**, representing **Trustworthy Servers and Controlled Interfaces.**)

The same concept of placing relatively less trust on end-user systems and much greater trustworthiness on servers is of course also applicable to systems that do not require any multilevel security, although this architectural concept appears to be much less widely appreciated in the conventional single-level case. Placing strong emphasis on trustworthy servers is a fundamental approach to minimizing the need for trustworthiness in systems and networks. Note that the trustworthiness requirements may differ considerably from one server to another, particularly among the various requirements for security (e.g., integrity, confidentiality, reliability, survivability, and prevention of denial-of-service attacks and so-called "man-in-the-middle attacks"). Furthermore, some of the nonserver components may have very specific but less stringent requirements for trustworthiness. For example, a user platform might have only a thin-client operating system that is auto-rebooted in a trustworthy manner from an unalterable read-only memory, but otherwise with no long-term storage and some integrity in its networking software. However, the essential characteristic of such architectures is that trustworthiness with respect to certain attributes need not be dispersed uniformly everywhere throughout a distributed system or network of systems. This approach can be particularly effective because in a relatively clean way it decouples the networking abstraction (strong and supposedly robust but presumably not necessarily secure) from the computer systems (which are separated physically rather than virtually, except for the server software).

- **Emphasis on certain trustworthy clients.** In some cases (such as control-system environments or intrinsically multilevel applications), it may be desirable to have completely self-contained or almost completely self-contained end-user platforms, in which case relatively less trustworthiness may be needed elsewhere. For example, if a thin-client system never needs to download software from elsewhere (except for upgrades, which might be handled off-line or otherwise constrained), and can be assured of never executing potentially executable content (such as active e-mail attachments), then perhaps it can satisfy stringent requirements for trustworthiness within its tightly constrained perimeters. This may be particularly desirable in architectures for certain handheld wireless devices with highly constrained and controllable communications, but also with inherently risky functionality such as being able to send and receive e-mail with executable attachments, and to browse the Internet. Also, significant simplifications can result whenever the local systems can be stateless, or else necessary state information can be quickly retrieved in an intact demonstrably sound state from a remote server. On the other hand, with the advent of inexpensive supercomputing power, we can expect handheld wireless devices with enormous operating systems that are not adequately trustworthy, in which case other architectural approaches must be considered.
- **Network-centric architectures.** The need to trust various network media can be greatly (but not entirely) reduced if a network-centric architecture considers networks as virtual entities

(for example, multi-system backplanes), with appropriate trustworthiness among the attached systems as needed (e.g., for reliability, integrity, confidentiality, and nondenials of service of the critical system functionality, and especially whatever networking software must be trustworthy). Typically, the network media themselves can be generally untrusted; the content can be protected cryptographically (for integrity and confidentiality); reliable delivery and defenses against denials of service can be enhanced through alternative routing, error-correcting codes, synchronization, monitoring, dynamic reconfiguration, and so on, all of which would be suitably trustworthy according to the specified requirements. Preventing or hindering denial-of-service attacks is a particularly nasty problem, which suggests that design of the networking should not put much trust on the dependability of the network media — other than in the Byzantine sense above (namely, that some portion of the communications functionality work according to its specifications) or via alternative routing. However, even when the networking has been established as an end-to-end or system-to-system encrypted virtualized backplane, there are still serious security and particularly integrity risks that relate to both the network media and the networking software. Although it is an example of a simplistic and popular approach with real applicability, the network-centric view is still only a partial solution.

- **Emphasis on trustworthy networking.** Several alternatives exist relating to trustworthy networking, including physically isolated dedicated networks and trustworthy subnetworks of perhaps less trustworthy networks. In concept, the idea of isolated trustworthy subnetworks with stringent user authentication is very appealing, such as the Navy Marine Corp Intranet (**NMCI**) or GOVNET. The goal is to have connectivity in and out of the subnetwork that is either extremely tightly constrained (e.g., by trustworthy firewalls with rigidly enforced security and integrity policies) or else completely nonexistent. In reality, the trustworthiness of the security and integrity of such subnetworks is extremely difficult to assure, especially in the presence of trusted insiders and the need for remote maintenance paths such as the outsourcing of system administrators that seems so desirable from the perspectives of reducing costs and manpower requirements. Alternatively, the creation of virtual trustworthy subnets (e.g., virtual private networks) implemented on less trustworthy networks can be pursued, using appropriate cryptography and some trustworthy servers, with other techniques such as assured alternative routing, and anti-jamming and increased prevention against denial-of-service attacks.

- **Trustworthiness enhancement.** the techniques enumerated in Section 3.5 for increasing system trustworthiness despite less trustworthiness of the constituent subsystems are all potentially relevant within trustworthy architectures. In essence, each of those techniques allows for the reduction in the need for trust that can reasonably be placed on the subsystems, while at the same time potentially increasing the trustworthiness that can be achieved in the system as a whole. Indeed, by using some of these trustworthiness-enhancing mechanisms, the trustworthiness of the system as a whole can be significantly greater than that of the subsystems individually. As a rather dramatic example, the Software-Implemented Fault-Tolerant (SIFT) system [232, 247, 378] involved the highly redundant composition of seven off-the-shelf avionics processors, and resulted in a probably of failure five orders of magnitude less than that of a single processor for the entire fly-by-wire avionics system, as noted in Section 3.5. So-called trusted paths and trustworthy bootloads (e.g., [18, 19]) are both very important potential techniques for enhancing security (see below). Trusted paths are essential in a variety of contexts (e.g., user-to-system, system-to-user, system-to-system), particularly to prevent "man-in-the-middle", and other spoofing attacks, as well as denial-of-service attacks. Cryptographic authentication can significantly increase trustworthiness. Finer-grained authorization can be very helpful, particularly relating to separation of privileges and least privilege, as well as in reducing opportunities for insider misuse. Anything that enhances the ability to perform tracebacks can be helpful to misuse detection and response. The ability to have trustworthy paths for code distribution and dynamic updates will also be helpful.

Each of these concepts can potentially be useful by itself or in combination with other approaches. However, it is important to realize that one approach by itself may be compromisible in the absence of other approaches, and that multiple approaches may not compose properly and instead interfere with one another. Thus, an architecture (or a family of architectures) must have considerable effort devoted to combining elements of multiple concepts into the effective development of trustworthy systems, with sufficiently trustworthy networking as needed. However, although these concepts are not necessarily disjoint and may potentially interfere with one another, each of these concepts is generally compatible with the notion of stark subsetting — which of course itself benefits greatly from extensive composability (and its consequence, facile decomposability).

Many other system properties of course can also contribute to achieving our desired goals. A few of these are discussed next. These items are somewhat second-order in nature, because they rely on the trustworthiness of the design and implementation of the architectural concepts in the above list — although they also can each contribute to increased trustworthiness.

- **Dramatically improved user authentication** is needed to overcome many of the common risks of reusable fixed passwords. Nonreusable and difficult-to-forge cryptographic tokens, nonreusable one-time pass phrases (as in the primitive but erstwhile transitionally useful S-key system), and biometrics are all potentially useful in increasing authentication trustworthiness, but are nevertheless vulnerable if embedded in flaky operating systems or insecure applications. Clearly, a multipronged approach is needed, rather than just relying on a single factor. Furthermore, user authentication needs to be fairly nonintrusive from the perspective of the user; otherwise, it tends to be avoided, bypassed, or trivialized.

- **Dramatically improved message authentication** is also important. A recent paper [360] (one in a chain of research efforts) is aimed at defining and analyzing effective mechanisms for authenticating each network packet against malicious and erroneous disruptions; the paper includes numerous references to previous efforts in that direction.

- **Fine-grained authorization** (e.g., differential or context-dependent access controls) can narrow down the extent of misuse, and can bring actual system behavior more closely in line with policies of intended behavior. It is particularly relevant in enabling access controls to more closely implement security policies — which may be particularly important whenever insider misuse is a serious concern. It is also helpful in constraining outsiders who have effectively become insiders as a result of system penetrations. However, it is rendered rather limited in effectiveness if the authentication is not trustworthy.

- **Trustworthy bootloads** can provide assurance of the genuineness and integrity of the underlying operating systems (e.g., Arbaugh [18, 19]). Authentication, authorization, monitoring, and accountability will always be suspect if the underlying operating systems are not trustworthy, but especially if tampering has resulted in the presence of trapdoors or Trojan horses.

- **"Trusted paths"** (or, more appropriately, *trustworthy paths*) can provide protected communication links in which there is at least a unilateral direction of dynamic confidence, namely that a user is truly in contact with the intended system (rather than a spoofed version of that system); in some cases, a trustworthy path may need to be bilateral — that is, also providing dynamic confidence that a given system is truly in contact with its intended user (rather than an interloper or imposter). In each of these directions, the term *user* may apply to systems or subsystems as well as people. Trustworthy paths themselves require considerably improved authentication of computational entities (e.g., systems, subsystems, processes, network components), trustworthy bootloads, and in some cases even dedicated physical resources.

- **Systemic support for traceback** (especially in packet-based networked systems) requires some ability to determine the origins of an attack and its misusers. Traceback in turn depends on meaningful authentication of users, routers, operating systems applications, some trustworthy bootloads, some trustworthy paths, and so on. In the foreseeable future, trustworthy traceback seems feasible at best only within relatively self-contained subnetworks. Dean et al. [97] provide an algebraic formulation of a practical approach to IP traceback during a denial-of-service attack; this paper should be very useful as a basis for future research on traceback. See also a subsequent optimization by Micah Adler [10] that reduces the required overhead to an extra bit per packet!

- **Trustworthy code distribution** is essential to provide assurance that downloaded software as delivered is in fact genuine and untampered, and has the expected provenance (pedigree). This is of particular importance in networked and Web-based environments and in thin-client system architectures in which software is normally downloaded dynamically.

- **Real-time monitoring and misuse detection** are fundamental to sound operation of systems and networks. Ideally, system monitoring might seem unnessary if all the above mechanisms were perfectly designed and properly working, but that of course is unrealistic. Actually, even if everything else in the above list worked perfectly (which, as we know, is an extremely unwise expectation), detecting, identifying, and responding to insider misuse would still be desirable. Furthermore, network monitoring is always important from the perspectives of reliability and availability as well as security (for example, see [34]). Partially automated responses to emergencies could also be useful, although they should be used with great care — especially if they can be used to induce denial-of-service attacks! (Drew Dean notes a similarity with game theory, in attempting to prevent the attacker from having the last move in an arms race.) See also work on intrusion tolerance (e.g., [103, 124]) as opposed to intrusion detection.

- **Alternative hardware** might be desirable in certain circumstances — for example, for critical servers in systems that require very high trustworthiness and very high assurance. Significant benefits can be derived from building critical software systems on high-assurance, robust, and more easily secured hardware platforms. Although special-purpose hardware in recent years seems to have been going the way of the dodo bird, there are applications in which customized hardware could be very useful — for example, in constraining what the software can do. Special-purpose co-processors are one example, as for example in the LOgical Coprocessor Kernel (**LOCK**), or more recently the Trusted Computing Platform Alliance (**TCPA**)/Trusted Computing Group (**TCG**), Intel's LaGrande technology, and Microsoft's Palladium/Next Generation Secure Computing Base (**NGSCB**), presumably to be used in Microsoft's next-generation operating system, Longhorn. (For some provocative background, particularly on potential limitations, see Ross Anderson's Trusted Computing FAQ, http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html.) The "No Execute" (NX) facility also has some merit, although it can in principle better be achieved with sensible domain architectures. In addition, there is some hope that newer hardware architectures, such as the IBM/HP efforts in blade computers (which can combine multiple conventional processors into a single circuit board) might actually lead to some efficient hardware isolation kernels and in the long run to the possibility of high-assurance multilevel secure systems. This remains a very interesting possibility for the future.

- **Preventing or at least hindering denial-of-service attacks** presents some enormous challenges for system and network architectures, particularly in highly distributed systems and networks in which communications are already vulnerable. Authentication and access control become extremely important in preventing unwanted access and facilitating traceback within perimeters of trustworthiness. However, attempts to achieve end-to-end security across untrustworthy networks are always vulnerable to denial-of-service attacks. Furthermore, networking protocols must provide some means of limiting adverse traffic in situations in which authentication and access control cannot. Monitoring and real-time analysis become increasingly important, particularly in providing early detection and rapid remediation in the presence of suspected denial-of-service attacks.

Appropriate architectures are then likely to be some sort of combination of the above approaches, encompassing (for example) heterogeneous subsystems and subnetworks, trustworthy servers and controlled interfaces (TS&CI) that that ensure satisfaction of cross-domain security and integrity, dramatic improvements in system and network-wide authentication, trustworthy bootloads, trusted paths, traceback, trustworthy code distribution, and other concepts included in the above enumeration, particularly in observance of the principles of Chapters 2 and 3. Such architectures (referred to herein as the **Enlightened Architecture Concept** would provide a basis for a wide class of systems, networks, and applications that can heterogeneously accommodate high-assurance security. This is particularly relevant concerning desires for multilevel security, which realistically are likely to involve collections of MLS clients, MSL clients, and MILS clients, all controllably networked together with a combination of servers, subject to the multilevel constraints, and with a similar assortment of assurance techniques for both conventional security and multilevel security. In that true multilevel security is overkill for many applications, this vision would provide that functionality only where essential.

4.4 Examples of Principled Architectures

> *In our experience, software exhibits weak-link behavior; failures in even the unimportant parts of the code can have unexpected repercussions elsewhere.* David Parnas et al. [292]

The pervasive nature of weak links is considered in Section 2.3.1 in connection with principles for avoiding them, and again in Section 3.5 in connection with the desire for reduced dependence on trustworthiness. In concept, we like to espouse **strength in depth;** however, in practice, we find **weakness in depth** in many poorly architected systems — where essentially every component may be a weak link. Even well-designed systems are likely to have multiple weak links, especially in the context of insider misuse. As a result, there is a fundamental asymmetry between defenders and attackers. The defenders need to avoid or protect all of the vital weak links; on the other hand, the attackers need to find only one or just a few of the weak links.

Several historically relevant systems are particularly illustrative of the concept of principled architectures that have sought to avoid weak links in one way or another. These are discussed next.

- **Multics.** Multics [277] (noted above in Section 4.3) was perhaps the first operating system to make extensive use of structure within the operating system itself. The eight concentric rings and protection domains permitted process protection within the operating system and allowed multiple application layers to be separated from the operating-system layers, and also allowed iterative implementation of policy-mechanism separation [150, 334, 345]. It also facilitated the subsequent retrofit of multilevel security [344], with only relatively minor repartitioning of Ring 0 and Ring 1. The multilayered directory hierarchy [92] permitted sensible directory structuring, symbolic file names, directory search strategies, and dynamic linking, along with access-control lists to permit finer-grained access controls. The virtual memory implementation [91] demanded separation of symbolic and physical addressing, where all real objects in memory are accessed by means of symbolically named virtual objects and dynamically linked on demand. Multics is considered in further detail in this context in [259]. An important contribution to the evaluation of Multics security is given by Paul Karger and Roger Schell in two papers [188, 189], the first from 1974 and the second revisiting that early paper in 2002. Among other things, the later paper notes the relevance of the choice of PL/I as the programming language for implementation and the relevance of the underlying hardware segmentation, paging, and protection, specifically in avoiding buffer overflows, data interpreted as executables, Trojan horses, and other characteristic security problems — and generally greatly enhancing security. See also Fernando Corbató's Turing lecture [84] for a survey of some of the lessons learned from the Multics development.

- **THE.** The hierarchically layered Eindhoven operating system THE [106] (noted in Sections 3.5 and 4.3) demonstrated that a strict hierarchical locking strategy could avoid deadly embraces between layers. This concept is important in preventing denials of service as well as ensuring high system availability.

- **PSOS and SIFT.** 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) [120, 268, 269] (noted in Section 4.3) and the Software-Implemented Fault-Tolerant System (SIFT) [232, 247, 378] (also noted in Section 4.3), both of which were formally specified according to the SRI Hierarchical Development Methodology (HDM). The object-oriented strongly typed hierarchically layered PSOS design is considered in Section 4.3. The SIFT design and prototype implementation represented a seven-processor fly-by-wire avionics system that was resistant to hardware faults or even complete outages of individual processors and memories; in the presence of extensive faults, it was self-diagnosing and self-reconfiguring.

- **SAT/LOCK/Sidewinder type-based protection.** The strongly typed PSOS design (which in several senses an early system adhering to the principle of minimization of the need for trustworthiness) 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) [123, 153, 154, 155, 156, 335, 347, 348] and the notion of trustworthy pipelines [53] — in which each stage in a pipeline can have its own security policy. The SCC Sidewinder firewall is another example using type-based protection.

- **Trustworthy servers.** A rather different lineage that adheres to the principle of minimal need for trustworthiness begins with the Rushby–Randell Newcastle Distributed Secure System DSS [327], which involved a collection of single-level Unix systems linked via trustworthy network interface units (TNIUs). A descendent of DSS is now also commercially available (DRA) in the United Kingdom. See also the TS&CI concept considered earlier in this section, which is a further refinement of DSS. The Gemini GEMSOS and Trusted Network Processor (GTNP) multilevel-security gateway efforts also bear considerable kinship with Multics and PSOS, and with the principle of minimizing the need for trustworthiness. Both LOCK and GTNP are operational systems with potential usefulness in the DoD Trusted Computer System Evaluation Criteria B3/A1-rated applications.

- **Kernel-based systems.** A TCSEC-orthodox system lineage [249] includes end-user systems with multilevel-secure kernels and associated trusted computing bases (TCBs) (a misnomer for *trustworthy computing bases*), such as the Ford Aerospace Kernelized Secure Operating System (Unix-based) KSOS [39, 221], the MITRE [336] and UCLA Secure Unix kernels, and the kernelized virtual-machine system KVM for IBM's VM [141, 332]. Rushby's separation kernel [323, 324] concept 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 or system safety can be implemented. This approach represents an important realization of the principle of separation of policy and mechanism, as does MIT's Exokernel operating system architecture [113, 114].

- **SeaView.** The SeaView DBMS noted in Section 4.3 provides a further example, especially in its ability to avoid having to trust the DBMS and user application software to enforce multilevel security — because of the constraints imposed by the MLS kernel.

- **EMERALD.** The architecture of SRI's EMERALD system [272, 304] for anomaly and misuse detection (also noted in Section 3.5) makes considerable use of hierarchical structure in providing the ability to correlate over multiple entities at various layers of abstraction.

4.5 Openness Paradigms

Closed-source paradigms often result in accidental **open-sesames.**
Can **open kimonas** *inspire better software?*

[This section is adapted from Neumann's paper for the 2000 IEEE Symposium on Security and Privacy, entitled "Robust Nonproprietary Software" [265].]

Various alternatives in a spectrum between "open" and "closed" arise with respect to many aspects of the system development process, including the availability of documentation, design, architecture, algorithms, protocols, and source code. The primary differences arise among many different licensing agreements. The relative merits of various paradigms of open documentation, open design, open architecture, open software development, and available source code are the source of frequent debate, and would benefit greatly from some incontrovertible and well documented analyses. (For example, see [210, 227, 263, 265, 338] for a debate on open source-code availability. See also [126] on the many meanings of *open-source*.) The projects in the DARPA CHATS program
(http://www.darpa.mil/ipto/research/chats/index.html) provided some strong justifications for not only the possibilities of openness paradigms, but also some realistic successes.

As noted throughout this report, our ultimate goal is to be able to develop robust systems and applications that are capable of satisfying critical requirements, not merely for security but also for reliability, fault tolerance, human safety, survivability, interoperability, and other vital attributes in the face of a wide range of realistic adversities — including hardware malfunctions, software glitches, inadvertent human actions, massive coordinated attacks, and acts of God. Also relevant are additional operational requirements such as interoperability, evolvability and maintainability, as well as discipline in the software development process and assurance associated with the resulting systems.

Despite extensive past research and many years of system experience, commercial development of computer-communication systems is decidedly suboptimal with respect to its ability to meet stringent requirements. This section examines the applicability of some alternative paradigms to conventional system development.

To be precise about our terminology, we distinguish here between *black-box* (that is, closed-box) systems in which source code is not available, and *open-box* systems in which source code is available (although possibly only under certain specified conditions). Black-box software is often considered as advantageous by vendors and believers in security by obscurity. However, black-box software makes it much more difficult for anyone other than the original developers to discover vulnerabilities and provide fixes therefor. It also hinders open analysis of the development process itself (which, because of extremely bad attention to principled development in many cases, is something developers are often happy to hide). Overall, black-box software can be a serious obstacle to having any objective confidence in the ability of a system to fulfill its requirements (security, reliability, safety, interoperability, and so on, as applicable). In contrast, our use of the term *open-box software* suggests not only that the source code is visible (as in *glass-box software*), but also that it is possible to reach inside the box and make modifications to the software. In some cases, such as today's all-electronic (e.g., paperless) voting systems, in which there is no meaningful assurance that votes are correctly recorded and counted, and no useful audit trails that can be used for a recount in the case of errors or system failures (for example, see [194, 233, 235]), black-box software presents a significant obstacle to confidence in the integrity of the entire application. On the other hand, completely open-box software would also provide opportunities for arbitrary software changes — and, in the case of electronic voting systems, that enable elections to be rigged by malicious manipulators (primarily insiders). Thus, there is a need for controls on the provenance of the software in both open-box and closed-box cases — tracking the history of changes and providing evidence as to where the code actually came from.

We also distinguish here between *proprietary* and *nonproprietary* software. Note that open-box software can come in various proprietary and nonproprietary flavors, with widely varying licensing agreements regarding copyright, its supplemental concept of *copyleft,* reuse with or without the ability to remain within the original open-source conditions, and so on.

Examples of nonproprietary open-box software are increasingly found in the Free Software Movement (such as the Free Software Foundation's GNU system with Linux) and the Open Source Movement, although discussions of the distinctions between those two movements and their respective nonrestrictive licensing policies are beyond the current scope. In essence, both movements believe in and actively promote unconstrained rights to modification and redistribution of open-box software. (The Free Software Foundation Web site is `http://www.gnu.org`, and contains software, projects, licensing procedures, and background information. The Open Source Movement Web site is `http://www.opensource.org/`, which includes Eric Raymond's "The Cathedral and the Bazaar" and the Open Source Definition.)

The potential benefits of nonproprietary open-box software include the ability of good-guy outsiders to carry out peer reviews, add new functionality, identify flaws, and fix them rapidly — for example, through collaborative efforts involving geographically dispersed people. Of course, the risks include increased opportunities for evil-doers to discover flaws that can be exploited, or to insert Trojan horses and trap doors into the code.

Open-box software becomes particularly interesting in the context of developing robust systems, in light of the general flakiness of our information system infrastructures: for example, the Internet, typically flawed operating systems, vulnerable system embeddings of strong cryptography, and the presence of mobile code. Our underlying question of where to place trustworthiness in order to minimize the amount of critical code and to achieve robustness in the presence of the specified adversities becomes particularly relevant.

Can open-box software really improve system trustworthiness? The answer might seem somewhat evasive, but is nevertheless realistic: *Not by itself, although the potential is considerable. Many factors must be considered.* Indeed, many of the problems of black-box software can also be present in open-box software, and *vice versa*. For example, flawed designs, the risks of mobile code, a shortage of gifted system developers and intelligent administrators, and so on, all apply in both cases. In the absence of significant discipline and inherently better system architectures, opportunities may be even more widespread in open-box software for insertion of malicious code in the development process, and for uncontrolled subversions of the operational process. However, in essence, many of the underlying developmental problems tend to be very similar in both cases.

Ultimately, we face a basic conflict between (1) security by obscurity to slow down the adversaries, and (2) openness to allow for more thorough analysis and collaborative improvement of critical systems — as well as providing a forcing function to inspire improvements in the face of discovered attack scenarios. Ideally, if a system is meaningfully secure, open specifications and open-box source should not be a significant benefit to attackers, and the defenders might be able to maintain a competitive advantage! For example, this is the principle behind using strong openly published cryptographic algorithms, protocols, and implementations — whose open analysis is very constructive, and where only the private and/or secret keys need to be protected. Other examples of obscurity include tamperproofing and obfuscation, both of which have very serious realistic limitations. Unfortunately, many existing systems tend to be poorly designed and poorly implemented, and often inherently limited by incomplete and inadequately specified requirements. Developers are then at a decided disadvantage, even with black-box systems. Besides, research initiated in a 1956 paper by Ed Moore [241] reminds us that purely external *Gedanken* experiments on black-box systems can often determine internal state details. Furthermore, reverse engineering is becoming quite feasible, and if done intelligently can result in the adversaries having a much better understanding of the software than the original developers.

Static analysis is a vital contributor to increasing assurance, and is considered in Section 6.6.

Behavioral application requirements such as safety, survivability, and real-time control cannot be realistically achieved unless the underlying systems are adequately trustworthy. It is very difficult to build robust applications on either proprietary closed-box software or nonproprietary open-box software that is not sufficiently trustworthy — once again this is like building castles in the sand. However, it may be even more difficult for closed-box proprietary systems.

Unless the fantasy of achieving security by obscurity is predominant, there seem to be some compelling arguments for open-box software that encourages open review of requirements, designs, specifications, and code. Even when obscurity may be deemed necessary in certain respects, some wider-community open-box approach may be desirable. For system software and applications in which security can be assured by other means and is not compromisible within the application itself, the open-box approach has particularly great appeal. In any event, it is always unwise to rely *primarily* on security by obscurity.

So, what else is needed to achieve trustworthy robust systems that are predictably dependable? The first-level answer is the same for open-box systems as well as closed-box systems: serious discipline throughout the development cycle and operational practice, use of good software engineering, rigorous repeated evaluations of systems in their entirety, and enlightened management, for starters.

A second-level answer involves inherently robust and secure evolvable composable interoperable architectures that avoid excessive dependence on untrustworthy components. One such architecture is noted in Section 4.3, namely, thin-client user platforms with minimal operating systems, where trustworthiness is bestowed where it is essential — typically, in starkly subsetted servers and firewalls, code distribution paths, nonspoofable provenance for critical software, cryptographic coprocessors, tamperproof embeddings, preventing denial-of-service attacks, runtime detection of malicious code and deviant misuse, and so on.

A third-level answer is that there is still much research yet to be done (such as on techniques and development practice that enables realistic predictable compositionality, inherently robust architectures, and sound open-box business models), as well as more efforts to bring that research into practice. Effective technology transfer seems much more likely to happen in open-box systems.

Above all, nonproprietary open-box systems are not in themselves a panacea. However, they have potential benefits throughout the process of developing and operating critical systems. Nevertheless, much effort remains in providing the necessary development discipline, adequate controls over the integrity of the emerging software, system architectures that can satisfy critical requirements, and well-documented demonstrations of the benefits of open-box systems in the real world. If nothing else, open-box successes may have an inspirational effect on commercial developers, who can rapidly adopt the best of the results. We are already observing some of the major commercial system developers exploring some of the alternatives for open-box source-code distribution. The possibilities for coherent community cooperation are almost open-ended (although ultimately limited in scale and controllability), and offer considerable hope for nonproprietary open-box software — if the open-box community adopts some concepts of principled architectures such as those discussed here.

Of course, any serious analysis of open-box versus closed-box and proprietary versus nonproprietary must also take into account the various business models and legal implications. The effects of the federal Digital Millennium Copyright Act (DMCA), the state Uniform Computer Information Transactions Act (UCITA), shrink-wrap restrictions, and other constraints must also be considered. However, these considerations are beyond the present scope.

A recent report [163] of the Carnegie-Mellon Software Engineering Institute provides a useful survey of the history and motivations for open-source software.

4.6 Summary

If carpenters built the way programmers program, the arrival of the first woodpecker would mean the end of civilization as we know it. Gerald Weinberg

In summarizing the conclusions of this chapter, we revisit and extend the quasi-Yogi Berra quote at the beginning of Section 4.1. A system is unlikely to be trustworthy if it does not have a sufficient supply of good designers, good programmers, good managers, and good system administrators. However, it is also not likely to be secure, reliable, generally trustworthy, evolvable, interoperable, and operationally manageable if the development does not begin with feasible requirements that are well specified and realistically representative of what is actually needed, *and* if it does not involve good specifications and good documentation, *and* if it does not use good compilers, good development tools, and lots more. Note that if a set of requirements is trivial or seriously incomplete, the fact that a system satisfies those requirements is of very little help in the real world.

Thus, appropriately well defined and meaningful requirements for trustworthiness are essential. Good system and network architecture is perhaps the most fundamental aspect of any efforts to develop trustworthy systems, irrespective of the particular set of requirements whose satisfaction is necessary. Wise adherence to a relevant set of principles can be extremely helpful. Architectural composability and implementation composability are of enormous importance, to facilitate development and future evolution. Policy composability is also useful if multiple policies are to be enforced. Good software engineering practice and the proper use of suitable programming languages are also vital. The absence or inadequacies of some of these ideals can sometimes be overcome. However, sloppy requirements and a fundamentally deficient architecture represent huge impediments, and will typically result in increased development costs, increased delays, increased operational costs, and future incompatibilities.

As we note at the end of Chapter 3, seamless composability is probably too much to expect overall, particularly in the presence of legacy software that was not designed and implemented to be composable; instead, we need to establish techniques that can provide composability sufficient to meet the given requirements. If that happens to be seamless in the particular case, so much the better.

We believe that the approaches considered in this report have almost open-ended potential for the future of trustworthy information systems. They are particularly well suited to the development of systems and networking that are not hidebound by compatibility with legacy software (and, to some extent, legacy hardware), but many of the concepts are applicable even then. We hope that these concepts will be adopted much more widely in the future by both open-box and closed-box communities. In any case, much greater discipline is needed in design, development, and operation.

Chapter 5

Principled Interface Design

> **Perspicuous:** *plain to the understanding, especially because of clarity and precision of presentation.* (Webster's International Dictionary)

Synopsis

This chapter considers system architecture from the viewpoint of external and internal system interfaces, and applies a principled approach to interface design.

5.1 Introduction

Interfaces exist at different layers of abstraction (hardware configuration, operating systems, system configurations, networking, databases, applications, control system complexes such as SCADA systems and air-traffic control, each with both distributed and local control) and should reflect the abstractions of those layers and any security issues peculiar to each layer, suitable for the specific types of users. In general, security considerations should be hidden where possible, except where it is necessary for control and understandability of the interfaces. In addition, some sort of automated (or at least semiautomated) intelligent assistance is essential, according to specific user needs.

Operators, administrators, and users normally have different needs. Those needs must be reflected in the various interfaces — some of which must not be accessible to unprivileged users. In particular, operators of control systems, enterprises, and other large-system applications need to be able to see the big picture at an easily understood layer of abstraction (e.g., dynamic status updates, configuration management, power-system error messages), with the ability on demand to drill down to arbitrarily fine-grained details. As a consequence, it is generally necessary that greater detail must be available to certain privileged users (for example, system and network administrators or system operators), according to their needs — either through a separate interface or through a refinement mechanism associated with the standard interface.

In general, it is important that the different interfaces for different roles at different layers be consistent with one another, except where that is prevented by security concerns. (This is a somewhat subtle point: in order to minimize covert channels in multilevel secure systems, it may be deemed advisable that different, potentially inconsistent, versions of the same information content must be accorded to users with different security levels. This multiplicity of content for seemingly the same information is known as **polyinstantiation**.) Most important is that the interfaces truly reflect the necessary trustworthiness issues.

Requirements must address the interface needs at each layer, and architectures must satisfy those requirements. This is very important, and should be mirrored in the requirements and architecture statements. In general, good requirements and good architectures can avoid many otherwise nasty administrative and user woes — viruses, malcode, patch management, overdependence and potential misuse of superuser privileges. As an example, the Trusted Xenix system requirements demanded a partitioning of privileged administrator functions rather than allowing a single superuser role. This illustrates the principles of separation of duties and a corresponding separation of roles.

In attempting to simplify the roles of adminstrators and operators, automated vendor-enforced updates are becoming popular, but represent a huge source of security risks. Their use must be considered very carefully – commensurate with the criticality of the intended applications. Remote maintenance interfaces are vital, especially in unmanned environments, but also represent considerable security risks that must be guarded against.

The rest of this chapter as well as Sections 7.6 and 8.4, and some of Section 7.11 are adapted from the body of a self-contained report, "Perspicuous Interfaces", authored by Peter Neumann, Drew Dean, and Virgil Gligor as part of a seedling study done for Lee Badger at DARPA under his initiative to develop a program relating to Visibly Controllable Computing. That seedling study was funded as an option task associated with SRI's CHATS project. Its report also included an appendix written by Virgil Gligor, entitled "System Modularity: Basis for the Visibility and Control of System Structural and Correctness Properties", which is the basis for Appendix B of this report, courtesy of Virgil Gligor.

5.2 Fundamentals

> *The Internet is arguably the largest man-made information system ever deployed, as measured by the number of users and the amount of data sent over it, as well as in terms of the heterogeneity it accommodates, the number of state transitions that are possible, and the number of autonomous domains it permits. What's more, it is only going to grow in size and coverage as sensors, embedded devices, and consumer electronics equipment become connected. Although there have certainly been stresses on the architecture, in every case so far the keepers of the Internet have been able to change the implementation while leaving the architecture and interfaces virtually unchanged. This is a testament to the soundness of the architecture, which at its core defines a "universal network machine". By locking down the right interfaces, but leaving the rest of the requirements underspecified, the Internet has evolved in ways never imagined.* Larry Peterson and David Clark [301]

This chapter seeks to provide guidelines for endowing system interfaces and their administrative environments with greater perspicuity, so that designers, developers, debuggers, administrators, system operators, and end users can have a much clearer understanding of system functionality and system behavior than is typically possible today. Although the primary concern is for interfaces that are visible at particular layers of abstraction, the approach is immediately also applicable to internal interfaces.

As is true with security in general, the notion of perspicuity is meaningful primarily only with respect to well-defined criteria (assuming suitable definitions). Some desirable perspicuity criteria and characteristics are considered in Section 5.2.3.

The approach here considers the traditional problems of design, implementation, operation, and analysis, and suggests ways to achieve the basic goal of perspicuity. It spans source-code analysis, the effects of subsystem composition, debugging, upgrades and other program enhancements, system maintenance, code generation, and new directions. It addresses the relevance of specification languages, programming languages, software engineering development methodologies, and analysis tools. It is applicable to multiple layers of abstraction, including hardware, operating systems, networks, and applications. It considers formal methods, ad-hoc techniques, and combinations of both. Other relevant architectural and system-oriented considerations are characterized in Chapter 4.

The main emphasis here is on the understandability of the interfaces and the functionality that they represent. Toward that end, we first seek evaluation criteria for and constraints on relevant interfaces (and on development processes themselves) that can help avoid many of the commonly experienced problems relating to security and reliability. We then explore a range of tools that might help detect and eliminate many of the remaining problems and that might also improve the perspicuity of critical software. It is clear that this problem is ultimately undecidable in a strict sense, but nevertheless much can be done to advance the developmental and operational processes.

This report is not intended as a detailed treatise on the subject of perspicuous interfaces. Instead, it provides an enumeration of the basic issues and some consideration of relative importance of possible approaches, as well as an understanding of how interface design fits into the overall goal of principled assuredly trustworthy composable architectures.

5.2.1 Motivations for Focusing on Perspicuity

There are several reasons for expending efforts on enhancing perspicuity.

- **Criticality of the roles of security administration and system administration.** Surveys from the mid-1990s suggested that security administration was the top DoD security concern. This appears still to be true, and perhaps even more so. DoD's increased outsourcing of system/security administrators (attributed to the complexities that their job responsibilities entail and the difficulties of paying them competitively) reflects an ever-increasing risk. The popular press suggests that buffer overflows may be our biggest security problem, but that supposition is clearly a gross oversimplification, and is dwarfed by the critical dependence on admins.

- **The growing numbers of SysAdmins.** In the United States, there are apparently more individuals listing their occupation as computer and network "system administrators" than individuals listing their occupation as "teacher" — according to an association of systems administrators. Even if this is not true, it would be nice to make the life of systems administrators significantly easier, wherever possible. Perhaps their votes should count more heavily when it comes to picking winners and losers in security research! (Extrapolating the present rate of growth, assuming that the current course does not change dramatically, one could estimate that by 2020 there would be more SysAdmins than people working in computer system R&D, although they all may be outsourced by then!)

- **Benefits of perspicuity.** Increased interface perspicuity can contribute to ease of use, scalability, maintainability, system evolution, and robustness of security administration. Each of these attributes has direct and indirect implications with respect to security and reliability — as well as other aspects of trustworthiness. As with many of these aspects, interface perspicuity clearly needs to be reflected in requirements and fully integrated into system architectures. However, some of the most significant benefits of this will remain largely invisible until systems and networks break, or are subjected to attack, or must be reconfigured. Overall, it seems that enlightened self-interest dictates that we devote development resources to usability and particularly to security administration, putting emphasis on interface perspicuity.

  In addition to the above reasoning on the potential reasons for focusing on perspicuity related to SysAdmins, numerous benefits can accrue to system programmers, application programmers, and a wide variety of users – particularly with respect to increased ease of use, program understandability, debuggability, maintainability, interoperability, and ease of integration — and of course the ability to explain unexpected errors and effects of malicious misuse.

5.2.2 Risks of Bad Interfaces

The archives of the Risks Forum are replete with examples of badly conceived and badly implemented interfaces, with consequential losses of life, injuries, impairment of human well being, financial losses, lawsuits, and so on. A few examples are summarized here — for which references and further details can be found in the RISKS archives at http://www.risks.org, a topical index for which is found in the ever-growing Illustrative Risks document [267]:
http://www.csl.sri.com/neumann/illustrative.html. (Some of the pre-1994 incidents are also described in [260].)

- In many aircraft accidents, airplane manufacturers have tended to place blame on pilots and air-traffic controllers, although in many of those cases the pilots and controllers have justifiably blamed the human interfaces of the computer-communication systems. However, as systems become increasingly automated, aircraft crew reliance on automation is considered a major future risk.

- Deficiencies in human-computer interfaces were directly implicated in various plane crashes and resulting deaths, including a China Air A300-600, the French Air Inter A320, and the Airbus A320 at the Paris Air Show — which involved a conflict between the pilot and the autopilot.

- Iran Air flight 655 (an Airbus) was shot down by the Vincennes' Aegis system, in part as a result of a seriously flawed human-computer interface that was missing several critical pieces of information, and that misidentified the Airbus as a fighter plane that had previously been co-linear on the runway before takeoff.

- A British Midland 737 crash was caused when the pilot shut off the good engine rather than the failing one, as a result of a cross-wired display.
- The KAL 007 shootdown has been attributed to the plane having flown on an erroneous autopilot course apparently missed by the pilot, who then left the cabin.
- An F-16 landing gear was retracted while the plane was on the runway, as a result of a missing interlock.
- A Special Forces GPS system accidentally targeted itself after a battery replacement that by default reset the target information to its own location, unbeknownst to the operator.
- Poorly designed interfaces on heart pacemakers, heart monitors, defibrillators, anesthesia controls, and so on, have resulted in patient deaths.
- In an experiment conducted from the Shuttle Discovery, a mirror intended to reflect a laser beam from the top of Mauna Kea was positioned upside down, focused upward to 10,023 *miles* instead of downward to 10,023 *feet* elevation, due to a confusion of units.
- A heart monitoring device with a standard wall-plug connector for connecting the probes to the monitoring device was discovered unplugged by a hospital attendant, who plugged it into a wall socket — electrocuting the patient.

Neumann's Inside Risks column from the March 1991 *Communications of the ACM* ("Putting Your Best Interface Forward") includes more detailed discussions of several examples, and is the basis for [260], pp. 206–209.

There are many other emerging applications that will have serious risks associated with nonperspicuity of their human interfaces, especially in systems intended to be largely autonomic. One critical application involves adaptive automobile cruise-control that adjusts to the behavior of the preceding car(s) (including speed and acceleration/deceleration, lane changes, and so on). Some of this functionality is beginning to emerge in certain new cars. For example, BMW advertises an automobile with an 802.11 access point that would enable downloading of new software (presumably by the factory or mechanic, but perhaps even while you are driving?). The concept of a completely automated highway in the future will create some extraordinary dependencies on the technology, especially if the human interfaces provide for emergency overrides. Would you be comfortable on a completely automated networked highway system alleged to be safe, secure, and infallible, where your cruise-control chip is supposedly tamperproof, is supposed to be replaced only by approved dealers, is remotely reprogrammable and upgradeable, and can be monitored and controlled remotely by law enforcement — which can alter its operation in a chase among many other vehicles?

5.2.3 Desirable Characteristics of Perspicuous Interfaces

The major issues underlying our main goal require a characterization of the requirements that must be met by system architectures and by their visible and hidden interfaces, as well as constraints that might be considered essential.

A popular belief is that highly trustworthy systems with nontrivial requirements are inherently complex. However, we observe in Chapter 4 that — in a well-designed system — complexity can be addressed structurally, yielding apparent simplicity locally even when the overall system is complex. To this end, **abstractional simplicity** is highly desirable. It can be achieved as a by-product of sound system design (e.g., abstraction with strong typing and strong encapsulation), well conceived external and internal interfaces, proactive control of module interactions, and clean overall control flow. For existing legacy systems in which abstractional simplicity may not be attainable directly, it may still sometimes be attainable through wrappers whose interfaces provide appropriate abstraction. In any case, aids to analysis can help significantly. Thus, a sensible approach to perspicuous computing needs to address the system design structure, all of the relevant interfaces (visible or not), and the implementation. Thus, techniques for analyzing interfaces for perspicuity and other characteristics would be very valuable.

We begin with a consideration of **desirable interface characteristics:**

- Interfaces should represent **layered modular abstractions,** with encapsulation and information hiding. Architecturally, each interface should mask the complexity within its implementation, hiding internal data structures and state information, and completely representing all visible inputs, outputs, expected behavior, and possible exception conditions. The module structure and the module interfaces should be constrained to avoid unspecified cross-dependencies. This can greatly simplify the apparent complexity of an interface, and can mask the complexity of the underlying implementation — particularly any unavoidable interactions among different modules or different instantiations of the same module.

- Interfaces should be **well defined,** with complete, accurate and consistently maintained documentation of all inputs, outputs, expected behavior, and possible exception conditions.

- **Uniform conventions** should exist within each interface and among different interfaces. This is particularly important for visible interfaces, but can also be beneficial to developers for hidden and internal interfaces.

- The existence of supposedly **hidden interfaces** should be truly invisible — except when they are explicitly needed (as in the case of debugging, integration, and remediation).

- Interface arguments (and their symbolic names or other identifiers) should reflect **strong typing** of objects to minimize errors arising from type mismatching.

- Certain interfaces may benefit from **self-defining arguments,** either as an alternative option or as a standard, particularly in the absence of or as a possible alternative to sensible typing. Ideally, strong typing can be enforced through a combination of programming language constructs, programming style, precompiler and compiler discipline, programming discipline, and possibly hardware support.

- Some interfaces or combinations of interfaces represent state information that is inherently complex, for example a network manager dealing with traffic density, security, reliability, and so on. Certain graphical techniques can display in two or three dimensions various projections of multidimensional spaces.

- Interface design should be an **integral part of system design.** In particular, sufficient information regarding internal states should be maintained in a suitably accessible form — to facilitate whatever analysis needs to be done externally. If this is not done properly, the real-time analysis tasks will be extremely difficult and costly, if not impossible.
- **Assurance** is desired that the implementation at any particular interface is consistent with the interface requirements and specifications. Ideally, the implementation should meet the requirements, and do precisely what is specified — and no more. However, the concept of doing nothing else (Section 3.2) is an extremely difficult one to assure, because of typically incomplete specifications and possible hidden side effects. (See Lillibridge's dissertation [159] for a discussion of the spectrum from opaque to transparent types.)
- **Composability of perspicuity** is also a desirable characteristic. If two modules A and B satisfy certain perspicuity criteria, it will not in general follow that the composition of A and B will also follow those criteria — for example, because of emergent properties or various types of negative interactions. Similar lack of composability is common with respect to security properties.

Desired properties of specifications, architectures, and implementations are considered in subsequent sections.

5.2.4 Basic Approaches

There are several different approaches to increasing perspicuity. Ideally, a combination of some of the following might be most effective, but each by itself can sometimes be helpful.

First, consider proactive efforts. Ideally, it would be most appropriate to develop new systems that satisfy all of the above desirable characteristics — and much more. However, suppose you have an existing system that fails to satisfy these characteristics, or is in some ways difficult to understand. Let us assume that you have identified an interface that is seriously confusing.

- **What can be done proactively?**
  - If source code is available and is modifiable, fix it.
  - If source code is not available, decompile the object code, fix the decompiled source code, and recompile.
  - If decompiling and recompiling are not possible (e.g., if decompiling is proscribed by licensing constraints, or if the code is so extensively obfuscated to hinder conventional reverse-engineering tools, or if no compiler is available), then patch the object code.

For the most part in this study, we assume that source code is available. However, we also include some approaches that apply to object code when source code may or may not be available on the fly.

Analytic efforts at enhancing perspicuity of software interfaces can also be useful even if they do not require modification of either the source code or the object code implementing those interfaces.

- **What else can be done?**

– Create a general environment within which code execution can be embedded that significantly enhances understandability of control flow. Interactive debuggers and Interlisp come to mind as examples.
– Create a wrapper for a specific interface, encompassing inputs, outputs, and all exception information that must be visible to the interface user, but otherwise hiding anything that does not need to be visible. If sufficient internal state information can be gleaned from the interface, the wrapper might have a chance of cognitively augmenting understandability. Eunice (which was essentially a Tenex/TOPS-20 command emulator for Unix) comes to mind.
– Assemble a collection of analysis tools that can statically and dynamically analyze the behavior of programs in the specific machine code language. In certain cases, input-output experiments can derive sufficient information about the internal structure to enable an external fix to be created (for example, along the lines begun in 1956 by Ed Moore's Gedanken experiments [241]). On the other hand, black-box testing is inherently incomplete in almost all systems with nontrivial state spaces. Thus, source-code availability is vastly preferable.

5.2.5 Perspicuity Based on Behavioral Specifications

At the IBM Almaden Institute conference on human interfaces in autonomic systems, on June 18, 2003, Daniel M. Russell stressed the importance of shared experience between users and system developers. The following speaker then continued that chain of thought:

> *People and systems are not separate, but are interwoven into a distributed system that performs cognitive work in context.* David D. Woods

An enormous burden thus rests on the human interfaces. As noted in Section 5.2.1, perspicuous interfaces offer their greatest advantage when something has gone wrong, and the system is not working as intended. To really gain leverage from perspicuous interfaces, we need three primary areas of support:

- **Behavioral specifications in interfaces.** Most programming languages support type information only in interfaces. Furthermore, the type systems in use in popular modern languages are insufficient to express rich behavioral properties of code. (Recall that all of logic can be cast into type theory. Such type systems are generally undecidable, and require far greater expertise in type theory than possessed by most software engineers. As such, they are of theoretical interest rather than practical importance today. However, further research could be helpful.)

- **Debuggers that can work with behavorial specifications** (e.g., dynamically verifying that all invariants hold). Ideally, we should be able to query a module to determine whether or not it is internally consistent. If there are bugs inside a single module, that's valuable information – particularly if modules have been written defensively, with proper attention to information hiding. Otherwise, we might assume that a module's client must have behaved "improperly" — i.e., not following specified rules for using the interface. (Model checking could help here as well.)

- **Language support** and other tools for multiple views of modules. Debuggers and other code analysis tools often need access to the internal representation of data structures, among other implementation artifacts that are properly hidden from module clients. For example, Standard ML of New Jersey and its compilation manager CM, offer an implementation of the necessary functionality [50]. As an extension of these ideas, mechanisms for secure modular programming in Java are considered in [32]. Working through all of the security implications of such language constructs may not be trivial, but needs to be considered up front.

Together, these capabilities could revolutionize the system debugging experience, by combining tool support with machine-usable documentation of what is supposed to happen, enabling comparisons of theory and practice.

5.2.6 System Modularity, Visibility, Control, and Correctness

To establish a baseline for the investigation of system modularity as a basis for establishing the visibility of a system's structural and correctness properties, a brief analysis of prior art was performed by Virgil Gligor, resulting in an appraisal of which methodologies and tools have and have not been effective in defining and analyzing system modularity in the past and to what extent. That analysis is the basis for Appendix B of this report.

Gligor's analysis investigates the following topics related to modular system structures:

1. A generally accepted definition of a module (and of module instances such as subsystem, submodule, service, layer, and type manager)

2. The separation of module interface from the module implementation

3. Replacement independence property of modules

4. Structural relations among modules (e.g., the `contains` and `uses` relations)

5. The correctness dependencies among modules and their manifestation as causal relations among module interfaces (e.g., "service," "data," and "environment", along with other dependencies) that could lead to some sort of calculus of dependencies.

Gligor's analysis also presents the relationships between module definition and its packaging within a programming and configuration management system, and outlines measures (i.e., metrics) of modularity based on the extent of replacement independence and the extent of global variable use, as well as measures of module packaging defects.

The intent of this analysis is to identify pragmatic tools and techniques for modularity analysis that can be used in practice. Of particular interest are tools that can be used to produce tangible results in the short term and that can be extended to produce incrementally more complex dependency analyses in the future.

Virgil Gligor notes that Butler Lampson [199] argues that module reusability has failed and will continue to fail, and that "only giant modules will survive." If we believe Butler's arguments (and they are usually hard to dismiss), this means that "visibility into giants" is more important than ever. [Thanks to Virgil Gligor for that gem.]

5.3 Perspicuity through Synthesis

We summarize here the main concepts and issues relating to system architecture, software engineering, program languages, and operational concerns. However, no one of those areas is sufficient for ensuring adequate perspicuity, security, reliability, and so on. Indeed, all of these areas should be important contributors to the overall approach.

From the synthesis perspective, there are two different manifestations of perspicuity: (1) making interfaces understandable when they are to be used under normal operation, and (2) making the handling of exceptional conditions understandable when remediation is required (e.g., recovery, reconfiguration, debugging, aggressive responses). Both of these are considered, although the most significant payoffs may relate to the second case. Note that perspicuity can also be greatly aided during development by the appropriate use of static analysis tools.

5.3.1 System Architecture

Issues: Hardware protection and domain isolation, software abstraction, modularity, encapsulation, objects, types, object naming and search strategies, multiprogramming, processes, domains, threads, context changes, concurrency, interprocess communication, multiprocessing, interprocessor communication, networking, wrappers, and so on.

Useful historical examples: Two system architectures are noted in which great emphasis was devoted to interface design within a hierarchical structure.

- The **Multics** development emphasized novel hardware and software. Its development was carefully reviewed, iterated, and approved, before any code could be written — with attention to detailed software module specifications, analysis of dependencies on other modules, and specific interfaces. Some critical modules were rewritten as many as ten times, each version being reviewed before coding, and again before integration. (Multics-related principles are noted in Sections 2.3 and 2.4; its architecture is considered in Sections 4.3 and 4.4.)

  Several aspects of Multics are particularly relevant to this discussion of perspicuous interfaces. The hardware-enforced hierarchical ring/domain separation meant that much greater attention had to be paid to the specific interfaces. Symbolic naming of objects (such as virtual-memory segments and virtualized input-output streams) plus the invisibility of paged memory implied that machine addresses were never visible to programmers. Dynamic linking of symbolic file names and dynamic paging provided an abstraction of virtual memory that completely hid physical locations in memory and secondary storage media (as well as the associative memory). There were strict interface standards for arguments and formats. A constrained subset (EPL) of the full PL/I language enabled almost all programming to be done in a higher-level language. Programming styles were rigidly enforced, and changed for the better as the compiler improved — enabling certain ugly language primitives to be avoided entirely. The stack discipline inherently avoided stack buffer overflows by making the out-of-frame stack elements nonexecutable, and reversing the direction of stack growth (e.g., see [188, 189]). Each of these concepts contributed to the abstraction, encapsulation, and cleanliness of the interfaces.

- SRI's **Provably Secure Operating System** (PSOS) design was based on a hierarchically layered object-oriented architecture with special attention to unified interface design at each layer, largely based on the pervasive use of strongly typed tagged objects and formal specifications for each module at each layer. The specifications provided an essentially complete, self-contained, and relatively easily understood description of every interface, including formal definitions of all arguments, inputs, outputs, and exception conditions. The abstract implementations relating the functions at each layer to those of lower layers added to the perspicuity of each interface, providing an intuitive high-level description of each function in terms of lower-layer functions — despite the complexity of the overall system. For example, see [120, 268, 269].

Other systems also pursued various aspects that addressed the importance of proactive interface design — for example, some other capability-based architectures, Secure Computing Corp.'s strongly typed systems, and to some extent others such as SE-Linux, Plan-9, and some of the multilevel-secure kernels.

The following concepts are relevant to the development of perspicuous interfaces.

- **Architectural structure.** System structure is layered, from hardware to operating systems to applications to user programs, with abstraction and encapsulation at each usable layer interface. At different layers, usable interfaces include microcode, instructions, internal operating system primitives restricted to privileged software, operating system commands, application primitives, and so on. Thoughtful layering, abstraction, and encapsulation can greatly enhance perspicuity, without necessarily diminishing performance.

- **System interfaces.** Each of the system layers has potentially multiple interfaces, including operating systems, user commands and other visible interfaces, kernelized architectures, domain isolation, process separation, authentication, access controls (some of which are clearly not perspicuous), accountability, administrative interfaces and other protected internal interfaces, concurrency management (deadlocks, race conditions, synchrony-dependent security flaws), device drivers, stack disciplines, object naming, shared library management, and so on.

- **Application interfaces.** Even if the underlying operating system interfaces are not user-friendly, application interfaces can be made so — to the extent that the appropriate application state information is adequate to explain behavior relevant to the application interface, and to the extent to which the operating system is hidden.

- **Interdependencies.** Dependency analysis among system specifications and detailed architectures in the design stage can greatly enhance the perspicuity of the resulting system interfaces, allowing simplification or removal of unwanted dependencies, and enabling documentation to accurately characterize the remaining dependencies.

- **Wrappers.** The introduction of a wrapper can mask the complexities of a particularly gnarly interface. However, it can also easily provide a false sense of perspicuity *and* a false sense of security. Perspicuity is reduced if (for example) vital details are hidden that are needed to determine the state of the implementation underlying an interface. Security is reduced (for example) if the wrapper fails to mask security flaws, or creates new flaws, or if its supposed mediation can be compromised from within or below.

- **Multilevel security and integrity.** If an environment is expected to enforce multilevel security, the architecture and all of its interfaces should mask the existence of all information at higher security levels. If an environment is to enforce multilevel integrity (as in the Biba model), the architecture and all of its interfaces should prevent — and thereby both mask and prevent — dependence on all activities at lower integrity levels. If designed properly (e.g., [308], these constraints can greatly increase the perspicuity of the various interfaces, although possibly complicating the real-time analysis of poorly designed applications. These issues are considered further in Section 5.3.6.

- **Networking issues.** Perspicuity issues arise extensively with respect to almost every aspect of networking, including reliability, security, integrity, availability, fault tolerance, survivability, connectivity, specific protocols, routing, stability of cooperating parallel executions, concurrency, and so on. Hiding unnecessary details is of paramount importance, but not suppressing error conditions without properly handling them is a critical need.

Relevance for perspicuity: All of these issues can seriously affect interface perspicuity.

5.3.2 Software Engineering

Unfortunately, "software engineering" is a term applied to an art form, not to an engineering discipline. Nevertheless, there are many principles (such as those in Chapter 2) of sound architectures, good software engineering, and good development practice, which — if they were followed wisely — can result in systems with much greater security, reliability, and so on, much greater assurance that those properties are indeed satisfied statically, and much greater perspicuity when something goes wrong.

Issues: architecture, distributed systems, real-time systems, requirements, specification, software development methodologies, abstract implementations, composability, abstraction, modularity, encapsulation, information hiding, uniform handling of objects, object-oriented approaches, development practices, integration, debugging, testing, modeling, simulation, fault injection, formal methods for specification and analysis of functional and nonfunctional properties, formal verification and model checking, performance analysis, tools for static and dynamic analysis, software process technology, Clean Rooms, Extreme Programming, and so on. Development environments, component technologies, and related approaches such as the Common Object Request Broker Architecture (CORBA), CORBA Component Model, the Component Object Model (COM), DCOM, ActiveX, Enterprise Java Beans (EJB), Java Remote Method Invocation (RMI), and so on.

For example, CORBA provides some basic help in dealing with the interface definitions of proprietary closed-source components without having access to the source code. CORBA defined the Interface Definition Language (IDL) as a method to provide language-independent interface definitions. IDL types are then mapped into corresponding types in each language; there are standard mappings for some languages (C++, Java, Smalltalk). While greatly aiding cross-language interoperability, to date, it has not been widely applied to COTS software. (Note: Netscape based much of its architecture in the mid-to-late 1990s on the goal of being a "platform" on CORBA. There are rumors that a good bit of custom, in-house software in large corporations uses CORBA.) In the open-source world, its greatest success has been in the GNOME project. Like other existing technologies, IDL does not support behavorial specifications. While the CORBA folks discuss using IDL to structure the interfaces of a monolithic program, this does not appear to be very popular. CORBA's success, rather, has been in providing object-oriented RPC services, where IDL is used as the RPC specification language.

Relevance for perspicuity: All of these issues can seriously affect interface perspicuity. In particular, bad software engineering practice can result in systems that are extremely difficult to understand, at all layers of abstraction (if there are any!). On the other hand, intelligently applied good software engineering practice can greatly enhance perspicuity, particularly for software for which human interface design is an integral part of the system architecture. However, the best programming analysis tools can not overcome inherently bad architectures, bad software engineering practice, and sloppy testing.

5.3.3 Programming Languages and Compilers

Issues. We begin with a brief enumeration of the most relevant issues that affect interface perspicuity. (Some of the items — particularly those relating to programming languages — are merely collections of thoughts for further discussion.)

- **Language issues** include modularity, abstraction, encapsulation, objects, types, strong typing, dynamic linking, tasks, threads, sharing of resources (files, address spaces, objects, libraries), intertask, interprocess and interuser communication, garbage collection, stack disciplines, executable specifications, analysis of source code versus analysis of object code, comments and interpretable annotations, and so on.

- **Compiler issues** include overzealous compiler optimization that obfuscates the analysis that can be done from the interfaces, especially if it eliminates security boundaries such as protection boundaries and reliability measures such as intentional redundancy. There is a need for programming language features that can prevent such overzealous optimization. In addition, tools for dependency analysis can also identify interfaces in which perspicuity is likely to be poor.

- **Execution issues** include execution environments, integrity of virtual-machine monitors, runtime support (libraries, search strategies), hand-tinkered byte-code (in the case of Java), and implications of decompilers where source code is not available.

- **Programming language characteristics.** Programming languages that have many bells and whistles and attempt to be all things to all programmers (Ada comes to mind) tend to have too many opportunities for introducing program flaws. On the other hand, extremely simple languages (Basic) seem to introduce too many opportunities for learning bad programming practice. Functional, declarative, applicative, constraint, object-oriented, aspect-oriented, rewrite-logic languages, and so on all have their own benefits and quirks. Is it time for a new programming language that really makes it difficult to write bad code — for example, inherently avoid buffer overflows and other characteristic flaws? Perhaps we just need better education stressing many of the concepts in this report.

- **Mostly positive examples.** PL/I (tasks, stacks, explicit exceptions), Modula 3, Java and JVM (threads, re-entrant monitors), CCured, Occam, CSP, Concurrent Pascal, Standard ML [239], Standard ML of New Jersey [17], Extended ML, Concurrent programming in ML [314], Eiffel, Common Lisp, typed assembly language and corresponding assemblers.

- **Mostly negative examples.** Untyped languages, environments that make characteristic flaws (such as buffer overflows) too easy, lack of bounds checks and argument validation, wild pointers, poor exception handling, lack of finalization, unsound concurrency primitives, etc. C, COBOL ALTER verb, GOTOs, ComeFrom(!).

- **Mixed examples.** C++ (use with STL helps somewhat in overcoming its complexity), C# (C# types are similar to those in COM), Ada (rendezvous), and so on.

- **Other examples.** Eiffel's method of signatures including pre- and post-conditions overriding inherent pre- and post- (unlike Java).

- **Concurrency.** Primitives for atomic transactions, interprocess communication, shared-memory locks, precedence protocols, consistent finalization, automated rollback and recovery, and so on.

Here are several guidelines for increasing perspicuity through good program languages and compiler-related tools, as well as good programming practice.

- **Choice of programming language.** Given any particular development effort or subsequent redevelopment, one of the most important decisions may be the choice of suitable programming languages and supporting tools.

- **Modularity.** As noted in Sections 5.3.1 and 5.3.2, well-chosen architectural modularity (e.g., abstraction and minimized interdependencies, particularly with careful encapsulation) can be very beneficial, as long as it is enforced by suitable programming languages and compiler-related tools that can aid in not compromising the architectural structure and interdependencies.

- **Concurrency.** Poorly implemented concurrency is particularly likely to diminish perspicuity in multiprogramming and multiprocessing systems, typically because of badly constructed primitives for sharing, locking, and isolating processes and objects. On the other hand, this tendency can be greatly diminished through well-conceived programming languages and suitable analysis tools (plus highly disciplined programming style).

- **Transactional programming.** In multiprogramming and multiprocessing environments, transactionally based programming has enormous advantages with respect to perspicuity, as well as robustness.

- **Static analysis.** Compilers that are augmented with static analysis tools can help significantly — if they go well beyond routine flow analyzers, dependency analyzers, and debuggers. However, compiler optimization can greatly reduce perspicuity, particularly if it destroys or masks the integrity of the architecture and the program structure.

- **Application-specific tools.** Several areas are particularly well suited to application-specific tools, such as detecting flaws in bounds checks, concurrency, and cryptographic protocols. For example, Static analysis tools for developing and implementing cryptographic protocols, such as BAN Logic [61, 4], Spi Calculus [2], and the recent work of Jon Millen and Grit Denker on CAPSL and MuCAPSL [99, 237] (see http://www.csl.sri.com/˜millen/capsl for further background) at SRI.

Analysis tools that can aid in determining the perspicuity of interfaces are considered in Section 5.4.

5.3.4 Administration and System Operation

Administrative-interface issues include ease of maintenance, autonomic system behavior and what happens when the autonomic mechanisms fail, self-diagnosing systems, configuration consistency analysis, and many other topics.

User-interface issues include ease of diagnosing system failures, ease of debugging application code, analysis tools, and so on. Of particular concern are the users who have critical responsibilities — for example, operators of SCADA systems and other critical infrastructure components, control systems, financial systems, and so on. In these cases, real-time monitoring and analysis for anomalous system behavior become part of the interface purview.

Relevance for perspicuity: Today's system adminstrator interfaces tend to put an enormous burden on the administrators. Simplistic would-be solutions that attempt to interpret and explain what has gone wrong are likely to be inadequate in critical situations that go beyond the low-hanging fruit.

5.3.5 No More and No Less

"What you see is what you get" might be considered as a basic mantra of perspicuity — especially if it is taken seriously enough and assuredly implies that what you get is *no more and no less* than what you see. (Recall this dictum at the beginning of Section 3.2, in the context of the effects of composition.) The extent of typical exceptions to *no more and no less* is astounding.

There are many examples of *more,* many of which can be very damaging: hidden side effects, Trojan horses, undocumented and unadvertised hardware instructions and software primitives (sometimes with powerful override abilities), lurking race conditions and deadly embraces, blue screens of death, frozen windows, misleading URLs (for example, a cyrillic o instead of a roman o, or a Zero in MICROS0FT waiting to take you somewhere else), and so on, ad infinitum. Les Lamport's definition of a distributed system noted in Chapter 1 suggests that what you might have expected to happen won't.

There are also various examples of *less,* many of which are likely to be frustrating or debilitating: expected resources that do not exist or are temporarily unavailable, such as URLs that point nowhere, even though they worked previously.

Perhaps the most insidious cases are those in which something more and something less both occur at the same time.

5.3.6 Multilevel Security and Capabilities

Several of the system architecture approaches in Chapter 4 provide elegant ways of achieving *What you see is exactly what you get:* multilevel-secure (MLS) systems and capability-based addressing.

In particular, if a multilevel-secure object is at a higher security level or in an inaccessible compartment to the would-be user, then the user simply is not supposed to know of the existence of that object; any attempt to name it or list a directory in which it exists is greeted with a single relatively neutral undifferentiated standard exception condition such as "no such object" that conveys no information. Note that any exception condition indicator that provides a variety of possible context-dependent error messages is likely to be subject to exploitable covert channels through which information can be signaled.

Similarly in capability-based addressing, if a user does not have a proper capability for an object, that object is logically equivalent to being nonexistent.

In a sense, this is a very satisfactory goal in terms of perspicuity of naming and accessing system resources. On the other hand, if anything goes wrong, life can become quite complicated. From a user's perspective, everything that supposedly needs to be visible is visible — except when it isn't. From an application developer's perspective, simply plunking a legacy software system into the multilevel environment may cause the application to break, perhaps as a result of short-sighted assumptions in the legacy code or a configuration problem in the installation of that code. From a system administrator's perspective, access across security levels may be necessary to determine what went wrong — unless the system is well designed and single-level analysis tools can suffice. Otherwise, there is a risk of violating the MLS properties. Thus, MLS and capabilities can improve perspicuity when things go well, and can decrease it when things go wrong — unless the architecture and implementation are well conceived in the first place and the analysis tools are effective. Furthermore, in the absence of some sort of multilevel integrity in an MLS system, hidden dependencies on untrustworthy components can undermine the integrity of the MLS levels.

5.4 Perspicuity through Analysis

5.4.1 General Needs

From the dynamic analysis perspective, there are again two different manifestations of perspicuity: (1) using static and dynamic analysis of a given interface to provide greater understandability as the interface is being used under normal operation, and (2) interpreting real-time exceptional conditions and making them understandable contextually— for example, whenever remediation is urgently required (as in the cases of recovery, reconfiguration, debugging, and aggressive automatic responses). Both of these cases are considered in this section, where we seek to identify, characterize, and exploit analysis techniques for defining and analyzing system interfaces so that the behavior of the systems and the dependencies among those systems can be more easily understood and controlled.

This is a multidimensional challenge. Some of the dimension are outlined as follows.

- **Nature of software openness:** Potential differences in perspicuity arise between available source code (under any of various licensing agreements) and the general unavailability of proprietary closed-source code, with respect to operating systems, libraries, applications, execution environments, and analysis tools. *Although tools for reverse engineering are becoming more effective, analysis of source code is generally much easier than analysis of object code.*

- **Range of analysis methods:** Many analytic techniques are relevant, from completely formal models to more conventional ad-hoc approaches, with ample middle ground between the two extremes. Skills required to use the methods and tools vary widely, as do the costs and impact on delivery schedules. Similarly, techniques vary from static to dynamic, with hybrids in between. Static analysis is useful in development and in facilitating dynamic analysis; dynamic analysis is often essential when something goes wrong. (See Chapter 6.) *A combination of analysis methods is likely to be most effective.*

- **Range of possible modalities:** For example, static analyses can provide feedback toward improving perspicuity through augmentations of the interfaces and their implementation. Static techniques can improve understandability and maintainability without altering existing system interfaces. Static analyses can contribute directly to more effective dynamic analyses. Dynamic analyses can facilitate rapid recovery, reconfiguration, restoration of service, and so on. *There are many possibilities.*

- **Possible results:** There are many possible by-products of perspicuity-increasing analyses. *The results may include enhanced operational security, reliability, survivability, and satisfaction of real-time performance requirements, as well as easier system adminstration.*

In general, it is advantageous to address the problem of interface perspicuity up front, and then consistently follow through. This suggests an approach that encompasses the entire development cycle and operations, which can make the analysis challenges much more accessible.

- **Establish clear requirements** for what is desired, including with respect to interface perspicuity.

- **Provide unambiguous behavioral specifications** for software components, paying particular attention to the interrelationships among components at each layer and the interfaces. Develop tools for analyzing these specifications.

- **Develop tools for identifying and analyzing inconsistencies between behavioral specifications and actual behavior,** for example, resulting from failures or misuse.

- **Develop analysis of implementations** that use soundly based (e.g., strongly typed and well structured) languages that are more amenable to analysis. Develop tools for analyzing the resulting software, statically and dynamically.

- **Develop tools** for dynamically analyzing behavior of systems in execution and would-be operational responses of system administrators.

5.4.2 Formal Methods

Issues: Methods and tools for demonstrating consistency of specifications with requirements, and consistency of code with specifications; formal verification and model checking; analysis tools for detecting characteristic security flaws, buffer overflows, and so on.

Examples: HDM hierarchical abstraction, formal specifications, state mapping functions, and abstract implementations; PVS and PVS interpretations; CCS, π-calculus, and so on.

Of particular recent interest are Drew Dean's PhD thesis [95], the Wagner-Dean paper [98] on static analysis of C source code, the work of Giffin, Jha, and Miller [130] on analyzing binaries for mobile code (extending the Wagner-Dean approach), and Hao Chen's effort [75, 77, 78] at formal model checking to search for characteristic flaws. (Of course, it is much easier to do analysis on source code, if it is available.) Also, see Mitchell and Plotkin [240] for a highly readable paper on theoretical foundations of abstract data types with existential types; it is of particular interest to type theorists. See Chapter 6.

5.4.3 Ad-Hoc Methods

Issues: Informal methods and tools for testing for the inconsistency of specifications with requirements and the inconsistency of code with specifications; other tools.

5.4.4 Hybrid Approaches

Purely formal tools tend to be difficult to use for ordinary mortals. Purely ad-hoc tools are limited in what they can achieve. Semiformal tools may provide a bridge between these two approaches. Examples include formally based testing (e.g., mechanically deriving test conditions) and machine-assisted code inspections.

5.4.5 Inadequacies of Existing Techniques

Some of the above existing techniques can have significant effect in the near-term future, if applied wisely. However, in the longer-term future, those techniques are not nearly adequate. Thus, in this section we consider several areas in which there are serious gaps in existing approaches.

Many problems are made worse by a lack of perspicuity:

- **Flaws and bugs:** Certain characteristic design flaws and software bugs (e.g., [260], Chapter 3) seem to recur pervasively, including security- and concurrency-related flaws and failures.

- **Interdependencies:** Complex interdependencies typically exist among different design entities, different source-code modules, and different object-code components, especially in poorly architected and poorly implemented systems.

- **System administration:** Enormous complexity typically arises in system administration, partly because of complex systems and partly because of bad interface design. This tends to induce configuration errors, upgrade inconsistencies, operational oversimplifications, and so on.

- **Proprietary bloatware:** Proprietary code is often an obstacle all by itself. Bloated proprietary code is of course generally even less perspicuous. Exigencies of backward compatibility with nonperspicuous legacy systems certainly do not help.

5.5 Pragmatics

5.5.1 Illustrative Worked Examples

We foresee various possibilities for something that can be described conceptually without having to do much implementation, or whose implementation could be outlined and be pursued in detail. Some of these examples can demonstrably enhance perspicuity both statically and dynamically. It might also be possible to characterize some measures of perspicuity that could be analytically determined, although this is deemed less likely and probably less realistic.

One possible overarching approach is the following. Given a combination of specifications, source code, and perhaps some knowledge of the possible operating environments, statically analyze them and transform them into a body of knowledge that can be interrogated dynamically, for example, when an environment is under stress. A combination of dynamically interpreted pre- and post-conditions could then directly produce analysis results that would facilitate the understanding of attacks and malfunctions, based on which conditions fail. Such an approach would provide help in recommending autonomic responses and human-aided responses, as appropriate. Note that this is not really a new concept. For example, the ESS Number 2 telephone switching systems had a diagnostic dictionary that covered almost all possible failure modes and suggested appropriate remedies. However, in the context of more modern programming language and operating system technologies, such an approach could now be significantly more effective — albeit significantly more complicated.

Several specific examples come to mind as candidates for worked examples.

- **System administrator example.** Sketch of an environment for system administrators that could draw on the analysis results of the tools under discussion here, for example, for Linux or a BSD system.

- **TCP/IP multilayer example.** At multiple layers, characterize the TCP/IP specifications and source code (e.g., [382]), and develop a framework within which effective dynamic analysis can be carried out largely automatically.

- **MLS example.** Develop a prototype multilevel secure environment (for example, based on the Proctor-Neumann architecture [308] and other concepts in Chapter 4, including possibly with some aspects of multilevel security and multilevel integrity) in which facile perspicuity is a major design requirement. In particular, such a development should pay careful attention to increasing the simplicity of subsequent application development and dynamic analysis tools that facilitate recovery from errors, without compromising the desired MLS (and MLI) properties.

- **Transactional example.** Specify and sketch the implementation of a small but realistic application in a transactional system and in a nontransactional system, and compare the two under a variety of circumstances — such as aborting a process in midstream, disabling an input stream, clobbering a temporary file, or performing a security attack.
- **Model-checking example.** Analysis of a system or subsystem, using available tools (including, for example, extensions of the Chen-Wagner model-checking approach) and identifying models that are missing but that could be added with relative ease and effectiveness.

5.5.2 Contemplation of a Specific Example

When we went looking for examples where behavioral specifications would be useful, the BSD TCP/IP stack seemed like a logical place to start: not only is the software open-source, but there is excellent documentation as well [229, 382]. Unfortunately, this plan did not succeed as originally hoped. Our first idea was to examine the implementation of the Address Resolution Protocol (ARP). Up through 4.3BSD, the ARP implementation was a small module with a simple interface to the rest of the TCP/IP stack. In 4.4BSD, a new, generalized routing table structure that integrated ARP was introduced. The ARP implementation no longer has a simple, clean interface to the rest of the kernel — it is now part and parcel of the routing code, a much larger and more complicated piece of the TCP/IP stack. (Of course, it is conceptually nice to deal with ARP-resolved Ethernet addresses in the routing framework, and eliminate the special handling of Ethernet addresses for machines on the local network.)

Our next target was the UDP implementation. UDP is a nice simple protocol, and would appear to be an ideal example. The networking code in the kernel uses an object-oriented design similar to that of the file system code, although the actual implementation is in plain C. The implementation combines both a message-passing style, à la naïve objects in Scheme, and a record of functions style more similar to C++. The message-passing style is used on output, and the record of functions style on input. With better language support, these paradigms could result in an extremely clean implementation, but with C requiring manual implementation of all the details, some generally difficult layering issues explicitly raise their ugly heads.

On output, the handoff from the socket layer occurs to the `udp_usrreq` function, which takes as arguments a socket, a command (i.e., message), and three `mbuf` chains: the data to be sent, the address to send it to, and some control information that is not used by UDP and will not be discussed further. If the command is `PRU_SEND`, then `udp_output` is called. In `udp_output` is where things start to get ugly, making a behavioral specification less elegant than one would desire: either the socket has been connected to a destination (Yes, this makes sense for UDP!), or a destination address has been supplied — but not both. The code, most unfortunately, knows details about *all* of the data structures, and peeks around inside them to enforce this addressing invariant. With better language support, including either method overloading on argument type, as in Java, or full multiple-dispatch, as in CLOS, this could be very elegant: whether or not a socket is connected to a destination, as well as whether or not a destination address is supplied, could easily be encoded in the type system. Then, there would be four separate implementations, three of which simply signal an error, with the fourth function prepending the UDP header, generating the UDP checksum, and eventually calling `ip_output`. The main implementation would not need explicit code to check the addressing invariant, as everything would be guaranteed correct by the programming language.

On input, things are much simpler. The code checks various validity conditions on the input packet, and assuming the packet is valid, then checks whether the packet is destined for a unicast or broad/multicast address. If the packet is destined for a unicast address, the code searches for a socket to deliver the packet to. Assuming that an appropriate socket is found, the data is appended to its receive queue, and the process is woken up. For broad/multicast packets, the data can be delivered to more than one socket, and the appropriate process(es) are woken up. If no socket is found, an ICMP error packet is sent back to the source of the packet.

5.6 Conclusions

This chapter is perhaps the most speculative in the report, based more on hopes for the future that are less supported by the past than was the case regarding the chapters on principles, composability, and architectures — all of which have long histories in the research and development communities. Interface architectures have seemingly been neglected, relegated to an afterthought of system design and implementation.

Chapter 6

Assurance

Synopsis

Even if requirements and architectures have been created composably and with serious observance of the most important principles, questions must be considered as to the trustworthiness of the resulting systems and their uses in applications. However, such analysis can be extremely difficult unless assurance has been an integral consideration throughout the development.

Thus far, we have considered how to achieve principled composable architectures and to informally provide integrity of an architecture and its implementation throughout the system development process, in attempting to develop, configure, and maintain trustworthy systems and networks. In this chapter, we consider assurance aspects associated with the development process and with its artifacts and end products. We seek a collection of assurance techniques and measures of assurance that can be associated with requirements, specifications, architectures, detailed software designs, specifications, implementations, maintenance, and operation, as appropriate.

6.1 Introduction

> *Regarding trustworthiness of critical systems, assurance is in the eye of the beholder. However, it is better to depend on systems worthy of being trusted rather than to be beholden to seriously flawed software and unknown components.* PGN

We seek to achieve trustworthy systems and networks, with some demonstrably sound measures of assurance — that is, rigorously addressing the question of how worthy really is the intended trustworthiness. Measures of assurance can be sought in a variety of ways, throughout the development cycle — and thereafter as well. For example, they might involve analyses applied to requirements, architectures and detailed system designs of operating system and application software, compilers, hardware, and operational practices. With respect to software developments, thorough formal analyses throughout the development cycle can provide some significant levels of assurance, although less formal techniques such as code inspection, testing, and red-teaming are complementary techniques that can also be very useful. Generally much less satisfying if not unworthy from a serious assurance point of view are measures of institutional goodness (as in the Capability Maturity Model) and individual programmer competence (as in certification of software engineers). Overall, no one assurance technique is adequate by itself; each — including those that are formally based — has inherent limitations that must be recognized and surmounted.

Perhaps the most important conclusion of this report in our efforts to attain sound and robust systems and networks is that the assurance associated with trustworthiness must be a pervasive and integral part of the development cycle and the subsequent operational use and long-term evolution of the resulting systems and networks. We repeat this conclusion emphatically, referring to it as the notion of **Pervasively Integrated Assurance (PIA)**.

Attaining some nontrivial measures of assurance is seemingly a labor-intensive process, but then so is conventional software development — including testing, debugging, integration, red-teaming, maintenance, and evolution. Ideally, assurance techniques should be incorporated into existing tools for software and hardware development. Furthermore, new tools for enhancing assurance should also be added to the development process. On the other hand, there are grave dangers in believing in the infallibility of development tools. Once again, we must depend on the intelligence, training, and experience of our system architects, designers, implementers, application system operators, administrators, and — in many cases — end users themselves.

Typically, there are enormous benefits from techniques that can be applied upfront in the development process, such as formal specifications for critical requirements, principled architectures, and formal or semiformal design specifications. It is clearly preferable to prevent flaws early that would otherwise be detected only later on in the development. However, there are many flaws that cannot be detected early — for example, those introduced during implementation, debugging, and maintenance that can nullify earlier assurance techniques. Consequently, assurance must be a distributed and temporal concept throughout development, maintenance, and operation, where constituent assurance techniques and isolated analyses must themselves be consistent, composable, and carefully coordinated. For example, careful documentation, disciplined development methodologies, coding standards, and thoughtful code inspection all have a place in helping increase assurance — as well as having secondary effects such as reducing downstream remediation costs, and improving interoperability, system flexibility, and maintainability. However, when it comes to providing meaningful assurance, the usual dictum applies: There are no easy answers.

6.2 Foundations of Assurance

> *"If a program has not been specified, it cannot be incorrect; it can only be surprising."*
> W.D. Young, W.E. Boebert, and R.Y. Kain [386]

Several basic issues immediately come to mind in seeking increased assurance. (See also a report by Rushby [326] on providing assurance relating to reliability and safety in airborne systems, whose conclusions are also applicable here.)

- **Requirements analysis.** Developing a system based on incomplete, incorrect, or infeasible requirements is clearly wasteful. Efforts to increase assurance are meaningful only when meaningful requirements have been established. However, having a well-defined relevant and extensive set of requirements is a very important beginning to increasing the assurance of the resulting systems. Detailed analyses of requirements (and particularly formal analyses where the requirements are formally defined) can be enormously beneficial.

- **Pervasively Integrated Assurance (PIA).** As noted above, techniques to provide assurance related to all relevant aspects of trustworthiness need to be an integral part of the development cycle and persistently invoked during operation, maintenance, administration, and long-term evolution.

- **Guarantees.** Although we can make carefully couched conclusions based on tightly constrained assumptions about paper designs and software, there can never be any absolute guarantees with respect to the behavior of computer-communication technology in actual use. No matter how careful we may be, we cannot anticipate all possible deleterious events — including hardware malfunctions, software flaws, human mistakes, malicious actions, and environmental disruptions. As a consequence, assurance techniques should never be used to create a sense of perfection. Instead, they should be used to delimit the imperfections and reason about the implications of what might go wrong.

- **Analyses of composability and specific compositions.** As noted in Section 3.1, composability is meaningful with respect to many entities, including requirements, specifications, protocols, implemented components, and proofs. Analysis of composability is important for each of these entities. Looking to a future in which special-purpose and general-purpose systems and applications might become routinely composable out of more-or-less compatible demonstrably trustworthy components, a new type of analysis tool would be highly desirable: analytically determining the composability (among other properties) of (ideally, well-designed and well-specified) software components — not just for the initial creation of a composed system, but also in subsequent reconfigurations, upgrades, and even dynamic installation of mobile code. This approach could (for example) take advantage of specifications and software formally shown to be consistent with those specifications, including descriptors relating to previously evaluated characteristics of the components — such as internal locks that might cause deadly embraces in certain contexts, assumptions regarding relative dependencies, identified interface limitations, and other attributes that might affect the compositionality. Ideally, this approach could then be used iteratively — for example, initially pairwise or n-wise, and then over successively wider scopes, possibly ascertaining obstacles to the desired compositions, or even potential failure modes that would suggest that a specific composition should not be permitted in certain types of environments because of the identified deficiencies. Other properties could also be included, such as dynamic trustworthiness, configuration stability, and operational factors. We realize that there are all sorts of limitations of such an approach, but even small steps forward could be very useful.

- **Analysis of dependencies.** In any would-be architecture, as well as in implementations, it is desirable to be able to identify explicitly all potential dependencies, and, in particular, all potentially adverse dependencies — that is, components or subsystems that rely on other entities that are less trustworthy or of unknown trustworthiness. It is also useful to identify cyclic dependencies, especially those that can result in indefinite looping. This would be especially useful in avoiding dependency problems in large software systems. (Of course, the use of structured architectures with modular encapsulation can also avoid such problems in the first place, rather than having to rectify them later.) Similar techniques apply to undesired dependencies among policies, specifications, and proofs. Also relevant here is the notion of guarded dependence and its intuitive relationship with multilevel integrity, considered in Section 3.4.

- **Detecting and eliminating vulnerabilities.** Assurance techniques in general, and formal methods in particular, are most fruitfully applied when they can dramatically improve the process of detecting and eliminating significant system flaws and vulnerabilities, and dramatically improve the quality of the resulting systems. To this end, it is appropriate to ascertain that 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 generally less effective. In particular, if the requirements are flawed and the specifications are flawed, then attempting to carry out code proofs that demonstrate consistency between code and specifications is of very questionable value. Once requirements are properly established, formal analyses and supporting tools can be of considerable value if they can identify certain types of flaws in designs, source code, and executables, whether or not they are formally based. In particular, formal techniques can be very useful in identifying flaws in authentication algorithms, access control mechanisms, network protocols, cryptographic embeddings, and trustworthiness in general.

- **Software and hardware consistency analysis.** We prefer here to avoid use of the term "correctness", and instead refer to more precise statements such as "consistency of specifications with respect to requirements" or "consistency of code with respect to specifications" (assuming that there are any specifications!). It is generally unwise to overemphasize the process of trying to prove "correctness" of software and hardware. **Correctness** is meaningful only with respect to a set of presumed conditions that must be met — and those stated conditions can themselves often be wrong or incomplete. (The quote at the beginning of this section is particularly pithy, and also applies to requirements for trustworthiness as well as specifications.) Besides, code correctness proofs are premature unless the requirements and the design are sensible and demonstrably sound. The same is true of hardware enchipments. Thus, efforts to increase the assurance with which system dependability can be attained by the implementation should be up-front concerns but not overly stressed at least until the continual discovery of new flaws in requirements and the design has dwindled considerably. However, at that point, code proofs and other forms of analyzing the implementation can be very valuable — especially if they are able to show the absence of nonspecified effects such as surreptitious Trojan horses or potential timing faults. In particular, there is an enormous potential for the use of formal methods in hardware implementations — for example, in specifications, mask layouts, and fabrication, and especially for critical coprocessors or chips such as cryptographic devices.

- **System-oriented analyses.** What is critically needed overall is a system-oriented view of assurance that coherently encompasses temporal beginning-to-end life-cycle issues as well as analyses across all layers of abstraction. Whether the analyses are linearly continuous (e.g., top to bottom, or bottom to top, or strictly beginning to end) or merely done piecewise and then joined together as appropriate is not so important as whether the separate analyses enable all of the pieces to fit together.

Continuing on the subject of composition (see the fourth previous item), horizontal (modular) composition and vertical (hierarchical abstraction) composition (discussed in Section 3.3) are both important subjects of analysis for the purpose of increasing assurance. Various past efforts are of considerable theoretical and practical interest, such as proving consistency within a successive refinement thread throughout the development effort, and proving consistency from top-level requirements through detailed design specifications to software implementation, and perhaps even subsequent operation and evolution, particularly when applied to large and complex systems. Similarly, when dealing with vertical and horizontal abstractions, efforts such as that of Robinson–Levitt [316] and the Boyer-Moore-Hunt-Young CLInc stack [244, 245] (including five papers, [41, 42, 170, 243, 385]) enable the functionality of higher-layer abstractions to be explicitly related to the functionality of lower-layer abstractions, iteratively across all layers as desired — for example, from applications to hardware — as well as the relationships among different modules at the same layer. 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 comprehensive analysis can be very compelling, even if labor intensive.

- **Transformations.** Considerable value can accrue from transformations that demonstrably preserve desired properties, such as maintaining the consistency of specifications with requirements despite changes to requirements, or maintaining the consistency of code to specifications despite the machinations of optimizing compilers. Being able to demonstrate that a transformation is property preserving can significantly simplify subsequent analysis. Transformations that result from nontrivial compositions, hierarchical layerings, and interpositions of mediators such as firewall systems and trusted guards are of considerable value. The transformational approach should also permit parametric architectural representations — for example, what simplifications or complications might result when a particular architecture undergoes a particular change in design, such as making a file server multilevel secure instead of having multiple single-level file servers. The effects of such changes should be formally derivable, if possible.

- **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 generally 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 badly overconstrained, or too low-level. Thus, considerable effort may be worth devoting to establishing an architecture that is amenable to selective uses of formal methods. However, there is also a danger that premature choices of methodology, approach, and programming languages will lock the development into a nonconstructive path. Thus, it is essential that all these factors be considered early in the system development process. (Historically important documents on formalism in software development include [58, 129, 190], and most recently [182].)

- **Development tools.** Various approaches to static analysis are considered in Section 6.6. Also of considerable interest here is the July 2004 special issue of ERCIM News [23], published by the European Research Consortium for Informatics and Mathematics. That issue is devoted to Automated Software Engineering, and includes sections on requirements, program understanding and architecture, testing, verification, aspects of language technology, configuration management and deployment, and models. (That issue also includes a section containing 16 contributions on R&D technology transfer.)

- **Research and development.** Consistent with the ability to apply formal methods to critical aspects of complex and critical systems, research must continue to explore the frontiers of the technology, and development must be carried out that employs formal methods constructively as suggested in these conclusions. Some specific suggestions for future R&D are given in Sections 8.2 and 8.3.

- **Composability within the evaluation process.** A remarkable paper by Paul Karger and Helmut Kurth [187] addresses the essential needs for cooperation and communication within various components of the evaluation process, particularly as it relates to the notion of Composite Evaluation within the Common Criteria. One major point of the paper is that the results of hardware evaluations must flow to the software developers and software evaluators, irrespective of proprietary constraints. The consequences of not doing this can be quite dire. This is a very important paper, and deserves considerable attention.

There have been many advances in assurance techniques, and particularly in formal methods, over the past thirty years. However, major successes are still awaited in the fruitful application of these methods. We conclude that, whereas considerable potential remains untapped for formal methods applied to security, we are now actually much closer to realizing that potential than previously. Many of the pieces of the puzzle — theory, methods, and tools — are now in place. It is unwise to put all your eggs in one basket (such as testing or penetrate-and-patch efforts). Thus, a more comprehensive combination of approaches is recommended, especially if the desired paradigm shifts are taken and if the considerations of the following section are observed.

6.3 Approaches to Increasing Assurance

Providing meaningful assurance of trustworthiness is itself a very complex problem, and needs to be spread out across the development process as well as into operational practice. Various approaches can be used in combination with one another to enhance assurance.

- **Assurance-enhancing principles.** Disciplined adherence to certain useful assurance-enhancing principles would be beneficial throughout system development. The notion of Pervasively Integrated Assurance is fundamental.

- **Sound requirements.** As we have noted already, anything that is done to increase the soundness, completeness, and specificity of the requirements would be very valuable. Assessments of the realistic implementability of the requirements long before any development is undertaken can have significant payoffs later in the development.

- **Sound system architectures.** Considerable assurance can result directly from an inherently sound architecture and analytic techniques for determining whether an architecture is capable of satisfying the given requirements. Such techniques can have enormous payoffs — as already noted, simplifying developments, reducing costs and overruns, and greatly increasing the likelihood of development success.

- **Sound algorithms and protocols.** Analysis can determine consistency with requirements for trustworthiness (security, reliability, and so on), with respect to both designs and implementations.

- **Analysis of subsystem compositions.** Chapter 3 characterizes some of the difficulties that need to be avoided in system composition (Section 3.2) and some of the desiderata for achieving predictable compositionality (Sections 3.3 and 3.4). Analytic techniques for increasing assurance can determine the existence or absence of unspecified side effects, as well as the satisfaction of higher-layer properties resulting from composition, with respect to both designs and implementations.

- **Analysis of properties.** Functional and nonfunctional properties of subsystems, systems, and networks are subject to analysis, at different layers of abstraction. For example, subsystems, operating systems, application packages, and entire enterprises can all be modeled and analyzed, formally and informally, with respect to designs and implementations. Hierarchical dependence analyses can uncover design flaws and implementation bugs and architectural inconsistencies in adverse trust relationships. Higher-layer properties — including emergent properties — can be analyzed hierarchically (e.g., as in the Robinson–Levitt approach [316] noted in Sections 3.4 and 6.2), relating lower- and higher-layer abstraction to one another, to permit analysis of systems in the large.

- **Software engineering.** Soundly based development methodologies can encompass techniques and tools for whatever measures of assurance are desired. Inherently sound programming languages that are formally based and about which properties can be proven are also helpful. Considerably improved assurance can result therefrom.

- **Code inspection, testing, and debugging.** The use of formally based tools can be highly beneficial for code inspection, testing, and debugging. Conventional uses of code inspection and testing are labor intensive and subjective, and very much dependent on the intelligence, education, training, and experience of the individuals involved; although those techniques can be aided by automated or semiautomated tools, there is always a risk that the weaknesses of the tools will prevail over the limitations of the people using those tools.

- **Red-teaming.** So-called **red teams** can also be used to increase assurance, if they can provide truly independent and objective assessments. (Beware of Accenture-Enronitis.) Although red teams have tended to be rather *ad hoc* in the past, the use of disciplined tool sets could actually be very helpful. The PIA concept noted in Section 6.1 suggests that the red-team concept be substantially broadened to include pervasive interactive assessments throughout. However, despite the desire for pervasive integration, it is important that the red team be independent of the development efforts — although still capable of proving valuable feedback. It is also important that the red team have direct access to project management, whether the concerns involve functional assurance, security assessments, quality assurance, and so on, to avoid the feedback being stonewalled.

- **Operational practice.** At present, system administration and operational practice are generally not thought of as amenable to high-assurance techniques. There are some potential opportunities for progress in this area.

Judicious use of formalisms and formal methods can add significantly to development and operation, but also can add complexity, delays, and cost overruns if not used wisely. Although formal models and formal specifications may seem to complicate the design process (with delays, increased costs, and greater intellectual demands), they can also substantively improve assurance, and also lead to earlier identification of problems that might otherwise be uncovered only in late stages of the development and use cycles. However, they need to be used with considerable care, primarily where they can accomplish things that design reviews, testing, and operational discipline cannot. In that errors in requirements formulation, design, and specification are perhaps the most difficult and costly to repair, formalisms can be particularly valuable in the early stages of development. Although some readers will consider assurance issues to be pie in the sky and unrealistic from the perspective of increased costs, project delays, and increased needs for education and training, the spectrum of assurance techniques does have something for everyone.

6.4 Formalizing System Design and Development

Historically, early examples of the use of formalism in system design and implementation are found in two SRI efforts during the 1970s. These rather early instances of uses of formal methods are reconsidered here for yet another visit because they represent some significant advances in the ability to analyze systems in the large that seem to have been otherwise ignored in recent years. (Please excuse a little duplication for contextual ease of reading.)

- **The Provably Secure Operating System (PSOS).** The PSOS architecture [120, 268, 269] — see Sections 2.6, 4.3, 4.4, and 5.3.1 — spanned layers of abstraction of hardware and operating system functionality that were extensively and precisely defined in a formal specification language. This approach enabled a complex hardware/software system to be represented as a horizontal and vertical composition of surprisingly simple but perspicuous formal specifications at each layer, with explicit mappings between the state spaces at different layers, and abstract programs that explicitly related the functionality at each particular layer to the functionality at lower layers. Various assurance properties were characterized and some proofs were outlined. The PSOS project developed and used SRI's Hierarchical Development Methodology (HDM) [119, 120, 268, 269, 316, 317, 318, 361], and its formal SPECIfication and Assertion Language (SPECIAL), which together in a unified way encompassed the writing of the specifications, state mappings, and abstract programs; it also provided the ability to carry out formal proofs of consistency between specifications and requirements — as well as formal proofs of consistency between code and specifications, if desired.

- **The Software-Implemented Fault-Tolerant System (SIFT).** The SIFT [232, 247, 378]) design and implementation abstraction hierarchy consisted of a model for hardware behavioral properties (seven Bendix BDX-930 avionics processors), Pascal code for the basic software, formal specifications for a real-time operating system (encompassing the scheduler, the 2-out-of-3 voter, dispatcher, buffer manager), a task activity model (dealing with system startup, broadcasting of results to other processors for voting, vote execution, and synchronization), a replication model (which produced the results of the 2-out-of-3 voting), and an I/O model that produced an overall indication that the current system configuration was safe). The hierachical analysis concluded that given the individual processor probability of failure of 10^{-5} per hour, the resulting seven-fold redundant system had a probability of failure of 10^{-10} per hour.

A general argument against such efforts seems to be that it is too difficult to deal with big-system issues, and much easier to focus on components. However, it is often the analysis of compositions and system integration that in the long run can be most revealing.

Incidentally, HDM's 1970s ability to analyze vertical compositions of hierarchical abstractions has been incorporated in SRI's PVS (beginning with version 3.0), in the form of PVS theory interpretations [278]. See http://pvs.csl.sri.com for PVS documentation, status, software downloads, FAQ, etc. See also http://fm.csl.sri.com for further background on SRI's formal methods work, including SAL (the Symbolic Analysis Laboratory, which includes three model checkers) and ICS (the Integrated Canonizer and Solver, a decision procedure). Symbolic analysis involves automated deduction on abstract models of systems couched in formal logic, and is the basis for much of CSL's formal methods work.

Some further early work on formal methods and verification applied to security is summarized in the proceedings of three VERkshops [275, 276, 205], from 1980, 1981, and 1985, respectively. (The titles of all of the papers in those three VERkshop proceedings are given in the appendix of [259].)

Considerable benefit can accrue from rigorous specifications — even if they are not formally checked, although clearly much better if they are. Specifications of what is and is not needed are generally more succinct than literal descriptions of how something should be implemented. Specifications can provide an abstraction between requirements and code that enable early identification of inconsistencies — between specifications and requirements, and between code and specifications. Furthermore, specifications can be more readable and understandable than code, especially if they can be shown to mirror the requirements explicitly early in the development process, before any code is written.

The long history of fault-tolerant computing has put significant effort on fault prevention (relative to whatever scope of faults was intended — from hardware to software faults to faults that included security misuse). Clearly, all of those assurance efforts relating to the avoidance of bad designs and bad implementations are relevant here, including the assurance that can result from inherently sound system and network architectures and good software-engineering practice.

With respect to the importance of programming languages in security, see Drew Dean's paper on The Impact of Programming Language Theory on Computer Security [96]. As a further useful reference, Chander, Dean, and Mitchell [69, 70] have some interesting recent work on formalizing the modeling and analysis of access-control lists, capabilities, and trust management.

6.5 Implementation Consistency with Design

The HDM approach noted in Section 6.4 is one methodology in which formal proofs could be carried out demonstrating the consistency of a software component with its formal specifications. The intent is that such proofs would be carried out only after proofs had shown that the specifications were consistent with the stated requirements (possibly subject to certain exceptions that would have to be tolerated or monitored, as in the case of unavoidable covert channels).

6.6 Static Code Analysis

Ideally, the up-front philosophy suggests that discipline embededded in the software development process can have considerable payoff. For example, programming languages that inherently enforce greater discipline would be very beneficial. Compilers and related pre- and post-processor tools that provide rigorous checking would also be useful. However, the integrity that can be provided by the best methodologies, programming languages, and compiler tools is potentially compromisible by people involved in design and implementation, debugging, integration, maintenance, and evolution.

Early efforts in the 1970s by Abbott [5] and the ISI team of Bisbey, Carlstedt, Hollingworth, and Popek [44, 45, 46, 67, 68, 165] attempted to identify a few characteristic flaws noted in Section 2.4 and to devise means of detecting their presence in source code. The conclusions at that time were generally rather discouraging, except in very constrained circumstances.

Contemporary analytic techniques and tools are much more promising. They are particularly appropriate for open-box source code, but of course also applicable to closed-box software — even if only by the proprietors. Examples include (among others), with varying degrees of effectiveness and coverage:

- Crispin Cowan's StackGuard (`http://immunix.org`)
- David Wagner's buffer overflow analyzer (`http://www.cs.berkeley.edu/~daw/papers/`)
- @Stake's L0pht security review analyzer slint
- Cigital's ITS4 function-call analyzer for C and C++ code (`http://www.cigital.com/its4/`)
- Ken Ashcraft and Dawson Engler's system-specific approach [20]
- Brian Chess's extended static checking [79]
- Purify
- Yuan Yu and Tom Rodeheffer's RaceTrack, for detecting race conditions in multi-threaded code (Microsoft Research)
- Hao Chen's MOPS (with some assistance from Dave Wagner and Drew Dean, whose earlier joint work [98, 371] provided a starting point); MOPS takes a formally based, approach to static code analysis (see Appendix A), in which formal models of undesirable vulnerability characteristics are the basis for formal model checking of the software, thus identifying software flaws.

There has also been some effort on formally based testing. (This work is particularly interesting when applied to hardware implementations.) However, the early results of Boyer, Elspas, and Levitt [57] suggest that formal testing is in some sense essentially equivalent to theorem proving in complexity. Nothing since that paper has fundamentally altered their conclusion, although formal derivation of test cases can be extremely effective in increasing the assurance that testing will cover a realistic span of cases. In particular, formal test-case generation has become increasingly popular in the past few years. (As just one example, see [49].)

6.7 Real-Time Code Analysis

There has been relatively little exploitation of formalism relating to real-time analysis in the past, but this area represents a potentially fertile ground for the future. One example might involve run-time checks derived from formally based analyses of potential vulnerabilities in source code, above and beyond what might take place in a compiler, or in a preprocessor — such as buffer-overflow checks and Trojan-horse scans that cannot be done prior to execution. Proof-carrying code [250] and checking of cryptographic integrity seals are two specific examples. Many other concepts remain to be considered.

6.8 Metrics for Assurance

In order to have any concrete measures of assurance, it is necessary to establish well-defined metrics against which requirements, architectures, specifications, software, tools, and operational practice can be measured. This is a very complicated area. We believe that it is unwise to do research on metrics for the sake of the metrics themselves, although it is important to establish parameterizable metrics with general applicability. The various metrics then need to be tailored specifically to the development stage in which they are invoked, and applied explicitly to those development efforts.

6.9 Assurance-Based Risk Reduction

The assurance techniques summarized in the previous sections of this chapter can have significant effects in reducing risks, particularly with respect to the extent to which critical system requirements are likely to be satisfied by system designs and implementations. These techniques may be applicable in many different ways, all of which are potentially relevant here. In particular, analysis at all development stages and all layers of abstraction within a development can contribute. (See Section 6.3.)

Several examples may help to illustrate how assurance techniques might be applied. In particular, we examine some of the cases summarized in Section 2.2 and Section 5.2.2, and consider what might have been done to prevent the effects that actually resulted. This is intended not as an exercise in *hindsight,* but rather as an explicit representation of what types of assurance might be applicable in future developments of a similar nature.

- **Human safety**
  - Design reviews and testing could have detected the missing interlock that allowed the Lauda Air thrust reverser to be deployed during flight, although more careful requirements, design, and implementation might have prevented the problem altogether.

  - Analytic comparison with topographic data might have revealed the erroneous course that caused the Air New Zealand crash into Mount Erebus. (In that case, a data error had been detected, but not yet fixed.)
  - Testing a medical device such as the Therac 25 for a race condition (which resulted from implementing the Therac 20 hardware interlock in software) or a heart pacemaker or defibrillator for electromagnetic interference would seem to be standard practice; yet these failure modes were not considered until a few people had died. Again, better attention to requirements, architecture, and good software engineering practice could have avoided the problems altogether. EMI problems clearly must be addressed through explicit requirements, threat models, and testing.

- **Reliability and availability**
  - The Patriot missile excessive clock drift could have been prevented through any of several assurance measures, for example, requirements that insisted that the systems should operate correctly if not rebooted for weeks, or alternatively, operational requirements that explicitly specified that the launch platforms had to be rebooted once a day. (On the other hand, it is surprising that a demonstrably more robust clock algorithm was not used, irrespective of the requirements. Apparently the implementers tacitly assumed the launch platform would be moved frequently — at least once a day — to reduce the likelihood of it being targeted by the enemy.)
  - The Yorktown Aegis missile cruiser disabled for 2.75 hours as a result of a divide-by-zero in an application could have been prevented by observing the shrink-wrap disclaimers (don't use this operating system in critical applications, and don't put engine controls inline with a system with poor survivability), or at least understanding their implications during requirement specification, development, and system certification. Fundamentally, the use of an operating system that cannot protect itself properly is unwise. The obvious answer is that we need better operating systems that adhere to the principles and structured architectures described in this report.
  - The backup computer synchronization problem on the very first Columbia Shuttle launch was apparently known to the designers, but not to the launch support team. The two-day delay could have been avoided by a simple retry that would have skipped over the one-in-64 known failure mode — if the documentation had noted this problem, or if the launch crew and the developers had been in closer contact. However, the synchronization problem could also have been prevented altogether by a more robust design, or explicitly diagnosed by a variety of analysis tools.

- **Avoidance of propagation effects.** The 1980 ARPANET collapse, the 1990 AT&T long-distance collapse, and numerous massive electrical power outages noted in Section 2.2 all share a common failure pattern — the propagation of a local failure into a system-wide outage. In each case, it had been widely believed by the developers and operators that single-point failures could not bring down the entire network. At the least, some detailed modeling of the interactions among components could have demonstrated the possibility of mutual contamination and propagating outages, enabled further analysis to determine what conditions might trigger such effects, and inspired resulting architectural changes necessary to prevent such problems. Some combination of fault-tree analysis and model checking would presumably have exhibited specific failure modes that could trigger such outages. (In the case of one of the extended System 7 telephone outages, detailed laboratory simulation environments were normally invoked, but because a particular change affected only a few lines of code, it was not thought to be necessary to recycle through the entire analysis process.)

- **Security.** The Risks archives are littered with security vulnerabilities and exploitations thereof. Buffer overflows, erroneous or missing bounds checks, and other annoying types of security flaws that seem to recur much too frequently are easily avoided with sensible system architectures and a variety of static and dynamic assurance techniques. Similarly, the insertion and execution of malicious code such as Trojan horses, Web- and e-mail-based executables, and so on need to be combatted by sound architectures and operational practice. Authentication problems can be avoided by eschewing the use of fixed passwords, instead using demonstrably robust cryptographically based authentication and analyses that demonstrate the security of their embedding into systems. Real-time misuse detection and response can also increase assurance of trustworthiness, especially with respect to integrity violations, insider misuse, denials of service, and so on.

- **Integrity of electronic election systems.** Election systems provide a paradigmatic example of the need for a wide range of requirements, including end-to-end security, system survivability, system integrity, data integrity, and voter privacy. Extensive use of formal methods, open standards, open testing, open evaluations, and — above all — the addition of cryptographic checksums on data and programs, as well as nonsubvertible voter-verified audit trails, could greatly increase the assurance of today's unauditable all-electronic election systems, thereby providing strong evidence that votes are processed correctly throughout. (Such assurance is almost totally absent in almost all of the currently used systems, and especially in the all-electronic systems.) In addition to demonstrating the integrity of the election software by itself, the architecture of the overall system should be such that formal proofs could provide assurances that no adverse compromises of the application software and the election data could occur as a result of manipulations of the underlying operating system or externally — for example, from dial-up lines, wireless communications, or networked connections. Furthermore, if such systems were composed out of components from independent vendors, it might be possible to overcome the potential untrustworthiness of one vendor's components — for example, having a separate system to provide a voter-verified nonspoofable recountable audit trail. Unfortunately, there is enormous resistance to such assurance from almost all of the commercial vendors.

- **Operational assurances.** More careful analysis of human interfaces would be particularly valuable with respect to system administrators, both statically (e.g., leading to warnings before permitting certain questionable upgrades and configuration changes) and dynamically (in anticipating needs for emergency response).

- **Sound user interfaces.** As noted in Section 5.2.2, numerous serious accidents have resulted from poorly designed human interfaces, as well as from misguided internal system interfaces. Assurance techniques analyzing human interfaces could perhaps have avoided many of the cases noted there. Testing should have caught the cross-wiring in maintenance of the British Midland 737, which caused the pilot to shut off the good engine rather than the failing one. Testing might have detected the missing interlock in the F-16 landing gear that allowed retraction on the ground. Sensible interface design might have proactively recognized the risk of using a GPS unit that reset the intended target after a battery change, but the designers of the weapons system might have anticipated the need for a warning. The heart-monitoring device should have had standards that prevented the developer from using a standard electrical wall-plug connector, and the certification process should have prevented that device from being available to the hospital. The Discovery laser-beam experiment failure (with the elevation of the target to be specified in feet, not miles) could easily have been avoided by a human interface that explicitly insisted on an input in feet, or that insisted on self-defining input definitions. Self-defining input types may seem like a low-assurance programming technique, but it would have avoided several past problems in the annals of the Risks Forum.

The above illustrative enumeration suggests that, among the wide variety of assurance techniques (some almost obvious, some more subtle), each potential system risk can benefit from the application of some subset of the total collection of approaches to increasing assurance. Establishment of sound requirements, sensible architectures, and good software development practice would undoubtedly avoid many of the problems discussed throughout this report, and could be significantly aided by formal or even semiformal requirements analysis, model-based design, model checking, formal test-case generation, static analysis, and so on. Of course, there is no one size that fits all; the particular techniques must be used in various coherent combinations, according to the circumstances, the development challenges and risks, and the competence of the developers and analysts. Once again, it is clear that there is a significant need for pervasively integrated assurance, throughout development and operation. However, the amount of resources and effort to be devoted to assurance needs to be commensurate with the overall long-term and nonlocal risks. Unfortunately, most risk assessments relating to how much effort to devote to assurance tend to be short-term and local. (The risks of short-sighted optimization are considered further in Section 7.1, and the importance of up-front efforts are discussed in Section 7.2.)

6.10 Conclusions on Assurance

Opportunities for seriously increasing the assurance associated with software development and system operations are abundant, but largely unfulfilled. Much greater commitment is needed to providing assurance of trustworthiness. Assurance techniques seem to have greater use and greater payoffs in hardware development than in software development, with heavier emphasis on the use of formalisms. However, assurance applied to operational practice lags far behind either hardware or software assurance.

The potential benefits of formal methods remain undiminished, particularly with respect to hardware and software, but perhaps also integrated into operational practice. The need for formal methods in the specification and analysis of critical systems and system components remains enormous. In the light of past events — including rampant system flaws and detected vulnerabilities, system failures, experienced penetrations, and flagrant system misuses — formal methods remain a potentially important part of the system development and assurance process. Their systematic use at appropriate places throughout the system life cycle can be extremely productive, if used wisely.

Recommendations for future research and development encompassing increased assurance for trustworthy systems and networks are discussed in Chapter 8.

Chapter 7

Practical Considerations

Synopsis

> *There's many a road 'twixt the need and the code.*
> (It's an especially rough road in the absence of requirements, design specifications, careful programming, sensible use of good development tools, documentation, and so on!)

The previous chapters pursue approaches that have significant potential to enable the development and operation of useful meaningfully trustworthy systems — if these approaches are applied wisely. This chapter considers various potential obstacles to the application of these approaches, and explores how they might be overcome. Some of the apparent obstacles are merely perceived problems, and can be readily avoided. Other potential obstacles present genuine concerns that can be circumvented with some degree of knowledge, experience, discipline, and commitment.

In this chapter, we address such topics as how an architecture can accommodate its relevant requirements (including requirements to be able to adapt to changing requirements!); whether inherently robust architectures are possible given today's mainstream hardware platforms and computer-communications infrastructures; the extent to which discipline can be effectively and pervasively introduced into the development process — for example, through methodologies, programming languages, and supporting tools; the relative effectiveness of various methodologies; problems peculiar to legacy systems; the practical applicability of formal methods; various alternative paradigms; management issues; relevant pros and cons of outsourcing and offshoring; and so on.

7.1 Risks of Short-Sighted Optimization

Many people (for example, system procurers, developers, implementers, and managers) continue to ignore the long-term implications of decisions made for short-term gains, often based on overly optimistic or fallacious assumptions. In principle, much greater benefits can result from far-sighted vision based on realistic assumptions. For example, serious environmental effects (including global warming, water and air pollution, pesticide toxicity, and adverse genetic engineering) are generally ignored in pursuit of short-term profits. However, conservation, alternative energy sources, and environmental protection appear more relevant when considered in the context of long-term costs and benefits. Similarly, the long-term consequences of dumbed-down education are typically ignored — such as diminution of scientific, engineering, and general technical expertise, poor system development practices, and many social consequences such as higher crime rates, increased reliance on incarceration, and so on. Governments tend to be besieged by intense short-sighted lobbying from special-interest groups. Insider financial manipulations have serious long-term economic effects. Research funding has been increasingly focusing on short-term returns, seemingly to the detriment of the future. Overall, short-sightedness is a widespread problem.

Conventional computer system development is a particularly frustrating example of this problem. Most system developers are unable or unwilling to confront life-cycle issues up front and in the large, although it should by now be obvious to experienced system developers that up-front investments can yield enormous benefits later in the life cycle. As described in earlier chapters, defining requirements carefully and wisely at the beginning of a development effort can greatly enhance the entire subsequent life cycle and reduce its costs. This process should ideally anticipate all essential requirements explicitly, including (for example) security, reliability, scalability, and relevant application-specific needs such as enterprise survivability, evolvability, maintainability, usability, and interoperability. Many such requirements are typically extremely difficult to satisfy once system development is far advanced, unless they have been included in early planning. Furthermore, requirements tend to change; thus, system architectures and interfaces should be designed to be relatively flaw-free and inherently adaptable without introducing further flaws. Insisting on principled software engineering (such as modular abstraction, encapsulation, and type safety), sensible use of sound programming languages, and use of appropriate support tools can significantly reduce the frequency of software bugs. All of these up-front investments can also reduce the subsequent costs of debugging, integration, system administration, and long-term evolution — if sensibly invoked. (Note that a few of the current crop of software development methodologies do address the entire software life cycle fairly comprehensively, such as the Unified Software Development Process (USDP) [174], whose three basic principles are use-case driven, architecture centric, and iterative and incremental; USDP is based on the Unified Modeling Language (UML).)

Although the potential fruitfulness of up-front efforts and long-term optimization is a decades-old concept, a fundamental question remains: Why has the sagest system development wisdom of the past half-century not been more widely and effectively used in practice? Would-be answers are very diverse, but generally unsatisfactory. These concepts are often ignored or poorly observed, for a variety of offered reasons — such as short-term profitability while ignoring the long-term; rush to market for competitive reasons; the forcing functions of legacy system compatibility; lack of commitment to quality, because developers can get away with it, and because customers either don't know any better or are not sufficiently organized to demand it; lack of liability concerns, because developers are not accountable (shrink-wrap license agreements typically waiver all liability, and in some cases warn against using the product for critical applications); ability to shift late lifecycle costs to customers; inadequate education, experience, and training; and unwillingness to pursue anything other than seemingly easy answers. Other reasons are also offered, as well.

Overly optimistic development plans that ignore these issues tend to win out over more realistic plans, but can lead to difficulties later on — for developers, system users, and even innocent bystanders. The annals of the Risks Forum (http://www.risks.org; see [267]) are replete with examples of systems that did not work properly and people who did not perform according to the assumptions embedded in the development and operational life-cycles. (One illustration of this is seen in the mad rush to paperless electronic voting systems with essentially no operational accountability and no real assurances of system integrity.) The lessons of past failures and unresolved problems are widely ignored. Instead, we have a *caveat emptor* culture, with developers and vendors disclaiming all warranties and liability, and users who are at risk. (In the case of electronic voting systems, the users include election officials and voters.)

We need better incentives to optimize over the long term (see Section 7.2) and over whole-system contexts (see Section 7.3), with realistic assumptions, appropriate architectural flexibility to adapt to changing requirements (Chapter 4), and sufficient attention paid to assurance (Section 6.9). Achieving this will require some substantive changes in our research and development agendas, our software and system development cultures, our educational programs, our laws, our economies, our commitments, and perhaps most important — in obtaining well documented success stories to show the way for others. Particularly in critical applications, if it's not worth doing right, perhaps it's not worth doing at all — or at least not worth doing without rethinking whatever might be problematic with the requirements, architecture, implementation, and/or operational practice. As an example, the essence of Extreme Programming (Section 2.3.6) seems interesting in achieving working partial systems throughout development, but would be applicable to critical systems only if it converges on products that truly satisfy the critical requirements. Once again, the emphasis must be on having well-defined requirements.

David Parnas has said, let's not just preach motherhood — let's teach people how to be good mothers. Indeed, the report you are reading seems to be preaching applicable motherhood. (Although the author of the report you are reading wrote in 1969 about the risks of overly narrow optimization and the importance of diligently applying generally accepted motherhood principles [255], the basic problems still remain today.)

One of the most ambitious efforts currently in progress is the U.S. Department of Defense Global Information Grid (**GIG**), which envisions a globally interconnected completely integrated large-scale fully interoperable end-to-end multilevel-secure networking of computer systems by 2020, and capable of providing certain measures of guaranteed services despite malicious adversaries, unintentional human errors, and malfunctions. The planning and development necessary to attain the desired requirements suggest the need for long-term vision, nonlocal optimization, and whole-system perspectives (see Sections 7.1, 7.2, and 7.3, respectively) — without which you realistically cannot get there from where we are today. The desirability of observing the principled and disciplined developments described in this report becomes almost self-evident, but still not easy to satisfy, especially with the desire to use extensive legacy software. However, the Enlightened Architecture concept noted at the end of Section 4.3 is fundamental to the success of any environment with the ambitious goals of the GIG.

7.2 The Importance of Up-Front Efforts

Perhaps the most important observation here is that if systems and applications are developed without an up-front *commitment to* and *investment in* the principles discussed here, very little that is discussed in this report is likely to be applied effectively. The commitment and investment must be both intellectual and tangible — in terms of people, funding, and perseverance. Looking at the recommended approaches as an investment is a vital notion, as opposed to merely relying on the expenditure of money as a would-be solution. Admittedly, the long-term arguments for up-front investment are not well understood and not well documented in successful developments — for example, with respect to the positive return on investment of such efforts compared with the adverse back-end costs of not doing it better in the first place: budget overruns, schedule delays, inadequacy of resulting system behavior, lack of interoperability, and lack of evolvability, to cite just a few deleterious results.

It would seem completely self-evident that the long history of system failures would suggest the need for some radical changes in the development culture. For example, this report strongly advocates realistically taking advantages of the potential benefits of up-front efforts (e.g., careful *a priori* establishment of requirements, architectures, and specifications). Certainly, this is not a new message. It was a fundamental part of the Multics development beginning in 1965 [84, 85], and it was fundamental to the PSOS design specifications from 1973 to 1980 [268, 269]. Nevertheless, it is a message that is still valid today, as for example in a new series of articles in the *IEEE Security & Privacy* [228] on building security into the development process, edited by Gary McGraw. Unfortunately, the fact that this is not a new message is in part a condemnation of our education and development processes, and in part a sign that our marketplace is not fulfilling certain fundamental needs.

A recent global survey of software development practices (Cusumano et al. [90]) strongly supports the wisdom and cost benefits of up-front development. Their survey includes some rather startling conclusions based on a sampling of software projects. For example, detailed design specifications were reportedly used in only 32% of the U.S. projects studied, as opposed to 100% of the projects in India. Furthermore, 100% of the Indian projects reported doing design reviews, and all but one of those projects did code reviews; this was characteristically untrue of the U.S. projects studied. Although it is unwise to draw sweeping generalizations from this survey, the issues considered and the results drawn therefrom are extremely relevant to our report. Besides, if the effectiveness of resulting foreign software developments is actually significantly better, then the rush to outsource software development might in some cases also be motivated by quality considerations, not just cost savings. This has very significant long-term implications — for the U.S. and for other nations with rapidly developing technology bases.

7.3 The Importance of Whole-System Perspectives

> *If you believe that cryptography is the answer to your problems, then you don't understand cryptography and you don't understand your problems.*
> Attributed by Butler Lampson to Roger Needham and by Roger Needham to Butler Lampson

Unfortunately, up-front effort is not enough by itself. Perhaps equally important is a system-oriented perspective that considers all of the pieces and their interactions in the large, with respect to the necessary requirements. Such a perspective should include (for example) the ability to have an overall conceptual understanding of all relevant requirements and how they relate to particular operational needs; an overall view of the entire development process and how it demands the ability to carry out cyclical iterations; and an overall view of any particular system-network architecture as representing a single virtual system in the large as well as being a composition of systems with predictable properties relating to their interconnections and interoperability. The challenge from the perspective of composability is then to understand the big picture as well as to understand the components and their interrelationships, and to be able to reason from the small to the large — and from the large to the small. Purely top-down developments are typically limited by inadequate anticipation of the underlying mechanisms, and purely bottom-up developments are typically limited by inadequate anticipation of the big picture.

There are many would-be short-term "solutions" that emerge in part from the lack of big-picture understanding, but that then take on lives of their own. For example, trusted guards, firewalls, virus checkers, spam filters, and cryptography all have benefits, but also have many problems (some intrinsic, some operational).

- Trusted guards (e.g., dirty-word filters or sensitive content checkers based on key words or security labels) represent a simple strategy for controlling undesired dissemination of documents. However, they tend to be overly simplistic, and often can be fooled into violating their own desired policies.

- Firewalls could be more effective if they were not configured to pass many types of executable content, such as ActiveX and JavaScript. However, many users want those features enabled, thus opening up some serious security holes.

- Viruses, worms, and other active content would be much less harmful in a well-architected environment that could confine executable content to a so-called sandbox in which it remains harmless (as can be accomplished in a domain-based or layered architecture or in a restricted virtual machine). Instead, vendors attempt to keep rule-based malware detectors up to date to recognize continual streams of new attacks and new attack types, which somehow seems to be the wrong battle to be fighting. Incidentally, to date, viruses and worms have been rather benign, considering the full potentials of really malicious code. Much more serious malware including insider-planted Trojan horses is lurking.

- Spam filters are also a potentially losing battle, as the spammers seem very adept at adapting to whatever defenses they encounter — new rules, Bayesian detectors, even certain simple challenge-response techniques. Legislation seems too simplistic to make real inroads against them, and may simply drive the spammers offshore. Major changes may be required — for example, to Internet-connected systems and to network architectures and protocols, facilitating traceback and accountability, as well as possibly to who pays for what services. (One possible alternative that may be of possible interest is Tripoli [376] — a user-Empowered E-Mail Environment, or Triple-E — which gives end users control over security, privacy, sender authentication, and spam defenses.)

- Cryptography is absolutely fundamental to the design and implementation of trustworthy distributed systems and networks. However, its effectiveness can be compromised by being poorly implemented or poorly embedded in a vulnerable environment (e.g., improperly encapsulated hardware or a flawed operating system), vulnerable to out-of band attacks (e.g., electromagnetic sensors, acoustic sensors [21, 349], differential power analysis [192, 193], light [13], and noise insertion [54]), or simply poorly used. (A DES crypto product considered over a decade ago comes to mind in which a pass-phrase was intended to generate a

56-bit key, but actually seemed to yield about 9 bits of key strength. We hope it is no longer on the market!)

The quote at the beginning of this section is symptomatic of the problem that the best cryptography in the world can still be compromised if not properly embedded and properly used. This entire section can be summed up by polymorphizing the quote at the beginning of this section, as symptomatic of the risks of overly simplistic solutions: for many different instantiations of X, *If you believe that X is the answer to your problems, then you don't understand X and you don't understand your problems.*

On the other hand, total systems awareness is a very rare phenomenon. It is not taught in most universities. Perhaps systems are considered to be lacking in theory, or uninteresting, or unwieldy, or dirty, or too difficult to teach, or perhaps just frustrating, or a combination of all of these and other excuses. As a result, system-oriented perspectives are slow to find their way into practice.

As a historical note, Edsger Dijkstra provides an example of a true pioneer who apparently lost interest in trying to deal with the big picture. In his earlier years, he was particularly concerned with the scalability of his elegant analytic methods to larger systems (for example, his work on structured programming [107], CSP [105], and the THE system [106] noted in previous chapters). Perhaps out of frustration that practitioners were not heeding his advice, he later became increasingly focused only on very elegant small examples (cf. [121]), trying to teach his beliefs from those examples in the hopes that others would try to extrapolate them to systems in the large. The essential thrust of this report is that systems in the large can be effectively developed and analyzed as compositions of smaller components, but only if you can see and comprehend the big picture.

One of the frequently heard arguments against spending more effort up front and optimizing over a longer term relates to situations in which there has never previously been an attack of such a magnitude that the need for extraordinary actions became totally obvious. This is the mentality that suggests that because we have not had a Pearl-Harbor or 9/11 equivalent in cybersecurity, there is no real urgency to take proactive action against hypothetical possibilities. This mentality is compounded by the use of statistical arguments that attempt to demonstrate that everything is just fine. Unfortunately, events that seemingly might occur with very low probabilities but with extremely serious consequences tend to be very difficult to comprehend. In such cases, quantitative risk assessments are particularly riskful, because of the uncertainty of the assumptions. For example, see Neumann's *Computer-Related Risks* book [260]. The entire book suggests a much greater need for realistic risk assessments and corresponding proactive actions. More specifically, pages 255–257 of the book provide a discussion of the risks of risk analysis (contributed by Robert N. Charette), and pages 257–259 consider the importance of considering risks in the large.

7.4 The Development Process

I would not give a fig for the simplicity this side of complexity, but I would give my life for the simplicity on the other side of complexity. Oliver Wendell Holmes

Returning once again to the Einstein quote at the beginning of Section 2.1, we note that the common tendency to oversimplify complex entities is perverse and usually counterproductive. The ability to clearly represent complexity in a simpler way is an art form, and usually very instructive — but difficult to achieve.

This section considers perceived and real difficulties with trying to use the concepts of the previous chapters, relating to requirements, architectures, and implementation. It suggests how the development process can be made much more effective, and how it can give the appearance of local simplicity while dealing with complexity more globally.

7.4.1 Disciplined Requirements

Well-understood and well-defined requirements are absolutely vital to any system development, and most particularly to those systems that must satisfy critical requirements such as security, reliability, safety, and survivability. They are also useful in evaluating the effects of would-be subsequent changes. Unfortunately, such requirements are seldom precisely defined *a priori*. Even more difficult are somewhat more subtle requirements, such as pervasive ease of use, interoperability, maintainability, and long-term evolvability — of the requirements as well as of the architectures and implementations. Jim Horning suggests that evolvability is to requirements as specification is to code, although at a higher level of abstraction. That is, if you don't delineate the space of possible future changes to requirements, you are likely to wind up with requirements that are as difficult to evolve as is code for which there are no specifications or specifications that do not anticipate change. However, well-understood and well-defined requirements are not common.

Even less common are explicit requirements for required software engineering sophistication, operational constraints, and specified assurance (such as the EAL levels of the Common Criteria). Requirements engineering should play a more prominent role in computer system development, which would necessarily entail adding discipline to both the process of defining requirements and to the statement of requirements themselves.

For example, the archives of the Risks Forum are littered with cases attributable to requirements problems that propagated throughout the development process into operational use. (See particularly the items denoted by the descriptor r in the Illustrative Risks compendium index [267]. Noteworthy examples include the Vincennes Aegis system shootdown of an Iranian Airbus, the Patriot missile clock-drift problem, and even the Yorktown Aegis missile cruiser dead in the water. See Section 6.9 for these and other cases.) Many lessons need to be learned from those cases. It is generally agreed that efforts to define and systematically enforce meaningful requirements early in system developments can have enormous practical payoffs; however, there seems to be enormous resistance to carrying that out in practice, because it increases up-front costs and requires greater understanding (as noted in Section 7.1).

7.4.2 Disciplined Architectures

The material in the foregoing chapters is basic to sound system architectures for trustworthy systems and their implementation. As a reminder of what we have thus far, Section 2.6 summarizes some of the primary caveats that must be observed in applying the principles of Chapter 2; these principles are not absolute, and must be used intelligently. Chapter 3 discusses constraints on subsystems and other components that can enhance composability, with Section 3.2 outlining obstacles that must be avoided. Chapter 4 considers further directions that can contribute to principled composable architectures. Chapter 5 stresses the importance of interface design. Chapter 6 discusses techniques for achieving higher assurance.

In this section we consider how to apply the approaches of the previous chapters into architectures that are inherently more likely to lead to trustworthy implementations. For example, realistic architectures should proactively avoid many problems such as the following:

- Risks of architectural oversimplification, such as systems whose requirements are seriously incomplete, systems that are implemented without a sensible architecture, operating systems that do not adequately react to hardware failure modes or that do not properly embed cryptography (whether implemented in hardware or in software), application software that does not properly encapsulate the operating system, systems and networking that ignore reliability and denial of service attacks, and so on.
- Risks of architectural overcomplexification, such as the immense bloatware common in various mass-market software products, the excessive interdependence and lack of encapsulation among different components.
- Risks of commonly observed flaws (for example, affecting security and reliability), malicious code, and so on.
- Popular but often mistaken beliefs, such as the belief that principles, hierarchical structures, modularity, encapsulation, and so on, are inherently inefficient or unworkable. (Constructive approaches are considered in Section 4.3.)
- Perceived and real problems with structural decomposition and composition, such as risks of modularization, overmodularization, modularity without encapsulation, modularity without separation of concerns, prematurely frozen modularity, and so on. As noted in Section 3.3, increased trustworthiness can result from stark subsetting, which can be achieved through composition rather than through efforts to decompose poorly designed systems or to use only restricted subsets of poorly conceived system interfaces. Well-defined component interfaces and clean encapsulation can contribute to increased trustworthiness.

Topics whose consideration might make critical system developments more realistic include the following.

- Emphasizing constructive architectures that are proactively designed to be better able to satisfy the given requirements, such as discussed in Chapter 4.
- Exploiting the potentials of architectural analysis tools, in early stages of development as well as in response to subsequent design changes later in the development cycle. However, risks of overendowing the soundness of such tools must be avoided.
- Pursuing fully worked and carefully documented successful examples of the practical effects of proactively sound architectures.

From a practical point of view, it may seem unrealistic to expect rigorous specifications — especially formal specifications — to be used in developments that are not considered to have critical requirements. However, even the informal English-language specification documents that were required in the Multics development (for example) had a very significant effect on the security, reliability, modular interoperability, and maintainability of the software — and indeed on the discipline of the implementation.

7.4.3 Disciplined Implementation

> *Technique is a means, not an end, but a means that is indispensable.* Maurice Allard, renowned French bassoonist in the Paris Opera from 1949–1983

The best architectures and the best system designs are of little value if they are not properly implemented. Furthermore, properly implemented systems are of little value if they are not properly administered. In each case, "proper" is a term that implies that the relevant requirements are satisfied. Thus, risks abound throughout development and operation. However, the notion of principled composable architectures espoused here can contribute significantly to proper implementation and administration. The notion of stark subsetting discussed in previous chapters can aid significantly in simplifying implementation, configuration, and administration.

Many security flaws that typically arise in design and/or implementation (such as those enumerated in Section 2.4) lend themselves to exploitation. Indeed, each of the enumerated problem areas tends to represent opportunities for design flaws and for implementation bugs (in hardware just as well as software). Buffer overflows represent just one very common example. For some additional background on buffer overflows and how to prevent them, see the discussion in the Risks Forum, volume 21, numbers 83 through 86, culminating in Earl Boebert's provocative contributions in volume 21, numbers 87 and 89. Boebert refers to Richard Kain's 1988 book on software and hardware architecture [185], which provides considerable discussion of unconventional system architectures for security — including the need for unconventional hardware platforms. Furthermore, the Multics operating system architecture constructively avoided most stack buffer overflows. The combination of hardware, the PL/I language subset, the language runtime environment, the stack discipline (nonexecutable stack frames; also, the stack grew to higher addresses, making the overflow of a buffer unlikely to clobber the return address in the stack frame), and good software engineering discipline helped prevent most buffer overflows in Multics. (See Tom Van Vleck's comments in the Risks Forum, volume 23, issues 20 and a follow-up in issue 22.) For other background, see also Bass [31] for architecture generally, Gong [145, 146] for the Java JDK architecture intended to provide secure virtual machines, and Neumann [264] for survivable architectures.

Many implementation issues create serious problems. Establishing sensible policies and sound configurations is an enormously complicated task, and the consequences to security, reliability, functionality, and trustworthiness generally are very difficult to predict. We need better abstractions to control and monitor these policies and configurations, and to understand them better.

Various popular myths need to be considered and debunked — for example, the fantasy that a perfect programming language would be able to prevent security bugs. Another myth is that precompile and postcompile tools can detect and remove many classes of bugs. In general, for nontrivial programming languages, both of these myths can be true in principle only for certain types of bugs, although even the best programmers still seem to be able to write buggy code.

7.5 Disciplined Operational Practice

> *System programming is like mountain climbing: It's not a good idea to react to surprises by jumping — that might not improve the situation.* Jim Morris

Principled composable architectures can contribute not only to trustworthy implementation (as noted at the beginning of Section 7.4.3), but also to sound operational practice — particularly if considerable attention is paid to system interface design that addresses the needs of system administrators. However, for existing (e.g., legacy) systems that have resulted from inadequate attention to human operational interfaces, other approaches must be taken — even if only better education and training.

Operational issues represent enormous potential problems, such as considerable operational costs, shortages of readily available in-house staff, risks of excessive complexity, poorly defined human interfaces, and typically systems that require an ever-present demand for system administrators — especially in crisis situations. This last concern may be escalated by increasing pressures to oursource operations and administration personnel.

One concept that in principle would greatly improve operational practice and operational assurance would be the notion of automatic recovery, mentioned in Section 4.2. The ability to recover from most deleterious state-altering events (whether malicious or accidental) without human intervention would be an enormous benefit. Autorecovery requirements have serious implications for system architectures, and would be greatly simplified by the principle of minimizing the need for trustworthiness. Assurance associated with that recovery (e.g., based on the soundness of the architecture itself and on real-time revalidation of the soundness of the system state) would also be valuable. However, making autonomic systems realistic will require further research and extremely disciplined development.

7.5.1 Today's Overreliance on Patch Management

> Dilbert: *We still have too many software faults. We'll miss our ship date.*
> Pointy-Haired Manager: *Move the list of faults to the "future development" column and ship it.*
> PHM, aside: *90% of this job is figuring out what to call stuff.*
> Scott Adams, three-panel Dilbert comic strip, 4 May 2004

Mass-market software as delivered in the past and in the present tends to have many flaws that can compromise the trustworthiness of systems, networks, and applications. As a result, system purveyors and system administrators are heavily dependent on patch management — that is, developers must continually identify vulnerabilities, create would-be fixes, test them, make those fixes available, and hope that further flaws will not be created thereby. Operational installations must install the patches in the correct order in a timely fashion, at the risk of breaking or otherwise affecting existing applications.

Patch management is an example of a slippery-slope rathole. Systems should be designed much more carefully and implemented with much greater care and attention to good software engineering practice, easily usable operational and system administration interfaces, and composable upgrade procedures that are integral to the architecture, applications, and user software. Better design and implementation must also be coupled with comprehensive testing, evaluations, and other analyses such as advanced tools to detect serious vulnerabilities; developers should do this before release, rather than simply foisting buggy software on unsuspecting customers who become the Beta testers. However, in the commercial rush to marketplace, essentially none of this happens. Thus, pouring palliative efforts into improving patch management completely misses the much more fundamental point that patches should ideally be minimized through better design and implementation, so that they become rare exceptions rather than frequent necessities. Putting the burden on patch management is somewhat akin to believing in better management of fixed reusable passwords — that merely increasing password length, including nonalphabetic characters, and changing passwords often will improve authentication; such simplistic approaches totally ignore the risks of fixed passwords that transit networks unencrypted or are otherwise exposed and the risks of exploitable vulnerabilities in systems that allow the password system to be completely bypassed. A better solution for authentication is of course *not to rely on* conventional fixed passwords as the primary means of authentication, and instead to move to trustworthy systems and trustworthy networking, cryptographically protected tokens or smartcards within the context of trustworthy systems, combined with layered protection, separation of privileges, and judicious observance of the applicable principles noted in Chapter 2, plus a much greater commitment to better system security and reliability throughout development and operation.

- One problem with many patches is that they require shutting down some services (or, indeed, rebooting the entire machine) for a noticeable period of time. This is not always acceptable, as in a hospital's telephone system and operational support. There are ways of engineering around this problem, but they tend to require enlightened system architectures, and hence are rarely developed — and even more rarely deployed.
- Patches are often faulty, and tend to break other functionality for some users. Research exists on how long you should wait before installing a patch, to enable other people test it for you before you have to, but the assumptions are often suspect. In the absence of an imminent threat, it may be wiser to wait — assuming that the experiences of others are available to you.
- No matter how wonderful an automated patch management system might seem, 100% deployment is very unlikely. Anticipating all sorts of patch problems rapidly leads to a combinatorial explosion of various multiple patches installed in the field, and makes testing the next patch and its interactions with previous patches nearly impossible. Besides, automated remote patching is extremely riskful.

Although it may be necessary evil, dependence on patch management as a major component of security defenses seems too much like micromanaging the rearranging of deckchairs on the Titanic. The barn door is already wide open, and the barn is empty of more fundamental ideas. See [375] for another view of patch management.

7.5.2 Architecturally Motivated System Administration

Clearly alternative approaches are needed that simplify system administration and minimize the downsides of patch management. Perhaps we need not just better software engineering in general, but also a methodology that encompasses "design for patching" when "design for avoiding patches" fails — just as hardware vendors have moved to "design for test" and "design for verification" methodologies. Design for patching should encompass system architecture (e.g., modularity and encapsulation) as well as operational characteristics (e.g., bilateral trusted paths for upgrades). Inherently sound architectures can minimize the need for patching — as for example in carefully designed autonomic systems and fault-tolerant systems that anticipate the need for rollback, hot standbys, or other alternative measures in response to detected anomalies. Greater attention to the human interfaces (see Chapter 5 and the next section) is also essential.

According to some reports, patch management is on the order of a $5-billion dollar problem per year. It is probably responsible for much more than that if hidden costs are included, such as down-time and lost work resulting from failed patches. Jim Horning notes that all automobile drivers once had to know how to patch an inner tube (or at least how to change a tire to drive someplace and get one patched). Today inner tubes are gone, and we go years between flat tires. That seems preferable to a highly efficient patching system.

7.6 Practical Priorities for Perspicuity

Returning to the notion of perspicuous interfaces considered in Chapter 5, this section considers some of the practical issues relating to interface design. Given the range of material addressed in this report, one important question that remains to be addressed is this: Where are the biggest potential payoffs, and what priorities should be allocated to possible future efforts, with respect to dramatically increasing the understandability of systems and their interfaces — especially under crisis conditions. The same question also applies to subsystem interfaces that may be invisible to end users but vital to application developers, integrators, and debuggers. It is important to note that good interface design is essential not only to human users, but also internally to systems themselves — especially autonomic systems.

One of the most important challenges relates to the roles that administrators play in configuring and maintaining operating systems, application software, networks, control systems, and so on. Even with the hoped-for advent of autonomic systems and networks, significant burdens will rest on admins when something fails or is under attack. Thus, perspicuity for admins must be a high-priority concern. This concern must be twofold: (1) System interfaces must be better designed with admins in mind. (2) Analysis tools must greatly facilitate the critical roles of admins. The potential payoffs for better perspicuity for admins are enormous, in terms of reducing operational costs, increasing speed of remediation, minimizing dependence on critical human resources, increasing job satisfaction, and — above all — improving system security and survivability.

A second challenge has to do with dealing with legacy systems that were not designed with adequate security, reliability, robustness, and interface perspicuity, and that therefore cannot be easily retrofitted with such facilities. This is an unfortunate consequence of many factors, including the inability of the marketplace to drive needed progress, generally suboptimal software development practices, and constraints inherent in closed-source proprietary software — such as a desire on the part of system developers to keep internal interfaces hidden and making it more difficult for competitors to build compatible applications. In this situation, much more perceptive analysis methods and tools are needed, although those tools would be applicable to closed-source as well as source-available software. To the extent that analysis tools can be applied to available source code (whether proprietary or not) rather than object code, the more effective they are likely to be.

A third challenge is that, whichever approaches are taken, they must include criteria and techniques for measuring and evaluating their effectiveness. This again suggests the need for better analysis methods, but in the long run also necessitates system developments that anticipate the needs of improved measurability of success.

Thus, our suggestions for realistic priorities are as follows, in several dimensions:

Prioritized Approaches for Achieving Greater Perspicuity

1. A combination of constructive interface design and analysis tools for newly developed software, recognizing a leverage advantage for available source code (the most effective alternative)

2. Analysis tools that rely on system source code to enhance interface perspicuity for legacy systems, but not on any substantially new or modified interfaces (an intermediate alternative)

3. Analysis tools that are restricted to see only object code to enhance interface perspicuity (a less desirable and often less effective alternative, although possibly useful when source code is not available)

Prioritized Human Targets for Enhanced Perspicuity

1. System developers, debuggers, and integrators (with highest payoff)

2. System administrators (with very high payoff)

3. Conventional application developers (with very high payoff) and users (with considerable payoff)

Potential System Targets for Enhancing Perspicuity

1. Linux or one of the BSD operating systems (attractive because of availability of source code)

2. TCP/IP related behavior (complex, but potentially very useful)

3. A realistic multilevel security system (less accessible, but with considerable potential use)

7.7 Assurance Throughout Development

> *The whole is greater than the sum of its parts.* This can be true particularly in the presence of effort devoted to sensible architectures, interface design, principled development, pervasive attention to assurance, and generally wise adherence to the contents of this report. In this case, emergent properties tend to be positive, providing evidence of trustworthiness.
>
> *The whole is significantly less than the sum of its parts.* This can be true whenever there is inadequate attention devoted to architecture, interface design, principled development, assurance, or foresight — for example, resulting in serious integration difficulties and the lack of interoperability, delays, cost overruns, design flaws, implementation bugs, overly

complex operations, deadly embraces, race conditions, hazards, inadequate security and reliability, and so on. In this case, emergent properties tend to be negative, providing evidence of untrustworthiness.

This section reassesses the approaches of Chapter 6 with respect to the practical thrust of the present chapter. In particular, Section 7.7.1 considers assurance related to the establishment and analysis of requirements. Section 7.7.2 reconsiders assurance related to system development, for example, potentially fruitful techniques for assuring the consistency of software and hardware with their respective specifications (Section 6.5). Section 7.8 then considers the practicality of assurance techniques applied to operational practice.

7.7.1 Disciplined Analysis of Requirements

It is a common misconception that establishing requirements carefully is generally not worth the effort. Nevertheless, further evidence would be useful in dispelling that myth, especially concerning formal requirements and formal analyses thereof, and particularly in cases of critical systems and outsourcing/offshoring of software development (see Section 7.10.2).

From a practical point of view, it is immediately obvious that the disciplined use of formal or semiformal analysis methods and supporting tools would have a significant up-front effect that would greatly reduce the subsequent costs of software development, debugging, integration, and continual upgrades. There is a slowly growing literature of such approaches, although there are still relatively few demonstrated successes. One example is provided by the use of formal methods for NASA Space Shuttle requirements [109] — where the mission is critical and the implications of failure are considerable.

7.7.2 Disciplined Analysis of Design and Implementation

Existing analysis techniques and supporting tools for system architectures and for software and hardware implementations tend to be fairly narrowly focused on specific attributes, certain types of design flaws, and specific classes of source-code and object-code bugs (the U.C. Berkeley MOPS analyzer, `purify,` trace tools, debuggers), security vulnerabilities (e.g., attack graph analysis using symbolic model checking [180, 352]), and hardware mask layout properties. Most of these approaches are limited to static analysis — although they may sometimes be helpful in understanding dynamic problems.

One of the most important problems raised in this report is the ability to determine analytically the extent to which systems, modules, and other components can be composed — that is, identifying all possible deleterious interactions. As discussed in Section 6.2, providing a set of analytic tools to support the practical analysis of the composability of requirements, specifications, protocols, and subsystems would be extremely valuable. For example, analysis should consider the interference effects of improper compositions, or else demonstrate the invariance of basic properties and the appropriateness of emergent properties under composition.

Static checking tools along the lines of `lint, splint, ESC, Aspect, Alloy` [173] (and, in general, what are referred to as "80/20 verifiers") can be extremely helpful. However, their infallibility and completeness should never be overendowed. Although all low-hanging fruit should certainly be harvested, what is harder to reach may have even more devastating effects.

A set of tools for the analysis of safety specifications [311] has been sponsored by NASA Langley, and is also worth considering — not only for safety, but for its potential application to other elements of trustworthiness.

7.8 Assurance in Operational Practice

Operational practice — for example, system administration, routine maintenance, and long-term system evolution — represents an area in which assurance techniques have not been used much in the past. There are various approaches that might be taken, some fairly ad hoc, and some formally based.

- **Configuration management and configuration analysis tools.** Configuration management and network management could both benefit greatly by the introduction of some assurance-enhancing methodologies and supporting tools — for example, assessing the consistency and the compliance with requirements for each configuration change, issuing warnings for potentially riskful changes, requiring confirmation for those changes, and tracking the historical record of all configuration-related actions. COPS and SATAN are examples of existing static analysis tools for detecting system and network configuration errors or suspicious irregularities.

- **Anomaly/misuse detection and automated responses.** Some of the research-oriented anomaly and misuse analysis systems are fairly effective at detecting potential misuse, although the false-positive and false-negative measures vary widely according to the desired application. However, there has been relatively little effort on understanding the meaning of the detected events and the actual intent of attackers. (For an example of a recent project on determining attacker intent, see [305] — which provides an additional analysis capability within SRI's EMERALD.) Furthermore, there has been relatively little serious work on intelligent near-real-time responses, particularly under emergency conditions and after detected anomalies. The risks of underreacting and overreacting are both important concerns. Although the potential benefits of intelligent response systems would be considerable, the effectiveness of such systems will depend heavily on the range of approaches discussed here for constructive system development and suitable analysis aimed at perspicuous interfaces. As a caution, iatrogenic reactions in response to would-be cures (wherein the remedy is worse than the disease) are often symptomatic of such problems.

- **Automated recovery from outages.** We note in Section 7.4.3 the desirability of the ability to recover from a wide variety of deleterious state-changing events, without human intervention except in the most difficult cases. There are numerous opportunities for assurance techniques associated with revalidation of the forward- or backward-recovery state. Any such mechanisms should also be coupled with the configuration management assurance. Effective hierarchical recovery strategies may be particularly important, as recommended to (D)ARPA [270] in 1973, in the context of fault-tolerant systems.

- **Incremental-closure analysis of would-be actions.** Every operational change has the potential to invalidate earlier analyses based on the previous software and the previous system configurations; ideally each change should result in reevaluation of the new configurations. Determination of the scope of what has changed and the minimum extent to which reevaluation must be done would be extraordinarily helpful in narrowing down the resulting analysis. (This approach also can apply to all changes throughout the development cycle, where a change to requirements typically results in the invalidation of certain specifications, and where a change to specifications typically invalidates certain software.)

- **Secrecy and privacy violations.** Privacy problems also need to be considered, precisely because they are so often ignored in analysis environments in which access to potentially sensitive state information is required. For example, the dynamic implications of who is accessing which data and which state information, and under what circumstances, has important privacy implications that could benefit from carefully controlled real-time analysis. Analyses of broken systems can often reveal sensitive information.

In addition, there are also many system architectural concepts that can contribute to assurance aspects of operations.

Significant effort is needed to harness existing analysis tools and to pursue new analysis techniques and tools, to accommodate dynamic understandability of systems in execution. For example, such effort would be valuable in responding to anomalous real-time system behavior and to evaluate the would-be effects of possible system changes, particularly regarding flawed systems and complications in operation and administration.

Configuring security policies into applicable mechanisms is a particularly important problem. To this end, Deborah Shands et al. are developing the SPiCE translation system[350] at McAfee Research. SPiCE automatically translates high-level access policies to configurations of low-level enforcers,

7.9 Certification

Cer.ti.tude: the state of being or feeling certain;
Rec.ti.tude: correctness of judgment or procedure
(Abstracted from Webster's International Dictionary)

Certification is generally considered as the process of applying a kind of blessing to a system or application, implying some kind of seal of approval. The meaning of that certification varies wildly from one environment to another, as noted in the following two paragraphs (which are adapted from [262], along with the definitions noted above).

There is a fundamental difference between certification (which is intended to give you the *feeling* that someone or something is doing the right thing) and trustworthiness (for which you would need to have some *well-founded reasons for trusting* that someone or something is doing the right thing — always interpreted with respect to appropriate definitions of what is right). Certification is typically nowhere near enough; an estimate of trustworthiness is somewhat closer to what is needed, although ideal trustworthiness is generally unattainable in the large — that is, with respect to the entire system in operation. Formal demonstrations that something is consistent with expectations are potentially much more valuable than loosely based certification. (Recall the discussion of consistency versus correctness in Section 6.2.) So, a challenge confronting us here is to endow the process and the meaning of certification — of systems and possibly of people (see below) — with a greater sense of rigor and credibility.

Numerous system failures (e.g., [260]) demonstrate the vital importance of people. Many cases are clearly attributable to human shortsightedness, incompetence, ignorance, carelessness, or other foibles. Ironically, accidents resulting from badly designed human interfaces are often blamed on operators (e.g., pilots, system administrators, and users) rather than developers. Unfortunately, *software engineering* as practiced in much of the world is merely a buzzword rather than an engineering profession [288, 289]. This is particularly painful with respect to systems with life-critical, mission-critical, or otherwise stringent requirements. Consequently, some of the alternatives discussed in this report deserve extensive exploration, such as these:

- Principled development practice, enforceable requirements, and sensible system architectures are all valuable. Their principled use should earn insurance discounts, contractual bonuses for on-time and under-budget delivery as well as satisfactory attainment of requirements, and perhaps even legal relief from certain aspects of liability (although this is not generally recommended). Bad development practice (including low bidders taking unwise shortcuts and risks) must not be condoned, and should somehow be strongly disincentivized.
- Critical systems should be developed by persons and companies with commensurate education, training, and experience.

Software certification is a slippery slope that can raise false hopes. However, its usefulness can be greatly enhanced somewhat if all of the following are present: (a) well-defined detailed requirements; (b) architectures that anticipate the full set of requirements and that can be predictably composed out of well-conceived subsystems; (c) highly principled development techniques, including good software engineering disciplines, serious observance of important principles such as layered abstraction with encapsulation, least privilege, defensive analytic tools, and so on; (d) judiciously applied assurance measures, pervasively invoked throughout development and evolution, including formal methods where applicable and effective; (e) meaningful evaluations such as consistency between specifications and requirements, consistency between software and specifications, and dynamic operational sanity checks. In this way, certification might have some real value. However, in practice, certification is far short of implying trustworthiness.

One horrible example of the inadequacy of certification in practice is provided by the currently marketed fully electronic voting machines without a voter-verified audit trail (for example, a paper record of the ballot choices, which remains within the system and is not kept by the voter); all of today's all-electronic paperless voting machines lack any meaningful trustworthiness with respect to system integrity, accountability, auditability, or assurance that your vote goes in correctly. These proprietary closed-source systems are certified against very weak voluntary criteria by a closed process that is funded by the developers. In addition, recent disclosures demonstrate that software used in the 2002 and 2003 elections was not the software that was certified; in many cases, potentially significant changes were introduced *subsequent to certification.*

However, simplistic strategies for institutional certification (such as the Capability Maturity Model) and personnel certification (such as the Certified Information Systems Security Professional — CISSP — examination and personal designation) are also slippery slopes. Reviews by Rob Slade of numerous books on the limitations of the CISSP exam can be found in the Risks Forum athttp://www.risks.org; for example, see volume 21, issues 79 and 90, and volume 22, issues

08, 10, 36, 49, and 57 (the last of these covering four different books!). (Note: The Risks Forum moderator stopped running Slade's reviews on this subject after all of the above-mentioned books seemed to have similar flaws reflecting difficulties inherent in the CISSP process itself; there are many other books on CISSP than these.)

Although there is some merit in raising the bar, unmitigated belief in these simplistic approaches is likely to induce a false sense of security — particularly in the absence of principled development and operation. In the case of the CMM, the highest-rated institutions can still develop very bad systems. In the case of the CISSP, the most experienced programmers can write bad code, and sometimes the least experienced programmers can write good code.

7.10 Management Practice

7.10.1 Leadership Issues

Some of the biggest practical problems relate to the role of Corporate Information Officers (CIOs) in corporate institutions, and their equivalents in government institutions. (Note: There is still no Federal CIO for the U.S. Government, which is increasingly causing certain problems.)

CIOs are generally business driven, wherein cost is often considered to represent the primary, secondary, and tertiary motivating forces. The advice of Corporate Technical Officers (CTOs) is often considered as close to irrelevant. The business issues generally motivate everything, and may override sound technological arguments. This has some unfortunate effects on the development and procurement of trustworthy systems and networks, which tend to be reinforced by short-sighted optimation and bottom-up implementations.

7.10.2 Pros and Cons of Outsourcing

Outsourcing is a real double-edged sword, with many benefits and risks, and with many problems that result from trying to optimize costs and productivity — both in the short term and in the long term (e.g., as suggested by the last paragraph of Section 7.2). It is seemingly particularly cost-advantageous where cheaper labor can be effectively employed without adverse consequences — for example, for software development, hardware fabrication, operations and administration, maintenance, documentation, business process work, and other labor-intensive services (such as call centers). However, there are many hidden costs; indeed, several recent studies suggest that the case for overall cost savings is much less clear-cut. Furthermore, other considerations may also be important, such as the ability to innovate compatibly, integrated workforce development, planning, coordination, intellectual property, security, and privacy. These tend to be less tangible and less easily represented in cost metrics.

From the perspective of a would-be controlling enterprise, we consider two orthogonal dimensions that relate to the extent of *outsourcing* and *offshoring*. Outsourcing typically involves contracts, subcontracts, or other forms of agreements for work performed by entities outside of the immediate controlling enterprise. Offshoring involves some degree of work performed by nondomestic organizational entities such as foreign subsidiaries, foreign companies, or foreign individuals. Thus, we can have widely varying degrees of both outsourcing and offshoring, with a wide range of hybrid strategies. The situation is simplified here by considering four basic cases:

- **DI:** Domestic in-house control (wholly internal)
- **DO:** Domestic outsourcing (on-shore outsourcing)
- **FI:** Foreign subsidiaries (e.g., wholly owned; in-house offshoring)
- **FO:** Foreign outsourcing (offshore outsourcing)

Table 7.1 outlines some of the issues that arise in each of these four cases. The left half of the table represents **I**n-house top-level control (I), and the right half represents some degree of **O**utsourcing (O). The upper half of the table represents wholly **D**omestic efforts (D), and the lower half involves some degree of **F**oreign offshoring (F).

The pros and cons summarized in the table are intended to be suggestive of concerns that should be raised before engaging in outsourcing and/or offshoring, rather than being dichotomous black-and-white alternatives. Indeed, the pros and cons for all quadrants other than the upper left tend to vary depending on the degree of outsourcing and/or offshoring, as well as such factors as relative physical locations, ease of communications, language barriers, standard-of-living differentials, job marketplaces, government regulation, and so on. Even the upper-left quadrant has variations that depend on management strength, centralization versus distributed control, employee abilities, and so on.

Several conclusions are suggested by the table.

- When considering outsourcing and/or offshoring of any normally in-house, hands-on, or up-front activities (and especially those that require close attention and possible substantive changes during development or subsequent use and evolution), beware of situations in which requirements, business models, and management plans are incompletely defined and inadequately specified.
- Beware of security and privacy problems resulting from offshoring, including Trojan horses, system integrity, accountability, and so on. Many of these problems typically remain hidden until much later.
- Beware of situations in which contractor or subcontractor discipline would be essential but difficult to ensure.
- Beware of weak links in chains of command. These are particularly riskful in outsourced and/or offshored efforts.
- Disciplined development and all of the other concepts discussed in this report remain vital irrespective of which quadrant of the table applies. Each quadrant has its own particular pitfalls.
- Serious attention needs to be devoted to better management of risks associated with outsourcing and offshoring. For example, see [369], which also examines a broad range of problems associated with defense acquisitions (which are themselves a serious source of risks).

Table 7.1: Pros and Cons of Outsourcing

DI: Domestic In-House Control	**DO: Domestic Outsourcing**
Pros: Closer access to business knowledge Tighter reins on intellectual property Tighter control of employees and development efforts	**Pros:** Resource balancing Potential cost savings, particularly for labor Offloading less desirable jobs
Cons: U.S. education often inadequate for system engineering, security, reliability, and trustworthiness Bad Government records in managing developments (Large corporations are sometimes not much better!)	**Cons:** Loss of business sense Increased burden on contracting Potential loss of control Bad records in managing contracted procurements Possible hidden offshore subcontracts (as in the ATC Y2K remediation) Greater security/privacy concerns
FI: Foreign Subsidiaries	**FO: Foreign (Offshore) Outsourcing**
Pros: Potential cost savings (esp. labor) In-house control largely retained Resource/labor balancing Choices exist for well-educated and disciplined labor. Up-front emphasis on requirements/ specs can increase product quality.	**Pros:** Potential cost savings (esp. labor), at least in the short term Resource/labor balancing Potential pockets of good education and disciplined labor in some cases Up-front emphasis on requirements/ specs can increase product quality.
Cons: Some loss of direct control More difficult to change requirements/specs/code/operations More risks of Trojan horses Possible language problems Hidden long-term costs Domestic job losses Loss of GNP and tax revenues Foreign laws generally apply, in addition to domestic laws. Potential political risks Privacy problems and other risks Risks of hidden subcontracts Some intellectual property concerns	**Cons:** Considerable loss of direct control Even more difficult to change requirements/specs/code/operations Greater risks of Trojan horses Possibly severer language problems Possibly more hidden long-term costs Domestic job losses Loss of GNP and tax revenues Foreign laws may cause conflicts with domestic laws. Greater potential political risks More privacy problems and other risks Further loss of control of subcontracts Intellectual property control degradation Hidden indirected (nth-party) outsourcing

7.11 A Forward-Looking Retrospective

Pandora's cat is out of the barn, and the genie won't go back in the closet.
Peter G. Neumann

In the same way in which the quote at the beginning of Section 7.3 can be parameterized to apply to many narrow would-be "solutions" for complex problems, the above polymorphic Pandoran multiply-mixed metaphor can be variously applied to cryptography, export controls, viruses, spam, terrorism, outsourcing, and many other issues.

Over the past forty years, many important research and development results have been specifically aimed at achieving trustworthy systems and networks. However, from the perspective of applications and enterprises in need of high trustworthiness, those results have mostly not been finding their way into commercial developments. Reasons given variously include increased development costs, longer delays in development, extreme complexity of adding significant levels of assurance, lack of customer interest, and so on. Perhaps even more important are factors such as the inadequacy of educational curricula and training programs that minimize or ignore altogether such issues as highly principled system engineering and system development, software engineering, system architecture, security, reliability, safety, survivability, formal methods, and so on. A lack of knowledge and experience among educators fosters a similar lack in their students, and is particularly riskful when also found among managers, contracting agents, legislators, and system developers. Perhaps the most important challenge raised by this report is finding ways of bringing the concepts discussed here realistically and practically into mainstream developments.

A strong sense of history is not inconsequential, particularly in understanding how badly computer software development has slid down a slippery slope away from perspicuity. Much of the work done in the 1960s to 1980s still has great relevance today, although that work is largely ignored by commercial developments and by quite a few contemporary researchers.

- **Christopher Strachey's Axiom on Prognostication:** "It is impossible to foresee the consequences of being clever." As a consequence, it is very difficult to identify the great technologies of the future and pursue them aggressively ahead of their time.

- **Myth.** There is a popular myth to the effect that if a particular technology fails to make it in the real world for more than a decade or two, then it is probably not worth remembering: "Let others advocate it."

- **History.** Success in picking great technologies of the future requires avoiding the research and development "square wheels" of the past. Ignorance of the past invites repeating the same mistakes over and over. Awareness of past innovations and lessons learned therefrom can be very productive. Reflection on much of the work from those decades can provide considerable insight as to pervasive obstacles experienced today and how they might be avoided.

Hence, we conclude that awareness of much of the work done in the 1960s to 1980s related to trustworthiness is potentially useful today.

Voltaire's famous quotation (see *Dictionaire Philosophique: Art Dramatique*), "Le mieux est l'ennemi du bien." is customarily translated as "The best is the enemy of the good." (However, the French language uses *mieux* for both of the corresponding English words, *best* and *better*; thus, in a choice between just two alternatives, a correct English translation might be "The better is the enemy of the good.") This quotation is often popularly cited as a justification for avoiding attempts to create trustworthy systems. However, that reasoning seems to represent another nasty slippery slope. Whenever what is accepted as *merely good* is in reality *not good enough,* the situation may be untenable. Realistically speaking, the best we can do is seldom ever close to the theoretical best. Perfect security and perfect reliability are inherently unattainable in the real world, although they can occasionally be postulated in the abstract under very tightly constrained environments in which all possible threats can be completely enumerated and prevented (which is almost always unrealistic), or else simply assumed out of existence (as in supposedly perfect cryptographic solutions that are implemented on top of an unsecure operating system, through which the integrity of those solutions can be completely compromised from below). Thus, we come full circle back to the definition of trustworthiness in the abstract at the beginning of this report. In critical applications, the generally accepted "good" may well be nowhere good enough, and "better" is certainly not the enemy. In this case of short-sighted thinking, we quote Walt Kelley's Pogo: "We have met the enemy, and he is us."

The need for Information Assurance in the Global Information Grid (GIG) (noted at the end of Section 7.1) — for example, see [52] — provides a fascinating example of an environment with a very large collection of critical needs, and extremely difficult challenges for the long-term development of an enormous extensively interoperable trustworthy network environment that far transcends today's Internet. Considerable effort remains to flesh out the GIG requirements and architectural concepts. The principled and disciplined approach of this report would seem to be highly relevant to the GIG effort.

Chapter 8

Recommendations for the Future

The future isn't what it used to be. Arthur Clarke

8.1 Introduction

In this chapter, we consider some potentially important areas for future research, development, and system operation, with direct relevance to CHATS-like efforts, to DoD more broadly, and to various information-system communities at large. The recommendations concern the critical needs for high-assurance trustworthy systems, networks, and distributed application environments that can be substantially more secure, more reliable, and more survivable than those that are commercially available today or that are likely to become available in the foreseeable future (given the present trajectories).

One of the biggest challenges results from the reality that the best R&D efforts have been very slow to find their way into commercial practice and into production systems. Unfortunately, corporate demands for short-term profits seem to have stifled progress in trustworthiness, in favor of rush-to-market featuritis. Furthermore, government incentives for commercial development have been of limited help, although research funding has been a very important contributor to the potential state of the art. We need to find ways to improve that unfortunate history.

8.2 General R&D Recommendations

The whole of science is nothing more than the refinement of everyday thinking. Albert Einstein, *Ideas and Opinions,* page 290

This section provides a collection of broad recommendations for future R&D applicable to the development, operation, maintenance, and evolution of trustworthy systems and networks, relating to composability, assurance, system architectures, software engineering practice, and metrics. It also addresses the use of formal methods applicable to system and network architectures intended to satisfy critical security requirements. These recommendations take an overall system approach, and typically have both short-term and long-term manifestations. Each recommendation would benefit considerably from observance of the previous chapters.

- **Principle-based system development and assurance.** Many of the principles outlined in Chapter 2 can contribute significantly to better systems and networks, with greatly enhanced assurance. Although the Saltzer–Schroeder and related principles (including GASSP, Section 2.3.4) are not primarily oriented toward assurance, their judicious use can contribute significantly to fewer flaws and more trustworthy systems. With respect to future government-funded R&D efforts and to commercial developments, it is vital that those efforts reflect a more conscientious and pervasive awareness of these principles. Principled software engineering needs to become more pervasively a part of research and development. (This also suggests an urgent need for incorporating principled system development into mainstream educational curricula, as discussed in the last bulleted item of Section 8.5.)

- **Pervasively Integrated Assurance (PIA).** As noted in Section 6.2, it is clearly desirable in the pursuit of trustworthy systems and networks that a wide range of formal and informal assurance-related techniques become an integral part of the development cycle, and also are persistently invoked during operation, maintenance, administration, and long-term evolution — wherever beneficially applicable.

- **Requirements.** Serious effort is needed to establish canonical sets of composable requirements for useful attributes. Requirement composability should be nonconflicting wherevever possible, and explicitly identified where conflicting — for example, mutually exclusive or otherwise interfering requirements. The requirements for systems, subsystems, and networks should be subsettable and parameterizable, allowing them to be used for different types of applications, encompassing what is essential for security, reliability, survivability, real-time performance, static and dynamic configuration control, and so on.

- **Properties, interrelationships, composability, and deleterious effects thereof.** More emphasis should be devoted rather pervasively to carefully defining and assuring important properties (including, but by no means overly focused on, MLS where relevant) of subsystems, systems, and networks (including distributed and networked systems, systems of systems, and so on). Interrelations between security and other types of properties (such as fault tolerance, real-time performance, and emergency overrides) are generally more difficult to handle, but should also be represented. A system cannot be adequately trustworthy if its reliability is in doubt; it also cannot be adequately trustworthy if its security properties can be compromised by malfunctions or misuse by insiders and outsiders, or indeed *must* be operationally compromisible under certain real-time emergencies such as lost passwords or crypto keys, or emergency recovery without any assurance. Furthermore, where formal methods are used, they should be composable and 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 formal understanding of composition of subsystems is required, including compositions of horizontal abstractions as well as vertical abstractions — and especially the vertical abstractions provided by the trustworthiness enhancement mechanisms of Section 3.5. Increased understanding is also necessary with respect to the ways in which subsystem properties can be preserved or transformed under composition, involving both horizontal abstraction and vertical abstraction.

- **Assumptions.** More emphasis should be placed on making explicit all of the otherwise unstated assumptions, functional dependencies, and property dependencies underlying any system — and especially any of those to be formally specified and analyzed. Assumptions of proper human behavior are typically much more important than is usually recognized, 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 made explicit, and factored into certain analytic approaches so that the external risks can also become explicitly manifest. Furthermore, 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 [102, 236] and error-correcting coding capable of detecting and perhaps correcting human errors could be useful in this regard, as noted in Section 3.5.

- **Criteria.** The TCSEC/ITSEC/CTCPEC/Common Criteria efforts represent useful iterations in what is still going to be a long process of attempting to create, apply, and enforce realistic and more easily applied trustworthiness criteria. Any set of criteria is likely to be incomplete. It is also likely to be difficult to interpret, and possibly overly explicit and restrictive. Nevertheless, detailed parameterizable Common Criteria protection profiles, evaluation criteria, and representative carefully documented successful evaluations are urgently needed that more thoroughly address distributed systems and networks.

- **System architectural concepts.** More emphasis should be placed on system architectural concepts such those included in the Enlightened Architecture Concept of Section 4.3, such as the minimization of the need for trustworthiness and the the Trusted Servers and Controlled Interface (TS&CI) systems noted there for both multilevel and single-level security applications. Attempts to get developers of secure computer systems to produce commercially viable TCSEC Orange-Book B2, B3, and A1 equivalent systems have almost always been rather dismal. Adoption of the TS&CI approach could have several effects: it could permit the rapid development of multilevel-secure system complexes using off-the-shelf single-level end-user systems; it could greatly increase the ability to configure systems to suit particular application needs; it could provide some real incentives for the development of minimal (for example, starkly subsetted) special-purpose servers that are highly trustworthy, particularly for multilevel security, survivability, and other critical requirements; it could simplify system evaluations; and it could simplify system and network evolution over time. This approach could also be very useful in developing conventional 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 TS&CI approach would increase the ease with which heterogeneous system complexes can be produced. We believe that such an approach can simplify demonstrations of assurance, including formal analyses. We also believe that, by removing some of the most stringent trustworthiness requirements 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 benefits would be the ability to obtain heterogeneous distributed systems with much greater overall trustworthiness, including multilevel-secure operating environments that could be assembled as configurations of off-the-shelf commercially available single-level end-user systems and a few extremely trustworthy servers, some of which could be multilevel secure as needed.

Hardware architectural issues are also relevant. Although existing processor architectures do include some mechanisms for increased system security (e.g., more than two processor states, and coprocessors), those facilities are seldom used in mass-market operating systems. It seems obvious that those operating systems are not demanding increasing hardware mechanisms for security. However, that does not mean that research in hardware should not be pursued. Indeed, there are many hardware directions that could be useful, such as domain support, type-based addressing, certain aspects of capability-based addressing, dynamic checking, greater error detection and correction, parallelism and pipelining that do not add new security vulnerabilities, and special-purpose coprocessors (e.g., for cryptography, multilevel security, type-based addressing as in the Secure Computing Corporation LOCK architecture, and highly trustworthy servers).

- **Trustworthy networking.** The networking area desperately needs coherent efforts related to trustworthy networking, including (for example) improved network protocols, trustworthy embeddings of cryptography into systems or special-purpose hardware, trustworthy router survivability in the face of attacks and accidental outages, and network management that integrates security management, network misuse detection, operational controllability, and automated or semiautomated responses to detected network irregularities.

- **Programming languages and compiler technology.** Most existing programming languages and compilers in widespread common use are typically dangerous because of the ease with which security flaws can arise, or because they are inherently difficult to use wisely, or both. Although we certainly do not recommend the creation of yet another committee-generated language (as was the case for Ada, although under unusual high-tech constraints), or continued extensions on languages that were not explicitly created for the writing of trustworthy software (e.g., C and its offspring), much has been learned about how to define relatively sound languages (e.g., Euclid, Modula 3, ML, and to some extent Java — especially in the context of the Java Virtual Machine environment). Perhaps a detailed assessment would be worth pursuing of what kind of a programming language might really enhance trustworthiness while simultaneously permitting relative ease of programming.

- **Automated and semiautomated recovery in systems and networks.** Clearly, it is desirable to design systems whose architecture is inherently resistant to crashes and other events that require manual intervention for recovery. However, such resistance is not always achievable. In the most difficult cases, huge operational benefits would result from autonomous systems and networks with capabilities for self-diagnosing, self-healing, self-reprotecting, self-reconfiguring, and self-optimizing — without the intensive hands-on intervention that is required today. However, this should require much greater attention to anticipating all possible emergency situations and their responses.

- **Operations, system administration, and maintenance.** An enormous burden is placed today on system administrators, critical-system operators, development managers, and other information technology professionals. Some of that burden can be reduced by better system trustworthiness, by sensible human interfaces that are easier to use, robust, and user-tolerant, and by the ability to handle automated self-recovery (as in the previous bullet). On the other hand, if systems become largely autonomous, and if significantly fewer administrators are required, then there will be risks that most of the remaining system administrators will be less prepared to deal with the most unusual events — because they will not have had experience in coping with such emergencies. Thus, a desire for systems that require less skilled administration under routine adverse circumstances may lead to systems that are less easy to administer when something does go wrong. Overall, the human interface issues deserve much greater emphasis in research and development than hitherto, including well-designed command-line interfaces that avoid many of the pitfalls of GUIs.

- **Tools.** More emphasis is needed on the realistic usability of ad hoc as well as formally based methods, together with their supporting tools. There are many potentially useful requirements 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 sensible and realistic 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 developers of real-time systems, incorporation of temporal logics or other approaches to representing real-time issues would also be desirable (although less accessible to conventional programmers).

- **Well-documented worked examples.** Readable and easily understood tutorial documents such as [62, 330] are essential for those who are able to use formal-methods environments. Carefully worked and well-documented examples of principled real system developments with high assurance would be extremely helpful, including those developments with serious use of formal methods. Benefits would accrue not only to the developers of those systems, but also to other people who could subsequently emulate successful developments and analyses.

- **Testbeds.** Architectural frameworks need to be developed that can support the establishment of networked system testbeds for exploring new system concepts, demonstrate the effectiveness of various approaches for developing highly evolvable heterogeneous systems, enable experimentation, and provide detailed studies of the relative merits of alternative architectures — including examining many of the recommendations enumerated in this report. For example, it could be very illuminating to have such a testbed to probe the strengths and limitations of IPSEC and IP Version 6. Such testbeds could also be useful for experiments regarding the effectiveness of various strategies for establishing and enforcing various privacy policies, including those related to database management and other security-related applications. Of course, there is a caveat that must be kept in mind: testbeds and testbed demos generally do not fully flex the things that can go wrong. Consequently, there is a real risk of attributing to them more than should reasonably be implied.

8.3 Some Specific Recommendations

The issues discussed in Section 6.2 and the general recommendations of Section 8.2 suggest various opportunities for the future. Each typically has both short-term and long-term implications, although some require greater vision and farsight than others. The typical myopia of short-term efforts almost always seems to hinder long-term evolvability, as discussed in Section 7.1. Incremental attempts to make relatively small changes to systems that are already poorly designed are less likely to converge acceptably in the long term.

We suggest in this report that considerable gains can be achieved by taking a fresh view of secure-system architectures, while also applying formal methods selectively — particularly up front, where the potential payoffs are thought to be 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 common sense, knowledge, experience, and even formal methods, if appropriate. However, there is still much worthwhile long-term research to be done, particularly where it can reduce the risks of system developments (which include cost overruns, schedule slippages, and in some cases total project failure) and increase the overall chances of success.

- **Principled prototypes and illustrative worked examples.** There is a great need for examples of real systems for which there are explicit critical requirements, sound architectures, well-developed software, and sensible metrics for assurance and determination of the success of the effort, all of which need to be carefully documented. This need is particularly important for systems in which high assurance has been sought through the use of formal methods as an integral part of system development (rather than through *post-hoc* analyses). These examples should be realistic, carefully elaborated, and thoroughly documented. Where formal methods are used, they should observe the concept of Pervasively Integrated Assurance (PIA) noted in Section 8.2, applied up front as well as throughout the development process, although incrementally it is of course wise to begin with analyses of requirements and specifications. Some of the more far-reaching examples should naturally extend to hierarchical mappings among different layers and explicit representations of properties at different layers and explicit representations of the dependencies on lower layers.

  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. (Some readers may be annoyed at our frequent mention of formal methods and their potential benefits. Formal methods are clearly not for everyone. However, for certain highly critical system developments, they can be extremely effective when used wisely.)

  Here are examples of a few potential problem areas and would-be approaches for which specific R&D efforts might be particularly valuable.

  – **A very complicated environment.** Apply some of this report to the Global Information

Grid. The vision of DoD's GIG (noted at the end of Section 7.1) represents an enormous challenge that would lend itself to the overall recommendations of this report. That vision cries out for a detailed and comprehensive set of short-term and long-term requirements (with appropriate transition strategy), the establishment of a viable evolvable heterogeneous enlightened architecture that can accommodate all of those requirements including a mix of MLS, MILS, and MSL subnetworks (with a detailed strategy for the refinement of the architectural concept), a highly principled development strategy, and a completely integrated approach to the assurance that is required for a pervasively multilevel-secure highly trustworthy survivable networked system of systems. (For background, see [261].)

- **A much more controlled distributed environment.** Develop a trustworthy networked distributed operating system environment that provides some well-defined but nontrivial aspects of security, reliability, survivability, ease of operation, and autonomous recovery, taking advantage of but transcending past research such as the Digital Distributed System Security Architecture [128], SDSI/SPKI [112, 315], survivability [133, 259, 261], and many other stepping stones.
- **A separation kernel.** Develop a reusable interoperable Rushby-style separation kernel with an appropriate interface, 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.
- **Enlightened MLS/MSL/MILS architectures.** Elaborate on one or more of the architecture families defined in [259], including the Proctor–Neumann trusted-server/controlled-interface (TS&CI) approach noted in Section 4.3, and develop prototypes for those that appear to have the greatest potential — for single-level environments as well as multilevel-secure applications. Despite all the shortcomings of the MLS efforts in the previous decades, the existence of some practical distributed and networked multilevel-secure environments would be extraordinarily useful.
- **A sound cryptographic embedding.** Perform a thorough modeling and analysis of a cryptographic encapsulation (in hardware and/or in software) — including making explicit all assumptions on which the confidentiality and integrity of the crypto-based security depend, and all reasoning based on those assumptions.
- **A robust messaging system.** Extend the effort begun in [181], in several possible directions: (1) refining the existing representation of the MISSI security policy or one of its successors in the Defense Messaging System area; (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. 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.
- **Common Criteria elaborations.** Establish some meaningful Common Criteria profiles encompassing security, reliability, survivability, and other critical requirements in a parameterizable way that would be applicable to many system developments.

- **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, as well as better understanding of the limitations.
- **Integration of model checking.** A somewhat longer-term but still near-term goal involves incorporating model-checking tools compatibly into existing analysis tools, as was done in the Berkeley subcontract summarized in Appendix A.
- **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.
- **Modularization and interoperability of different tools.** At present, various tools for formal methods operate in their own environments, but do not interoperate with others. It would be very useful to have modularized components of these toolsets that could interoperate across institutional boundaries.
- **Reasoning about requirements.** However, it is clearly desirable to have well-defined requirements that can be precisely stated and whose consistency can be analyzed. Some formal reasoning would be particularly desirable where critical requirements are involved, especially if the requirements are modular and can be reused in other contexts. (For development efforts that begin with ill-defined and poorly stated requirements, there is relatively little hope for applying analytic techniques.)
- **Reasoning about designs, specifications, and interfaces.** When nontrivial specifications are part of the development process prior to implementation, those specifications should advantageously be such that they are amenable to analysis — for example, for being syntactically well formed, locally consistent with requirements, having consistent interfaces and being interoperable across multiple specifications, and ultimately consistent with overall system requirements. Formal or semiformal specifications can aid significantly toward this end. Properties of interfaces (Chapter 5) and the functionality that they represent (Chapter 4) are both accessible to such analysis.
- **Reasoning about compositions.** Significant effort should be devoted to a generalized theory of compositions and the property transformations they induce. This work should encompass system and network configurations generally, the interconnections involved in TS&CI architectures, the interposition of trusted gateways, networking, and the necessary criteria to facilitate evaluation of modular systems, and reasoning about emergent properties. For example, it is highly desirable that analytic tools be developed that can determine the extent to which a set of modules can be composed without deleterious effects, as noted in Sections 6.2 and 7.7.2.
- **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 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 work begun by Moriconi [246], to provide a framework for reasoning about design changes, although that effort has spawned many more recent advances.) 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.

- **Reasoning about evaluations.** Based on reasoning about requirements, specifications, compositions, and implementations noted above, it is also desirable to reason about the evaluations of subsystems and systems, and how the evaluations might compose. Also of enormous potential value would be reasoning about the effects of changes that might occur throughout the development cycle. Past efforts to attack this problem have fallen far short of what is needed to avoid the pitfalls of upgrades and evolutionary changes.

In those efforts, an appropriate mix of experienced is recommended — such as systems, development, formal methods, and analysis tools.

8.4 Architectures with Perspicuous Interfaces

Section 5.4.5 lists several gaps in existing analysis techniques and tools. Each of those gaps suggests various research and development areas in which new techniques and significant enhancements of existing techniques could advantageously be pursued, developed, and explored experimentally, with the goal of achieving analysis aids that can significantly improve the understandability of interfaces and system behavior.

Fundamentally, a combination of techniques is essential, encompassing better system architectures, better use of good software engineering practice, better choices of programming languages, better analysis techniques for identifying deficient perspicuity, a willingness to iterate through the development cycle to improve perspicuity, and greater discipline throughout development and operation. Here are a few specific suggestions.

- **Integral interface design.** Provide a wider perspective of understanding that encompasses and interrelates system and network architecture, software design and implementation, and operational characteristics.

- **Flaw avoidance.** Avoid characteristic design flaws and software bugs, through a combination of architecture, software engineering, programming language choice, enforceable constraints on programming style, design for interface perspicuity, static analysis, and dynamic interpretation of anomalies.

- **Dependency analysis.** Provide detailed analyses of the dependencies among different design entities, different source-code modules, and different object-code components, as well as the potential impact of those dependencies on perspicuity, security, reliability, and so on.

- **Concurrency analysis.** Provide detailed analyses of concurrency-related flaws and failures and how to cope with them. (Consider the Edinburgh Concurrency Workbench.)
- **Pervasive analysis.** Integrate various approaches throughout the development cycle and operation, including analysis approaches that integrate system architecture, system development methodologies, requirements, specifications, programming language syntax and semantics, object code characteristics, and operational considerations. This is a far-sighted and difficult item.
- **Privacy policies.** Provide detailed privacy policy specifications relating to how interface information should be made available. Ideally, what is needed is a framework within which specific policies can easily be specified, understood, and implemented — through a combination of dynamic analysis and constructive enforcement.

8.5 Other Recommendations

The recommendations of the previous sections focus primarily on research and development with potential impact on the development, operation, and maintenance of trustworthy systems and networks. Some other issues with less R&D content could also be extremely effective.

- **Comparative studies.** Comparative studies are needed to explore the explicit benefits that can result from the serious application of this report to system and network development. In particular, substantive evidence regarding the efficacy of consistent use of principles, Pervasively Integrated Assurance, and formal methods throughout system developments would be very valuable. 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 under way 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. The relative merits and disadvantages of open-box software (under varying licensing arrangements) also need to be objectively documented [122]. (Protests by certain closed-source developers that open-source paradigms will destroy the world as we know it seem somewhat overinflated.)
- **Operations, system administration, and maintenance.** As noted above, the burden on operational staff and management is becoming overwhelming. In addition to architectural approaches such as autonomous component systems that might reduce these problems, much greater awareness of the vulnerabilities, threats, and risks is also necessary on the part of the involved personnel. For example, as a consequence of the serious burden on an inadequate number of system administrators, there is a widespread movement to outsource operational support. This represents a significant potential threat to system trustworthiness in critical systems, and its would-be cost-effectiveness needs to be balanced by the security risks. (See Section 7.10.2.)
- **General awareness.** The educational issues relating to university students who are would-be system developers are noted above. But there is a much broader need for increased awareness

outside of the information technology profession — for example, on the part of legislators and their staffers, law enforcement and judicial officers, Pentagon brass and other military personnel, government officials across the board, and indeed computer user communities overall.

- **Education, academic research, experience, and training.** Although education is not an R&D goal on its own, it has far-reaching implications for trustworthy systems development. In particular, R&D results need to play an important place in teaching and training. More emphasis should be placed on the education of students and industry employees to help them appreciate the potential practical utility of principled development (including formal methods) applied to real systems, as well as their theoretical beauty (for example, see [121]). For this to be successful, teachers must have a better understanding of the fundamental issues of software engineering concepts and of discrete mathematics and logic, and those topics must permeate the instruction and training. Much greater emphasis is also needed in training system administrators, irrespective of the extent to which autonomous systems and networks might become a reality. Also needed is a much better understanding of some of the research directions noted here, in motivating graduate students toward pursuing more relevant PhD thesis topics.

  The unfortunate lack of a more ubiquitous systems perspective in educational curricula represents a fundamental problem in education and training at many universities. Undergraduate and graduate computer science and computer engineering programs desperately need to have requirements engineering, system and network architecture, good software engineering practices, security, reliability, survivability, and other related concepts pervasively integrated into course curricula.

  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, which implies that students tend to develop very little system sense. Good software engineering (as opposed to overly simplistic panaceas) is rarely emphasized — especially from any rigorous basis — and seems to be considered more or less irrelevant in favor of a predilection toward programming in the small. (There are of course some exceptions.) Security, reliability, survivability, human safety, and other critical-system issues seem to be widely underrepresented. Unfortunately, the practical needs of system developers seem to be the tail trying to wag the dog. Our universities must embody more diversities (or even multiversities), teaching much more than just C, Unix, Windows, HTML, and XML. 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 steadily narrowing rather than broadening experiences. A greater appreciation of the need for system perspectives should permeate education at all levels.

All in all, the existence of systems and networks that are inherently more trustworthy — through sound architectures, better development practice, and other approaches discussed in this report — would greatly simplify the vicissitudes of system operation and administration. By reducing the labor-intensive efforts required today, we could thereby greatly improve the overall trustworthiness of our operational systems and networks.

A very useful recent assessment of future research directions [339] has been compiled for DARPA by Fred Schneider on behalf of Jay Lala, as a set of nicely annotated slides. It provides a complementary view to that presented here, although there are (not surprisingly) many areas of overlap. In particular, it outlines several approaches to robustness: runtime diversity (as opposed to computational monocultures), scalable redundancy (especially asynchronous), self-stabilization, and natural inherent robustness (as is found in various biological metaphors).

Chapter 9

Conclusions

The merit of virtue is consolidated in action. Cicero

9.1 Summary of This Report

This report addresses the main elements of the DARPA CHATS program — composable high-assurance trustworthy systems — with emphasis on providing a fundamental basis for new and ongoing developments having stringent requirements for trustworthiness. We believe that significant benefits can result by emphasizing the importance of the judicious use of principles, the fundamental need for inherently composable architectures as the basis for development, and the underlying need for a highly principled development process. We believe that principled development can also contribute to improved operation, maintenance, and long-term evolution. However, these benefits depend largely on the education, training, and experience of its practitioners, and on a continuing flow of relevant research and development that is suitably motivated by well-defined and realistic requirements.

9.2 Summary of R&D Recommendations

If the road to hell is paved with good intentions, then by duality, the road to heaven must be paved with bad intentions. However, the road to good systems development and good management practice is evidently infinitely precarious, no matter which route is taken. PGN

Chapter 8 summarizes some of the potentially far-reaching areas for future R&D, at a relatively high layer of abstraction. Of these recommendations, the most important are perhaps the following.

- **Principled research and development.** Research and development efforts should be encouraged if not required to declare the principles to which they aspire to adhere, should honor those principles where practicable, and provide metrics to determine the degree of success. This recommendation should be reflected in future procurements.

- **Parameterizable reusable requirements.** We need to develop parameterizable sets of composable requirements for systems with critical trustworthiness, whereby specific sets of requirements can be explicitly tailored for any particular system within a wide range of realistic system and network developments.

- **Parameterizable reusable composable components.** Based on the worthiest of research concepts and architectures, we must establish a collection of principled composable interoperable distributed subsystems out of which trustworthy systems and networks can be predictably and readily developed, and we must carefully document any potential adverse interactions that may result from less-than-seamless compositions. These components should also include well-designed, easy-to-use, efficient applications and user-interface subsystems, so that entire systems can be readily established.

- **Parameterizable reusable distributed and networked architectures.** We need to establish families of composable architectures that can provide frameworks for a wide range of systems, and that can be effectively adapted to specific sets of requirements — including not just aspects of trustworthiness, but also maintainability, evolvability, and heterogeneous interoperability among systems, networks, and applications (to name just a few). Several potential architectural approaches are noted in the first major bullet in Section 8.3.

- **Perspicuous interfaces.** Further work is needed to enhance the design and implementation of system and network architectures that pervasively integrate perspicuous interfaces into system architectures, as discussed in Chapter 5.

- **Automated and formally based analytic tools.** Tools for static and dynamic analysis need to be integrated systemically into development and operational practices, not just to identify security flaws but also to identify potential limitations or breakdowns in intended compositions.

- **Pervasively Integrated Assurance (PIA).** As noted in Sections 6.1 and 8.2, ideally, a comprehensively wide range of formal and informal assurance techniques must be an integral part of the development cycle and persistently invoked during operation, maintenance, administration, and long-term evolution. The results of assurance analyses should be used to leverage the entire development process and subsequent system evolution.

- **Existence proofs of the efficacy of principled approaches.** As noted in Section 8.2, well-documented highly principled development successes would be extremely valuable — along with detailed analysis of some major failures (for example, elaborating on some of the cases considered in [260] and [267]).

9.3 Risks

"The essence of risk management lies in maximizing the areas where we have some control over the outcome while minimizing the areas in which we have absolutely no control over the outcome and the linkage between effect and cause is hidden from us."
Peter L. Bernstein [38], p. 197

There are many risks that need to be considered. Some risks are intrinsic in the development process, while others arise during operation and administration. Some relate to technology, whereas many others arise as a result of human action or inaction, and even environmental causes in some cases. Some involve risks of systems that fail to do what they are expected to do, whereas others involve risks that arise because an entirely unexpected behavior has occurred that transcends the normal expectations.

The Bernstein book quoted above is slanted largely toward a perspective of financial risk management, but an understanding of the nearly millennium-long historical evolution that it presents is also quite appropriate in the context of trustworthy systems and networks. Indeed, that quote echoes our view of the importance of carefully stated comprehensive requirements, sound architectures, principled developments, and disciplined operations as strong approaches to avoiding risks that can be avoided, and to better managing those that cannot.

Neumann's book, *Computer-Related Risks* [260], provides a complementary view of the origins of those risks and some constructive ways on how to combat them. Various articles in the ACM Risks Forum, the *IEEE Spectrum,* and the *Communications of the ACM* monthly Inside Risks columns have documented selected failures and a few successes. However, Henry Petroski has often remarked that we seldom learn much from what appear to be successes, and that we have a better opportunity to learn from our mistakes — if we are willing to do so. This report attempts to do exactly that — learn from past mistakes and dig more deeply into approaches that can reduce the risks related to trustworthiness in critical systems and networks.

9.4 Concluding Remarks

> *Hindsight is useful only when it improves our foresight.* William Safire (*The New York Times,* 6 June 2002)

There are many lessons to be learned from our past attempts to confront the obstacles to developing and consistently operating systems with stringent requirements for trustworthiness. This report is yet another step in that direction, in the hopes that it is time for constructive action.

We began Chapter 1 of this report quoting Ovid:

> *We essay a difficult task; but there is no merit save in difficult tasks.*

We began Chapter 4 on principled architectures quoting Juvenal:

> *Virtue is praised, but is left to starve.*

We began Chapter 9 quoting Cicero:

> *The merit of virtue is consolidated in action.*

Each of these three two-millennium-old quotes is still extremely apt today.

With regard to Ovid, the design, development, operation, and maintenance of trustworthy systems and networks represent some incredibly difficult tasks; however, we really must more assiduously confront those tasks, rather urgently. Today's commercially available systems, subsystems, and applications fall significantly short — for example, with respect to trustworthiness, predictable composability and facile interoperability, assurance, ease of maintenance and operation, and long-term evolvability.

With regard to Juvenal, it is easy to pay lip service to virtuous principles and good development methodologies, but those principles are seldom observed seriously in today's system and network developments.

With regard to Cicero, we recognize that it is extremely challenging to practice what we preach here. For example, incompatibility problems with legacy systems tend to make exceedingly difficult the kind of cultural revolution that is likely to be necessary to achieve trustworthy systems in the future. However, it is our sincere hope that this report will help consolidate some of its readers into action toward much more disciplined and principled design and development of composable trustworthy systems and networks, with nontrivial measures of assurance. The alternatives of not doing so are likely to resemble something conceptually approaching the decline and fall of the Roman Empire.

The U.S. DoD **Global Information Grid** (GIG), (discussed briefly at the end of Section 7.1) is a work in progress that illustrates the importance of far-sighted thinking, principles, predictable composability, and a viable system-network architecture concept. As noted earlier, the planning and development necessary to attain the desired requirements also strongly suggest the need for long-term vision, nonlocal optimization, and whole-system perspectives (see Sections 7.1, 7.2, and 7.3, respectively). Considering the very considerable difficulties in achieving high-assurance trustworthiness over the past four decades, and the dismal record noted in this report, the challenges of finally overcoming the lurking hurdles in the next 16 years are indeed daunting. As noted at the end of Section 7.1, the content of this report is fundamental to such efforts as the GIG.

Once again, we reiterate a mantra that implicitly and explicitly runs through this report: In attempting to deal with complex requirements and complex operational environments, **there are no easy answers.** Those who put their faith in supposedly simple solutions to complex problems are doomed to be disappointed, and — worse yet — are likely to seriously disrupt the lives of others as well. If the principles discussed here are judiciously applied with a pervasive sense of discipline, systems and networks can be developed, administered, and operated that are significantly more robust and secure than today's commercial proprietary mass-market software and large-scale custom applications. Perhaps most important, complexity must be addressed through architectures that are composed of well-understood components whose interactions are well understood, and also through compositions that demonstrably do not compromise trustworthiness in the presence of certain untrustworthy components. The approaches offered herein are particularly relevant to developers of open-source software, although they are equally important to mass-market developments. Those approaches may seem to be difficult to follow, but selective application of whatever may be appropriate for given developments should be considered.

In concluding this report on how we might develop systems and networks that are practical and realistically more trustworthy, the author recognizes that he has given his readers a considerable amount of seemingly repetitive evangelizing. Although such arguments by many authors seem to have fallen on deaf ears in the past, hope springs eternal. Besides, the risks of not taking this report to heart are greater now than they ever have been.

Acknowledgments

I am especially grateful to Doug Maughan, who sponsored the CHATS program and was its Program Manager for the first two years of our project (when he was at DARPA). His vision and determination made the CHATS program possible, and his inspiration and encouragement have been instrumental in this project. In addition, Lee Badger (also at DARPA) provided the impetus for the work on perspicuous interfaces on which Chapter 5 is based.

I enormously appreciate various suggestions from members of our project advisory group (Blaine Burnham, Fernando Corbató (Corby), Drew Dean, George Dinolt, Virgil Gligor, Jim Horning, Cliff Jones, Brian Randell, John Rushby, Jerry Saltzer, Sami Saydjari, Olin Sibert, David Wagner) and other individuals whose comments have been very helpful directly or indirectly in guiding the progress of this report.

In particular, Drew Dean suggested several examples of conflicts within and among the principles, and exposed me to Extreme Programming; we had many ongoing discussions on composability, architecture, and other subjects. He was instrumental in our joint work on perspicuous interfaces (Chapter 5). Virgil Gligor early on reminded me of several important papers of his on composability; his contributions to the seedling effort on visible interfaces for Lee Badger strongly resonated with that of Drew Dean and me. Virgil also generously contributed the material on which Appendix B is based.

Sami Saydjari offered numerous valuable comments during the first year of the project. Blaine Burnham drew my attention to the documents on composability from the 1992 time frame noted in the bulleted item on other past research on composition in Section 3.4. Jim Horning offered wonderful suggestions based on his long experience — including the quote from Butler Lampson at the beginning of Section 2.3, and profound thoughts on Chapter 7, which I gladly incorporated. Eugene Miya offered the quote from Gordon Bell at the beginning of Section 3.7. Tom Van Vleck expressed various doubts as to the efficacy of the object-oriented paradigm. Many years of interactions with Brian Randell have resulted in our pursuing similar research directions; his insights have influenced this report both directly and indirectly. Some detailed comments from Fred Cohen on an early draft of the composability chapter gave me considerable food for thought.

I am delighted with the results of the subcontract to the University of California at Berkeley and thankful to David Wagner for his excellent leadership, to Hao Chen for carrying out most of the work, and to Drew Dean for his vital participation in that effort. The material in Appendix A summarizes the results of the Berkeley subcontract, plus some further work by Hao Chen conducted during the summer of 2003 at SRI and subsequently at Berkeley. Appropriately, Chen's work uses an approach to static analysis of would-be robust programs that itself contributes significantly to the composability of the analysis tool components.

Appendix A

Formally Based Static Analysis (Hao Chen)

Formally Based Static Analysis for Detecting Flaws

This appendix summarizes the results of the first-year project subcontract to the University of California at Berkeley and some subsequent related work, culminating in the thesis work of Hao Chen.

A.1 Goals of the Berkeley Subcontract

The one-year CHATS project subcontract task involved a short-term potentially high-payoff approach, with static analysis capable of detecting fundamental characteristic common security vulnerabilities in source code. The approach combines models of the vulnerabilities with model checking related to the source code. The approach is intentionally open-ended, with linearly increasing complexity of composability as various new vulnerability types are accommodated. The team for this task includes Professor David Wagner and his graduate student Hao Chen in the Computer Science Department at the University of California at Berkeley, with participation of Drew Dean at SRI and supervision of Peter Neumann.

A.2 Results of the Berkeley Subcontract

One of the things that makes computer security challenging is that there are many unwritten rules of prudent programming: “Don’t do X when running with root privileges.” “Always call Z after any call to Y.” And so on. These issues contribute to the prevalence of implementation errors in security-critical systems.

In this project, our goal was to help reduce the incidence of implementation vulnerabilities in open source software by developing an automated tool to warn when programmers violate these implicit rules of thumb. We have done so. Our hypothesis was that new ideas in software model checking could prove very helpful in this problem, and our research goal was to experimentally assess the utility of our methods. Our studies give strong evidence in favor of the benefits of this style of automated vulnerability detection and avoidance. This project was undeniably a high-risk, high-payoff, novel combination of theory and practice, but we feel that it has already been very successful.

In this appendix, we give some details on our progress during the year. We have also written research papers [77, 78] on our work, which provide further technical details. Here we give a high-level overview of our results and experimental methodology.

First, we developed a general platform for automatically scanning C source code and verifying whether it follows these rules. We developed new techniques, based on model checking of push-down automata, for this problem, and we built a prototype implementation of our algorithms. Our tool, called MOPS, supports compile-time checking of large C programs against temporal safety properties. Please note that the latter two sentences hide a significant amount of investment and implementation work to achieve this goal, but as we will argue next, it has paid off nicely.

Next, we selected several examples of implicit rules of defensive coding. Several of our rules studied proper usage of the privilege management API in Unix, namely, the `setuid()`-like calls, and several rules associated with this API. The specific guidelines selected were as follows:

1. Programs that drop privilege should do so correctly: They should call `setgid()` before `setuid()`. Moreover, they should avoid the Linux capability bug: they should be aware that, on some older versions of Linux, the `setuid()`-like calls may fail to drop privilege in certain special situations.

2. Programmers should avoid situations where a `setuid()`-like call may fail. Such situations are dangerous, because these failure modes are often not adequately tested.

3. Programmers should avoid so-called "tractorbeaming attacks". In a tractorbeaming attack, unexpected interactions between signal handlers, `setjmp()/longjmp()`, and Unix uid's can create security vulnerabilities. To avoid this, programmers should ensure that every call to `longjmp()` will be done in exactly the same security context as the preceding call to `setjmp()`, no matter what intervening code path may be followed between the two.

4. When writing a setuid program, one should avoid making any assumptions about the environment inherited from the parent process. In particular, file descriptors 0, 1, and 2 are usually bound to an input/output device (e.g., the user's terminal) in normal operation, but for setuid programs, there is no guarantee that the parent process will abide by this convention. If this fact is overlooked, there is a specific class of vulnerabilities that can ensue: for instance, if a setuid program calls `open("/etc/passwd", O_RDWR)` and then calls `printf()` to display some output to the user, it may be possible for an attacker to corrupt the password file by calling the `setuid` program with file descriptor 1 closed, so that the `open()` call binds the password file to `fd 1` and the `printf()` unintentionally writes to the password file rather than to the screen.

This is by no means an exhaustive list. Rather, the rules listed above were selected to be representative, of interest to open-source practitioners, and theoretically challenging to automatically check.

Then, we devoted effort to experimentally assessing the power of our technique. We chose several large, security-critical programs of interest to the open source community as a target for our analysis. In several cases, we were able to find older versions of these programs that contained security vulnerabilities arising from violations of the above rules. The selected programs include `wu-ftpd`, `sendmail`, and `OpenSSH`. We set out to apply our tool to check the above rules to these programs.

We started by codifying the above rules in a form understandable by our modelchecker, MOPS. We described them as finite state automata on the traces of the program. Along the way, we discovered that we needed to solve an unanticipated research challenge: What are the exact semantics of the Unix `setuid()`-like system calls? We realized that these semantics are complex, poorly documented, and yet critical to our effort. To reason about the privileges an application might acquire, we must be able to predict how these system calls will affect the application's state. We spent some time working on this problem, because it does not seem to have been addressed before.

We also developed new techniques for automatically constructing a formal model of the operating system's semantics with respect to the `setuid()`-like system calls. In particular, our algorithm extracts a finite-state automaton (FSA) model of the relevant part of the OS. This FSA enables us to answer questions like "If a process calls `setuid(100)` while its effective `userid` is root, how will this affect its userids?" and "For a process in such-and-such a state, can `seteuid(0)` ever fail?".

Our new techniques, and the FSA models they produce, are useful in several ways. First, they form one of the foundations of our tool for static analysis of applications. Because we have an accurate model of both the application and the operating system, we can now predict how the application will behave when run on that operating system. Second, they enable us to document precisely the semantics of the setuid API on various operating systems, which we expect will help open-source programmers as they develop new applications. Third, they enable us to pinpoint potential portability issues: we have constructed FSA models for Linux, Solaris, and FreeBSD, and each difference in the respective FSAs indicates nonportable behavior of the setuid API that application programmers should be aware of.

Our paper [78] on constructing formal models of the operating system also documents several subtle pitfalls associated with privilege management. We expect that this work will help developers of open-source applications and maintainers of open-source operating systems to improve the quality and security of their software.

With this research challenge tackled, we were now able to encode rules (1) to (4) in a form readable by MOPS, and we used MOPS to check whether the applications we selected follow the rules. MOPS found several (previously known) security vulnerabilities in these programs, as follows:

- MOPS found security holes in earlier versions of sendmail. In sendmail 8.10.1, MOPS found an instance of the Linux capabilities bug. In `sendmail` 8.12.0, MOPS found that sendmail can fail to drop privilege in group IDs properly, due to a violation of rule 1).

- We used MOPS to verify that OpenSSH 2.5.2 properly uses the `setuid()`-like system calls in the sense that no uid-setting system call can fail.

- MOPS found a tractorbeaming bug in wu-ftpd version 2.4 beta 12. This in fact was a source of a security hole in this older version of `wu-ftpd`, and was later fixed. MOPS also confirmed that the latest version of `wu-ftpd` correctly obeys our rule regarding `setuid()`, `longjmp()`, and signal handlers.

- Experiments are still under way with respect to rule (4), but we found security bugs in several programs, including `login` and `crontab` on Linux.

In each case, MOPS ran efficiently, taking at most a minute or two to scan the source code. Since each of these application programs is of nontrivial size, this is a very positive result.

This experimental evidence indicates that MOPS is a powerful tool for finding security bugs, for verifying their absence, and for ensuring that various principles of good coding practice are observed. We have publicly released the MOPS tool under the GPL license at http://www.cs.berkeley.edu/˜daw/mops/. Our current prototype includes the compiler front end, the modelchecker, and a primitive user interface. However, we should warn that there are several known limitations: the current release does not include an extensive database of rules to check; also, the user interface is rather primitive, and intended primarily for the expert programmer rather than for the novice. We hope to address these limitations in the future.

Along the way, we developed several theoretical and algorithmic techniques that may be of general interest. First, we extended known modelchecking algorithms to allow backtracking: when the modelchecker finds a violation of the rule, our algorithm allows finding an explicit path where the rule is violated, to help the programmer understand where she went wrong.

Second, we developed a compaction algorithm for speeding up modelchecking. Our observation is that, if we focus on any one rule, most of the program is usually irrelevant to the rule. Our compaction algorithm prunes away irrelevant parts of the program — our experience is that compaction reduces the size of the program by a factor of 50x to 500x — and this makes modelchecking run much more efficiently.

Our compaction algorithm gives MOPS very good scalability properties. In principle, the time complexity of pushdown modelchecking scales as the cube of the size of the program (expressed as a pushdown automaton) and the square of the size of the rule (expressed as a finite state automaton). However, in practice, the running time is much better than this would indicate, because our compaction eliminates all irrelevant states of the program. With compaction, the running time now depends only on the cube of the size of the relevant parts of the program, and as argued above, this is generally a very small figure.

As a result, MOPS is expected to scale well to very large programs. We have already shown that it runs very fast on fairly large programs (on programs with 50,000 lines of code or so, modelchecking runs faster than parsing). Moreover, MOPS enables programmers to verify global properties on the entire system, even though each programmer may know only local information about one part of the system. Thus, our approach is very friendly to composition of large systems from smaller modules.

In summary, we have developed, implemented, and validated new techniques for improving the quality of security-critical software. Our tool is freely available. This points the way to improvements in security for a broad array of open-source applications.

The relevant papers are "Setuid Demystified" [78], by Hao Chen, David Wagner, and Drew Dean:
http://www.cs.berkeley.edu/˜daw/papers/setuid-usenix02.ps. and "MOPS: An Infrastructure for Examining Security Properties of Software" [77], by Hao Chen and David Wagner:
http://www.cs.berkeley.edu/˜daw/papers/draft-mops.ps.

A.3 Recent Results

Subsequent to the first-year subcontract, Hao Chen continued to work on MOPS and its applications for his Berkeley doctoral dissertation. MOPS acquired its first external user, the Extremely Reliable Operating System (EROS) project at Johns Hopkins University [351]. The EROS project has already uncovered multiple, previously unknown coding errors by using MOPS to analyze the EROS kernel. Based on user feedback, we are working on tuning the performance of the tool. Work has focused on some minor modifications to key data structures to reduce memory pressure on the garbage collector (MOPS is implemented in Java). A small amount of work produces a very large payback: our initial tests indicate a 300%-400% speed improvement over the earlier version. This improvement has recently been completed, and was shipped to Johns Hopkins. These results enhance MOPS's already impressive scalability for analyzing real-world software such as Sendmail and OpenSSH.

Hao Chen spent the summer of 2003 at SRI, funded by SRI project 11679, under Contract N00014-02-1-0109 from the Office of Naval Research. Building on the prior work on modeling the setuid family of system calls in Unix-like operating systems, the above-mentioned programs were examined for security problems relating to uid handling, concentrating on global properties of the programs. The concentration on global properties was chosen for two reasons: (1) Local properties can easily be checked with less sophisticated tools. Why swat a fly with a sledgehammer? (2) Global properties, being more difficult to check, for both humans and machines, have had poorer tool support, so the probability of interesting discoveries is higher. The experience gained using MOPS to check more properties of more software also uncovered areas in which MOPS needed further improvement.

In addition to the above mentioned improvements in MOPS, Hao Chen applied MOPS to study selected security properties of widely used open source software. The programs studied included BIND, Sendmail, Postfix, Apache, and OpenSSH. To demonstrate the power and utility of MOPS, these programs were model checked for each of five properties, (a) proper dropping of privileges, (b) secure creation of *chroot jails*, (c) avoidance of file system race conditions, (d) avoiding attacks on standard error file descriptors, and (e) secure creation of temporary files.

Hao Chen's work at SRI during the summer 2003, under the guidance of Drew Dean, resulted in the discovery of several hitherto undetected security problems in these programs, as well as the identification of other flaws that had been previously discovered elsewhere. The results of this application of MOPS to real programs are summarized in [75].

This work provides key capabilities for progress in information assurance. It provides a principled foundation for analyzing the behavior of programs based on traces of system calls, or, for that matter, any functions of interest. This approach to program analysis can directly take advantage of research in both model checking and static analysis to become more precise over time, something that is not directly true of ad-hoc approaches to analyzing programs for security vulnerabilities. Future improvements to underlying technology, in addition to more engineering improvements to MOPS, should allow MOPS to scale from today's ability to handle 100KLOC comfortably (substantially more than competing tools), to 1MLOC. Such scalability will be necessary for DARPA to provide an assured future for the network-centric warfighter.

Hao Chen's doctoral thesis [74] is now finished and available. Also, a recent paper by Hao Chen and Jonathan Shapiro [76] describes their experience running MOPS on EROS. In addition, a group of students in Professor Wagner's group ran MOPS on all 839 packages in RedHat Linux 9 and found many security bugs and weaknesses, being described in a new paper.

A.4 Integration of Static Checking into EMERALD

It is useful to contemplate how the software developments of the Berkeley effort could subsequently be integrated into an anomaly and misuse detection system such as provided by the EMERALD framework and its successor technologies. Several different approaches are potentially of interest:

- Apply the static analysis techniques and tools to the EMERALD modules to determine EMERALD's compliance with the existing and subsequently emerging Chen-Wagner formal models.
- Establish new models to be specifically suitable to analysis of the EMERALD software, and apply them to EMERALD.
- Develop a means for automatically coupling the vulnerability models with EMERALD rule bases, or otherwise incorporating the results of the analyses into EMERALD.
- Develop a coherent environment that encompasses static and dynamic checking and real-time analysis.

Appendix B

System Modularity (Virgil Gligor)

Basis for the Visibility and Control of System Structural and Correctness Properties

This appendix is based on material written by Virgil D. Gligor under DARPA Contract number MDA 972-03-P-0012 through VDG Inc, 6009 Brookside Drive, Chevy Chase, MD. 20815, telephone 1-301-657-1959, fax 1-301-657-9021, in connection with Lee Badger's Visibly Controllable Computing initiative at DARPA. Gligor's original text appeared as the appendix to an unpublished report, "Perspicuous Interfaces", written by Peter Neumann, Drew Dean, and Virgil Gligor, as part of a seedling study for Lee Badger; it is adapted as an appendix to this report with the permission of Virgil Gligor, with the explicit intent of increasing its availability to the R&D and academic communities. The earlier work of David Parnas on module decomposition [281] and on module dependence [283] (e.g., the various forms of the *uses* relation) is particularly relevant here.

B.1 Introduction

The study of Visibly Controllable Computing has the goals of reducing systems complexity and applying automated reasoning and learning techniques to create systems that can not only explain their current state but also adapt to new environments by

(1) Connecting their self-knowledge to knowledge of external-environment entities and constraints.

(2) Warning users if new mission demands cannot be satisfied.

(3) Exploring alternative configurations and reconfiguring to fit changing needs.

In general, by establishing the visibility of a system's structural and correctness properties we mean the identification of a system's components and their relationships, and the reasoning about properties like the correctness, fault tolerance, and performance. A first step toward this goal is that of investigating system modularity. This step is necessary if knowledge of system structure and state need to be gained and if systems need to reconfigure on-the-fly to satisfy changing mission requirements. Of particular interest is the investigation of properties that help (1) reconfigure systems by module replacement, and (2) establish causal correctness dependencies among modules (e.g., correctness of module interface A implies correctness of module interface B) in addition to structural visibility and reconfigurability. Of additional interest is the investigation of the properties that help reuse extant modules for different missions. Finally, of significant interest is the identification of a set of *simple, practical* tools for establishing and verifying system modularity.

Software systems that employ modular design, and use data abstraction and information hiding to achieve layering [179] offer the following advantages:

(a) Allow an incremental, divide-and-conquer approach to reasoning about correctness and other important system properties (e.g., fault tolerance, performance).

(b) Support replacement independence of system components based on well-defined interfaces and uniform reference (i.e., references to modules need not change when the modules change).

(c) Provide an intuitive packaging of system components with ease of navigation through the system, layer by layer, module by module.

(d) Allow an incremental, divide-and-conquer approach to system development, with many individuals per development team possible.

(e) Enable the reuse of software modules in different environments.

Note that Clark [81], and later Atkins [22], suggest that layering may sometimes be a potentially undesirable form of system structuring because it can lead to poor performance. Also, Nelson suggests the use of protocol "delayering" (i.e., combining protocol layers) to achieve an efficient remote procedure call mechanism [251]. Thus, while layering is a generally useful technique for system structuring, the extent of system layering depends on specific criteria, such as correctness, fault tolerance, and performance. Lampson [199] argues that the reuse of software modules is and will remain an unrealistic goal, in practice.

Early uses of layered abstraction include Multics [91, 92, 277] (with rings of protection, layering of system survivability and recovery, and directory hierarchies), Dijkstra's THE system [106] (with layers of object locking), and SRI's Provably Secure Operating System [120, 268, 269]. The PSOS hardware-software architecture provided numerous layers of abstraction for different types of objects, and distinguished between objects and their type managers. The architecture explicitly contradicts the above-mentioned Clark and Atkins claim that layering inherently leads to poor performance. For example, the PSOS layering enabled user-process operations (layer 12) to execute as single capability hardware instructions (layer 0) whenever appropriate dynamic linkage of symbolically named objects had been previously established. (The bottom 7 layers were conceived to be implemented directly in hardware, although the hardware could also encompass all or part of higher layers as well.) Thus, repeated layers of nested interpretation are not necessarily a consequence of layered abstraction, given a suitable architecture. (Also, see Section 3.4 for further background on PSOS relevant to composability.)

B.2 Modularity

In this section we define the term "module," illustrate system decomposition into modules, and present several correctness dependencies among modules. The following key notions are required to define and implement modular systems:

- Module and module synonyms
- Interface versus implementation

- Replacement independence
- Reusability
- “Contains” relation
- Module hierarchy
- “Uses” relation
- Correctness dependency among modules

B.2.1 A Definition of “Module” for a Software System

In general, a module is a system component (part, unit, building block). Synonyms for “module” include “system,” “platform,” “layer,” “subsystem,” “submodule,” “service,” and “(abstract) type manager.” A software module is part of a software system and has the following six properties:

P1. Role. A module has a well-defined unique purpose or role (responsibility, contract) that describes its effect as a relation among inputs, outputs, and retained state.

P2. Set of Related Functions. A module contains all and only the functions (procedures, subroutines) necessary to satisfy its role. Each function has well-defined inputs, outputs, and effects.

P3. Well-Defined Interface. A module has an interface (external specification) that consists of the public (visible) items that the module exports:

- declarations of its public functions (i.e., those invocable from outside the module) and their formal parameters;
- definitions of its exported types and exported manifest constants;
- declarations of global variables associated with the module;
- declarations of its signaled exceptions and handled exceptions;
- definition of the necessary resources and privileges;
- rules (discipline, restrictions) for using the above public functions, types, constants, and global variables.

P4. Implementation. A module has an implementation (internal design) that details how its interface is satisfied. It should be possible to understand the interface (and role) of a module without understanding its implementation.

P5. Replacement Independence. A module implementation can be replaced without also replacing any other module implementation in the system.

P6. Reusability. A module implementation can be reused in different software systems with little or no added code.

The role of a module describes its effects or behavior on inputs. The effects of a module can be reflected in the values of outputs, the state of the module, or the state of the system. With software, for example, the state of the module or system can be represented by a set of variables (e.g., simple variables, structures). A well-defined role should have a short and clear description, preferably one sentence. A module should have a simple name that reflects its role. Typically, module roles are system unique; no two modules in a system have the same role (no duplication of role). However, the system may intentionally duplicate modules to achieve other system goals (e.g., performance, reliability).

For a module function to be well-defined, its inputs and outputs and effects should be well-defined. The name of a function should reflect its purpose. Functions should, but need not, be named. In software, for example, some functions are expanded in-line for performance reasons; also, the programming language may not have a way to express in-line expansion of named functions. Continuing the software example, the inputs and outputs of a function can be formal parameters or informal (global, environment) parameters or (request-response) messages. It should be simple to distinguish the public from the private functions (if any) in a module. It is desirable, but not necessary, that the functions of a module be nonredundant; function redundancy is undesirable but at the discretion of the designer of the system or module. Regarding the all and only nature of a module's functions, certain functions typically have a complementary twin: get-set, read-write, lock-unlock, do-undo, reserve-unreserve, allocate-deallocate, and so on.

A module interface is well-defined if it contains all and only the module assumptions that a module user needs to know. The discipline of an interface, if any, may explain a legal order in which to use the public functions. For software, a well-defined interface contains declarations of exported (public) functions, data, types, manifest constants, exceptions raised, exceptions handled, exception handlers, and, the associated restrictions or discipline [387]. It may be inappropriate or impossible to capture certain programming restrictions or discipline within programming language constructs, in which case they should be provided in associated specification or commentary. Note that a module interface includes variables that are global to that module.

A module implementation contains module-construction assumptions and programming details that a module user should not have to know; for example, order of resource use, algorithms employed.

The typical notion of replacement independence for a module is that, if the module breaks or no longer functions correctly, then if a new module with the same interface is available, we can replace the original module with the new one without replacing any other modules. However, in software systems, the notion of replacement independence has a somewhat different meaning. While replacement independence is implied by "information hiding," [281, 59] and information hiding disallows global variables, replacement independence does not necessarily rule out the use of global variables in modules provided that the global variables are explicitly defined in the module's interface, and that the dependencies among the modules using those global variables are known.

The typical notion of module reuse requires that a module be (1) general in its purpose or role, so that it is useful to users other than the few people working on the same project; (2) fully specified, so that others can use it; (3) tested, so that general expectations of quality are met; and (4) stable, in the sense that the module's behavior remains unchanged for the lifetime of the user system [199]. Other related properties of module reusability include simplicity of interface (i.e., foreign users should understand the module's interface with little effort), and customization (i.e., foreign users should be able to tailor module use by using special parameters or special-purpose programming language [199].

B.2.2 System Decomposition into Modules

The decomposition of any system into modules relies on two intermodule relations, namely, (1) the "contains" relation, and (2) the "uses" relation. These relations imply certain correctness dependencies among modules that are fundamental to the defining the module structure of a system.

B.2.3 The "Contains" Relation

Internally, a module may (but need not) contain component submodules. If it is necessary or desirable to identify a set of component parts of a module as submodules, then that set of submodules partitions (i.e., is collectively exhaustive and mutually exclusive) the parent module. The decision as to

when to stop partitioning a system into modules is generally based on designer discretion and economics — when it is no longer necessary nor desirable economically to identify and to package and to replace that subpart. Other than this, no generally accepted criterion exists for when to stop partitioning a software system into additional modules.

Applied system-wide, the "contains" relation yields a module hierarchy (i.e., tree). Nodes of the tree represent modules; arc (A, B) means that module A directly contains submodule B. The root of the tree, the whole system, is the 0-th level of the tree. The system itself should be considered as the 0-th level module. The (n+1)-th level consists of the children (direct submodules) of the n-th level. Modules with no submodules are called leaf modules. We can define a part hierarchy system as modular if the system itself, and recursively each of its subparts that we identify as should-be-modules, satisfies the definition of module.

EXAMPLE. The UNIX kernel is a software module; the system calls compose its set of related functions. The manual pages for the system calls describe the role, set of functions, and interface of the kernel. Figure B.1 shows an example of the "contains" relation, the major subsystems of the Unix kernel.

EXAMPLE. Figure B.2 shows another example of the "contains" relation, a decomposition of the File Subsystem of the Secure XENIX kernel [115] into a module hierarchy. In Figure B.2, ACL means Access Control List. The darkened boxes identify the files of source code in this design. The Superblock Service manages the attributes of a file system as a whole object. In this decomposition, the Mount Service is part of Flat File Service and not part of Directory (pathname) Service. The Mount Service maintains a short association list for substituting one (device, i-node number) pair, a "file handle," for another. The mounted handle represents a file system, and the mounted-on handle represents a directory. The Mount Service knows nothing about directories and pathnames; it knows about pairs of file handles.

B.2.4 The "Uses" Relation

In software, if a module uses another module, then the using module imports (all or part of) the interface of the used module to obtain needed declarations and definitions. We define the "uses" relation between functions and modules as follows. Function A uses function B if and only if (a) A invokes B and uses results or side effects of that invocation and (b) there must be a correct version of B present for A to work (run, operate) correctly. A function uses a module if and only if it uses at least one function from that module. A module uses another module if and only if at least one function uses that module. The "uses" relation is well-defined. From the "uses" relation we can draw a directed graph for a given level, where the nodes are same-level modules, and arc (A, B) means that module A uses module B. Also, we can draw a "uses" graph of the leaf modules.

EXAMPLE. Figures 3, 4, and 5 show an example of an intrasubsystem "uses" graph for the File Subsystem of the Secure XENIX kernel. They shows a progression of versions of a "uses" graph. Version 0 (Figure B.3) shows the entire subsystem. Version 1 (Figure B.3) shows all File Subsystem system calls in one box to simplify the picture, and shows how this layer uses all three top level services of the File Subsystem. The lines from the Flat File Service to the ACL Service (and back) show a "circular dependency" between the two; each uses the other. Version 2 (Figure B.4) replaces the Flat File Service with its three component services. Version 3 (Figure B.5) shows another level of detail of the "uses" graph. (Note the circular dependencies in Figure B.5. For approaches that help eliminate such dependencies see the PSOS abstraction hierarchy [120, 268, 269] and [179]. PSOS inherently removed such circular dependencies as a fundamental part of the architectural abstraction hierarchy.)

B.2.5 Correctness Dependencies Among System Modules

Correctness dependencies between modules are basic to describing, evaluating, and simplifying the connectivity of modules, and thus basic to system restructuring and evolution. For modules P and Q, P depends on Q, or P has a correctness dependency on Q (or "the correctness of P depends on the correctness of Q"), if and only if there must be a correct version of Q present for P to work correctly. Based on earlier work by Parnas, Janson et al. [344, 179] identify several types of correctness dependencies, which were later combined into the following three classes by Snider and Hays [358]: service, data, and environmental dependencies.

Service Dependency: "P invokes a service in Q and uses results or side effects of that service. The service may be invoked through a function call, message, or signal (e.g., a semaphore operation), or through hardware, such as via a trap." [358]

It is important to point out that not all invocations are service dependencies. "Note that if P transfers control or sends a message to Q and neither expects to regain control, nor counts on observing consequences or results from its interaction with Q, then P does not depend on Q. It is said simply to notify Q of an event (without caring about what Q does once it is notified)" [179]. In layered systems, certain upcalls [81] that provide advice or notifications can be viewed as not violating the downcall-only layering discipline if such upcalls do not correspond to correctness dependencies.

Data Dependency: "P shares a data structure with Q and relies upon Q to maintain the integrity of that structure." [358]

Modules that are either readers or writers of shared data depend on other modules that are writers of the same shared data. Thus, shared data with multiple writer modules produce mutual dependencies and increase module connectivity.

Environmental Dependency: "P does not directly invoke Q and does not share a data structure with Q but nevertheless depends upon Q's correct functioning." [358]

"One example is the dependency of most of the system on the interrupt handling subsystem. Although this is not generally called directly from the kernel, much of the kernel depends on its correct operation (e.g., properly restoring processor state at the end of an interrupt service routine) in order for the kernel to fulfill its specifications. ... In practice, we did not find that environmental dependencies presented many structural problems." [358]

Service dependencies are more desirable than data dependencies because service dependencies are explicit; if all dependencies are service dependencies, then the system call graph (the graph of what invokes what), which is usually explicit and easy to compute, represents all dependencies. By introducing information-hiding modules [281, 286, 59] throughout a system, where system data is partitioned among modules and accessed only via function (subroutine, procedure) calls, each data dependency can be converted into a service dependency.

B.2.6 Using Dependencies for Structural Analysis of Software Systems

For structural analysis, it is desirable to represent correctness dependencies between system modules with the "contains" and the "uses" relations (and graphs). As seen above the "contains" relation among modules is unambiguously defined by syntactic analysis. In contrast, the "uses" relations can be defined in three possible ways: (1) as representing all correctness dependencies; or (2) as representing only service and data dependencies; or (3) as representing only service dependencies.

Fundamentally, there is no difference between service and data dependencies since both are correctness dependencies. Further, data dependencies can (and should) be converted to service dependencies if we drive the structure toward desirable information hiding. To simplify system structure, we need to minimize correctness dependencies and eliminate all circular dependencies. To do this, we first minimize data dependencies, because they contribute to circular dependencies, then we remove other circular dependencies. The resulting measurable goal is that of eliminating global variables and acyclic structure, and minimizing the cardinality of the "uses" relation. If this "uses" relation represents all system correctness dependencies and if its graph is cycle-free, then showing correctness of the system parts in a bottom-up order (the reverse of a topological sort of the "uses" graph) leads to correctness of the system.

In practice, it is not necessarily possible, nor desirable [22], to eliminate all structural imperfection (i.e., all globals, some cyclic structure). Cycle-freedom of the "uses" graph is not a precondition of system correctness; we can scrutinize each cycle on a case-by-case basis to understand and explain correctness, rather than removing cycles by rethinking system structure or by duplicating certain code (e.g., by "sandwiching"). Also, explicit function calls may not represent all correctness dependencies. Implicit correctness dependencies, which include shared memory and sharing through globals and timing dependencies, may or may not be problematical.

B.3 Module Packaging

A module is separable from the whole and packageable. We distinguish between "module" and "package"; a module is a logical container of a system part, whereas a package is a physical container of a system part. If there is not a strong reason to the contrary, each module should have a separate package. The modules of a system should be manifest (i.e., obvious) from the packaging. For a software system, the module interface and module implementation should be in separate packages, or there should be a well-defined reason why not.

EXAMPLE: Packaging the Secure Xenix(TM) Kernel

To make the nature of each function in the Secure XENIX kernel more conspicuous, one can add the following adjectives before function names:

SYSTEM_CALL,

PUBLIC, and

PRIVATE.

Also, to make more explicit the modules of the Secure Xenix(TM) kernel, one can add a subsystem-identifying prefix to each module name, for example, fs_dir.c would indicate that the directory manager is part of the File Subsystem ("fs"). Another way to make more explicit the modules of a kernel is to represent each major module (or subsystem) as a subdirectory. For example, the modules of the Secure Xenix(TM) kernel can be packaged as subdirectories of the directory kernel/, as follows:

- kernel/conf Configuration Subsystem
- kernel/dd Device Drivers (part of I/O Subsystem)
- kernel/fp Floating Point Support
- kernel/file File Subsystem
- kernel/i Interfaces (.i files) to Kernel Modules
- kernel/init Initialization Subsystem
- kernel/io Low-Level I/O Support (part of I/O Subsystem)
- kernel/ipc IPC Subsystem
- kernel/memory Memory (Management) Subsystem
- kernel/misc Miscellaneous Functions Subsystem
- kernel/process Process Subsystem
- kernel/syscall System Call (Processing) Subsystem
- kernel/security Security Subsystem

Examples of Modularity and System Packaging Defects

The definition of a module of Section B.2.1 allows us to derive certain measures of modularity, or of modularity defects. Most modularity defects arise from unrestricted use of global variables. This makes both the understanding of system structure difficult [383] and module replacement dependent on other modules.

B.4 Visibility of System Structure Using Modules

System modular structure also becomes visible by examining (1) the design abstractions used within would-be modules, (2) the hiding of information (i.e., data) within the would-be modules, and (3) the use of the would-be modules within systems layers.

B.4.1 Design Abstractions within Modules

Data abstraction, together with the use of other design abstractions, such as functional and control abstractions, significantly enhances the ability to identify a system's modules and to structure a system into sets of (ordered) layers. As a result, the visibility of system properties and their formal analysis become possible.

For illustration, we differentiate sic forms of abstraction that are typically implemented by a system's modules:

- functional abstraction,
- data abstraction,

- control abstraction,
- synchronization abstraction,
- interface abstraction, and
- implementation abstraction.

A module implements a *functional abstraction* if the output of the module is a pure mapping of the input to the module. That is, the module maintains no state. The module always produces the same output, if given identical input. The primary secret of a functional abstraction is the algorithm used to compute the mapping.

A *data abstraction* is "a description of a set of objects that applies equally well to any one of them. Each object is an instance of the abstraction" (cf. Britton and Parnas [59]). A module implements a data abstraction if it hides properties of an internal data structure. The interface of a data abstraction module can export a transparent type or an opaque type. A transparent type allows visibility (reading) of its internal fields, whereas an opaque type does not. A transparent type is typically represented as a dereferenceable pointer to the object, whereas an opaque type is typically represented by a "handle" on the object, a nondereferenceable "pointer" like a row number in a private table or a "capability."

A module implements a *control abstraction* if it hides the order in which events occur. The primary secret of a control abstraction is the order in which events occur and the algorithms used to determine the order (e.g., scheduling algorithms).

A module implements a *synchronization abstraction* if it encapsulates all synchronization primitives necessary for its concurrent execution [164]. The primary role of the module as a synchronization abstraction is to hide the details of these primitives (e.g., mutual exclusion, conditional wait, signals) from the module user and to restrict the scope of correctness proofs to the module definition (to the largest possible extent).

A module implements an *interface abstraction* if, by [59], it "represents more than one interface; it consists of the assumptions that are included in all of the interfaces that it represents." An operating system may contain an X-interface table, where each row is an interface (e.g., pointers to public functions) to a type X interface. As examples, the operating system may have an I/O device-interface table, a communication protocol-interface table, and a filesystem-interface table.

A module includes an *implementation abstraction* if it represents the implementation of more than one module. For example, if an operating system contains a number of similarly structured tables with similar public functions, then it may be possible to represent all such tables with one implementation schema (or abstract program).

In [179], Janson defines the concept of "type extension" as a hierarchy of data abstractions. The idea is to build abstract data types atop one another by defining the operations of a higher-level type in terms of the operations of lower-level types.

B.4.2 Information Hiding as a Design Abstraction for Modules

Information hiding [59, 281] is a software decomposition criterion. Britton and Parnas in [59] give the following description of information hiding.

"According to this principle, system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change. Every data structure is private to one module; it may be directly accessed by one or more programs within the module but not by programs outside the module. Any other [external] program that requires information stored in a module's data structures must obtain it by calling module programs [public functions]." ...

"Three ways to describe a module structure based on information-hiding are (1) by the roles played by the individual modules in the overall system operation; (2) by the secrets associated with each module; and (3) by the facilities [public functions] provided by each module. ..."

"For some modules we find it useful to distinguish between a primary secret, which is hidden information that was specified to the software designer, and a secondary secret, which refers to implementation decisions made by the designer when implementing the module designed to hide the primary secret."

In general, each module should hide an independent system-design decision. If a table (with related data) is involved, for example, a table manager module mediates all access to that table and it hides the representation of that table. The module secrets are the facts about the module that are not included in its interface — that is, the assumptions that client programs are not allowed to make about the module. The correctness of other modules must not depend on these facts.

Note that a system entirely based on "information hiding" is always modular.

B.4.3 Layering as a Design Abstraction Using Modules

A layer is a module. We say that a system is layered if the "uses" graph of its leaf modules is a linear order (a reflexive, transitive, and asymmetric relation). (If the layer is actually a collection of modules, then the linear order is on the layer rather than on the individual modules, although a lattice ordering could be used instead.)

Pictorially, we represent a layered system with horizontal stripes or bands, with one stripe per layer. Also, we typically show only the transitive reduction (i.e., remove all transitively implied arcs) of the "uses" graph. "Traditionally, a layer is thought of as providing services to the layer above, or the user layer. The user has some mechanism for invoking the layer, such a procedure call. The layer performs the service for its user and then returns. In other words, service invocation occurs from the top down." [81] We define layering as a system structuring (organizing) principle in which system modules are partitioned into groups such that the "uses" graph of the system module groups (the layers) is a linear order (although it could also be viewed as a partial order, e.g., in the form of a lattice). Classical layering permits only downcalls, not upcalls. Some experience suggests that upcalls can be valuable (as long as security and integrity are not violated).

"In classical layering, a lower layer performs a service without much knowledge of the way in which that service is being used by the layers above. Excessive contamination of the lower layers with knowledge about upper layers is considered inappropriate, because it can create the upward dependency which layering is attempting to eliminate. ... It is our experience, both with Swift and in other upcall experiments that we have done, that the ability to upcall in order to ask advice permits a substantial simplification in the internal algorithms implemented by each layer." [81]

We understand the phrase layers of abstraction as just type extension where the hierarchy is a layering.

Two popular types of layering in an operating system are policy/mechanism layering and hardware-independent / hardware-dependent (HI/HD) layering – also called device-independent / device-dependent (DI/DD). This was a fundamental part of the Multics input-output system architecture.

Also, see [358]. The idea of policy/mechanism layering is that design decisions about system policies (or mechanisms) should tend to reside in higher (or lower) layers. The rationale for HI/HD layering, with the HI layer above the HD layer, is to localize machine dependent modules to simplify porting. In practice, these two layering criteria may not be compatible, since "some design decisions which one would be tempted to label as policies rather than mechanisms can be machine dependent" [358].

B.5 Measures of Modularity and Module Packaging

The goal of identifying simple and practical tools for establishing and verifying modularity properties requires that we define *simple and practical* measures for modularity. Below we define four classes of such measures, namely, for (1) replacement dependence, (2) global variables, (3) module reusability, and (4) component packaging. While we believe that these classes are important for modularity assessments, the specific examples of measures are provided only for illustrative purposes. Other specific measures may be equally acceptable in each class.

B.5.1 Replacement Dependence Measures

One way to define a modularity defect is by replacement dependence, a violation of our property P5 of a module. We define two replacement dependence measures below.

Measure M1. We define modularity measure M1 on an "almost-module" *m* (satisfying properties P1-P4 but not P5 of the module definition of Section B.21.1) as the number of files that must be edited to replace the implementation of module *m*, less one (the minimum).

Measure M2. We define modularity measure M2 on an "almost-module" *m* as the number of lines of source code, not in the "primary" implementation file, that must be edited to replace the implementation of module *m*.

B.5.2 Global Variable Measures

Although global variables can be useful for system performance, they should be avoided whenever they can produce replacement dependence and extra-module correctness dependencies not represented by explicit service dependencies. It is tempting, if inconsequential, to argue that correctness should be determined only from explicit service dependencies and thus from module interfaces, and not from data dependencies, and to conclude that all globals should be eliminated.

We have defined two software modules as data dependent if they share a common global variable. A data dependency can sometimes be harmless, or safe or easy to understand, and sometimes it is harmful, or unsafe or difficult to understand [383]. We can define a hierarchy of module dependency (coupling) from very safe to very unsafe with the following types of variables:

(a) local to a function,
(b) formal of a function,
(c) global (but private) to one module,
(d) global with one writer module (and many reader modules),
(e) global with a few writer modules (and many reader modules) and a well-defined use discipline,

(f) global with many writer modules (and many reader modules) and a well-defined use discipline, and

(g) global with many writer modules (and many reader modules) and an ill-defined (or undefined) use discipline.

Both types of global variables (a) and (b) are safe, while a global variable of type (g) is unsafe. In general, (a) is safer than (b), which in turn is safer than (c), and so on, and (g) is the most unsafe. In general, module independence is valuable because one can understand (thus, replace, fix, evolve) the module by understanding only its interface and what it uses. On the other hand, a module dependency is undesirable when one cannot understand (thus, replace, fix, evolve) a module without understanding the implementations of other modules. In this sense, a module including global variables of type (d) is easier to understand than a module including a global variable of type (e); a module including global variables of type (e) is easier to understand than one including a global variable of type (f); a module including global variables of type (g) is virtually impossible to understand. By "use discipline" we mean "correctness rule."

If a global variable can and should be converted to either a local of one module, or a formal of one or more public functions, or a local of a public function, then this new scope is generally better than its old scope as a global. In general, use of formals is a better programming discipline than use of informals (globals, environment variables). As one reason, it makes the function parameters more explicit; this makes the functions simpler to understand, and simpler to evolve (e.g., as a remote procedure call). As another reason, for recursion, use of formals is a less error prone programming discipline than use of informals; care must be taken to save current informal parameters before a recursive call and to restore them after the call.

Measure M3. We define measure M3 on an "almost-module" *m* as the number of globals that it writes that are also written by other modules or almost-modules.

B.5.3 Module Reusability Measures

Another way to define a modularity defect is as a reuse impediment, or a violation of our property P6 of a module. The major technical source of module-reuse impediments is the violation of *compatibility* [134] between a module's interface and its new environment of use. Whenever this impediment materializes, we say that the module cannot be *composed* within its environment of use. In particular, we are interested in the amount of extra (correct) code that has to be written and the amount of administrative effort that has to be expanded to remove interface incompatibility. We define four measures of reuse impediments below. These measures can also be viewed as simple estimates of ability to compose modules.

For all the measures below we assume the module being reused satisfies properties P1-P5 but not P6 of the module definition in Section B.2.1.1.

Measure M4. We define modularity measure M4 on an "almost-reusable" module *m* as the number of exception handlers that must be written to cover all the exceptions signaled by the reused module.

Measure M5. We define modularity measure M5 on an "almost-reusable" module *m* as the number of resource allocation and deallocation instructions that must be written to enable the execution of the reused module, less two (the minimum).

Measure M6. We define modularity measure M6 on an "almost-reusable" module *m* as the number of lines of code that must be written to satisfy the module invocation discipline (i.e., type matching and coercion, if necessary, setting constants and global variables) to enable the execution of the reused module, less than the number of formal parameters (the minimum).

Measure M7. We define modularity measure M6 on an "almost-reusable" module *m* as the number of permissions/system privileges that must be granted to enable the execution of the reused module, less one (the minimum required to invoke the module).

Clearly, zero extra code or administrative effort is best for all measures, but a small number (e.g., one to three) of extra programs and administrative effort is acceptable, provided that this is a one-time requirement.

B.5.4 Component-Packaging Measures

Component-packaging defects can also be evaluated using measures of how closely the module package reflects the module definition. Since packaging is not a modularity property by our definition, packaging-defect measures are not necessarily modularity measures. However, packaging-defect measures are useful in evaluating visibility of system structures by standard tools (e.g., browsers). The examples of packaging-defect measures provided below are only for illustrative purposes. Other measures may be equally acceptable.

Measure M8: Assume that a leaf module has one implementation file and one interface file. We define packaging measure M8 to be the number of files (e.g., .c, assuming C) over which the implementation of a leaf module is spread, plus the number of files over which the interface of a leaf module is spread, then subtracting two (the minimum).

M8 is a measure of packaging defects with granularity of a per-file count. For a module *m* with exactly one implementation file and exactly one interface file, M8(m) = 0. In general, for system *s*, and measure M, M(s) is the summation of M(m) for each leaf module *m*.

Measure M9: We define packaging measure M9 as similar to Measure M8 except that we count the number of lines of source code (SLOC) not in "the primary" implementation file plus not in "the primary" interface file.

As a class of approximate measures of modularity, additional packaging defect measures can be defined.

B.6 Cost Estimates for Modular Design

Different system designers may argue over the specific costs of different features of modular design, but few would disagree with the following "less expensive than" (<) and "included in" (<<) orders:

"Just Code It," < Module with Properties P1-P4 < Replacement Independence (P5) < Reusability (P6); and

"Just Code It," << Module with Properties P1-P4 << Replacement Independence (P5) << Reusability (P6).

In fact, the first seven modularity measures presented above illustrate the different types of complexity involved in modular design and provide intuition for the above cost and feature-inclusion ordering. Specific cost figures for modularity are hard to come by, as insufficient experience is available with systems designs where modularity is an important requirement. However, based on (1) experience with Secure Xenix(tm) [115] (aka Trusted Xenix(tm)), the only Unix(tm) system to achieve a security level where modularity was required (i.e., TCSEC level B2), and (2) experience reported by Lampson [199], we can estimate the relative costs using a "Just Code It" (JCI) unit of modularity currency.

Lampson estimates the costs for a "good module for your system" and "reusable component." We approximate a "good module for your system" with modularity properties P1–P4 and "reusable component" with property P6. Using these estimates and our approximation, we obtain the following cost ranges:

Cost of Module with Properties P1–P4 (i.e., "good module for your system") = 0.5 JCI - 2 JCI, depending on how many of the properties P1 – P4 are desired (or "how lucky you are" [199]) and

Cost of Module Reusability (P6) = 3 JCI - 5 JCI.

For the purposes of this study, it seems important to be able to estimate the cost of the module "replacement independence" property (P5). The cost ordering presented above allows us to interpolate the cost estimate for this property. Since

Module with Properties P1-P4 < Replacement Independence (P5) < Reusability (P6),

the above estimates suggest that

Cost of a Module's Replacement Independence (P5) = 2 JCI - 3 JCI.

However, is our approximation of a "good module for your system" with module properties P1 – P4 valid? We attempt to validate this approximation using cost estimates of Secure Xenix(tm), where modularity properties P1-P4 were satisfied. The Secure Xenix(tm) estimates can be split roughly as follows: $1/3$ design cost (including modularity properties P1 - P4, but excluding that of testing and documentation required for P6), $1/3$ assurance cost (including testing to the level required for P6), and $1/3$ documentation (including everything that a user and an evaluator might want for level B2, and hence including documentation required for P6). These estimates suggest the following approximate relationships:

Cost of Module with Properties P1-P4 = 0.33 of Cost of Module with Properties P1-P6.

Hence, using the above cost estimates, we obtain:

Cost of a Module with Properties P1 - P4 = 1 JCI - 1.67 JCI,

which is consistent with Lampson's estimate that

"good module for your system" = 0.5 JCI - 2 JCI.

We stress that the above estimates of modularity costs are *very* rough. However, they appear to be consistent with each other for modularity properties P1 - P4. Further, the requirements of property P6 seem to be consistent with modularity requirements of TCSEC level B2-B3. Hence these cost estimates could be further validated using examples of systems rated at those levels.

B.7 Tools for Modular Decomposition and Evaluation

A variety of *tools* (e.g., algorithms, methods, programs) for modular system decomposition have been proposed during the past dozen years. The motivation for the development of these tools has been driven primarily by the need to improve the understandability and maintainability of legacy software, and to a lesser extent to enable module reusability. Few of these tools were motivated directly by concerns of module replacement independence and correctness, and consequently few support formal dependency analyses among identified modules.

In general, most tools for modularization can be divided into two broad classes, namely those based on (1) *clustering* functions and data structures based on different modularity criteria, and (2) *concept analysis* – an application of a lattice-theoretical notion to groups of functions and function attributes (e.g., access to global variables, formal parameters and types returned). The primary difference between the two classes is that approaches used by the former use metrics of function cohesion and coupling directly whereas those used by the latter rely mostly on semantic grouping of functions that share a set of attributes. Both approaches have advantages and disadvantages. For example, although clustering is based on well-defined metrics (which overlap but are not necessarily identical with M1-M7 above) and always produces modular structures, it does not necessarily provide a semantic characterization of the modules produced and often reveals only few module-design characteristics. In contrast, concept analysis helps characterize the modules recovered from source code semantically, but does not always lead to easily identifiable modules (e.g., nonoverlapping and groupings of program entities covering all functions of a program). Neither approach is designed to characterize correctness dependencies among modules (e.g., systems analysis) and neither is intended to address the properties of systems obtained by modular composition (e.g., systems synthesis). We note that metrics M1-M7 suggested above could be applied equally well to the modular structures produced by either approach. In the balance of this appendix, we provide representative examples of tools developed for modularity analysis using each approach.

B.7.1 Modularity Analysis Tools Based on Clustering

The first class of clustering tools is that developed for the identification of *abstract types and objects* and their modules in source code written in a non-object-oriented language, such as C. The modular structure produced by these tools is partial as it represents only the use of data abstractions in source code. Other modularity structures manifest in the use of other abstractions (e.g., functional, control, synchronization) are not addressed. All tools of this class recognize abstract-object instances by clustering functions that access function-global variables, formal parameters, returned [211] or received [65]. Relations among the identified components (e.g., call and type relations [65], relations of procedures to internal fields of structures [384]) are used to build dominance trees or graphs, and dominance analysis is used to hierarchically organize the abstract type/object modules into subsystems [132]. A typical problem that appears with this class of tools is that of "coincidental links," which is caused by procedures that implement more than one function, and "spurious links," which is caused by functions that access data structures implementing more than one object type. Both types of links lead to larger-than-desirable clusterings of functions.

A second class of clustering tools is that based on measuring high internal cohesion and low external coupling (e.g., few inter-module global variables and functions). Tools of this class define cohesion principles and derive metrics based on those principles. For example, the measure of "similarity" among procedures defined for the ARCH tool [346] is derived from the "information hiding" principle and used to extract modules with internal cohesion and low external coupling. Further, genetic algorithms have been used by other tools (e.g., BUNCH [217]) to produce hierarchical clustering of modules identified using cohesion and coupling metrics [216]. Clustering techniques have also been used to group module interconnections into "tube edges" between multiple modules that form subsystems [215].

B.7.2 Modularity Analysis Tools based on Concept Analysis

The notion of concept analysis is defined in lattice theory as follows. Let X be a set of objects, or an *extent*, and Y a set of attributes, or an *intent*, and R a binary relation between objects and attributes. A *concept* is the maximal collection of objects sharing common attributes. Formally, a concept is the pair of sets (X, Y) such that $X = \tau(Y)$ and $Y = \sigma(X)$, where σ and τ are *anti-monotone* and *extensive* mappings in R (i.e., they form a *Galois connection*). A mapping, say, σ is said to be anti-monotone if $X_1 \subseteq X_2 \Rightarrow \sigma(X_2) \subseteq \sigma(X_1)$. Mappings σ, τ are said to be extensive if $X \subseteq \tau(\sigma(X))$ and $Y \subseteq \sigma(\tau(Y))$. Further, a concept (X_0, Y_0) is a *subconcept* of concept (X_1, Y_1), if $X_0 \subseteq X_1$ or, equivalently, $Y_1 \subseteq Y_0$. The subconcept relation forms a partial order over the set of concepts leading to the notion of *concept (Galois) lattice*, once an appropriate top and bottom are defined. A fundamental theorem of concept lattices relates subconcepts and superconcepts and allows the least common superconcept of a set of concepts to be computed by intersecting the intents and finding the common objects of the resulting intersection.

Concept lattices have been used in tools that identify conflicts in software-configuration information (e.g., as in the RECS tool [356]). During the past half a dozen years, they have been used to analyze modularity structures of programs written in programming languages that do not support a syntactic notion of a module (e.g., Fortran, Cobol, C) [206, 353]. More recently, they have also been used to identify hierarchical relationships among classes in object-oriented programming [136, 137, 357].

The use of concept analysis in identifying modular structures of source code requires (1) the selection of an object set (i.e., function set) and attribute set (i.e., function characteristics such as a function's access to global variables, formal parameters returned, types, and so on); (2) the construction of the concept lattice, and (3) the definition of concept partitions and subpartitions as nonoverlapping groupings of program entities (that is, functions and attributes). Note that if concept partitions are used to represent modules, they must be complete; that is, they must cover the entire object (i.e., function) set [353]. The notion of subpartitions was introduced to remove the completeness restriction that sometimes leads to module overlaps caused by artificial module enlargements (to cover the function set) [366].

Concept analysis applied to modularity analysis has the advantage of flexible determination of modules. That is, if the proposed modularization is too fine-grained, moving up the (sub)partition lattice allows a coarser granularity of modules to be found. Conversely, if the proposed modularization is too coarse, additional attributes can be added to identify finer-granularity modules. Similar flexibility in clustering can be achieved in a more complex manner, namely, by adding and removing metrics to the set used by clustering analysis. Concept analysis also has the advantage that the resulting modularity has a fairly precise semantic characterization.

B.8 Virgil Gligor's Acknowledgments

Much of my understanding of "modularity" is based on joint work with Matthew S. Hecht on defining practical requirements for modular structuring of Trusted Computing Bases in the late

1980s. Figures 1-5 were generated by the Secure Xenix modularity study led by Matthew. Virgil D. Gligor

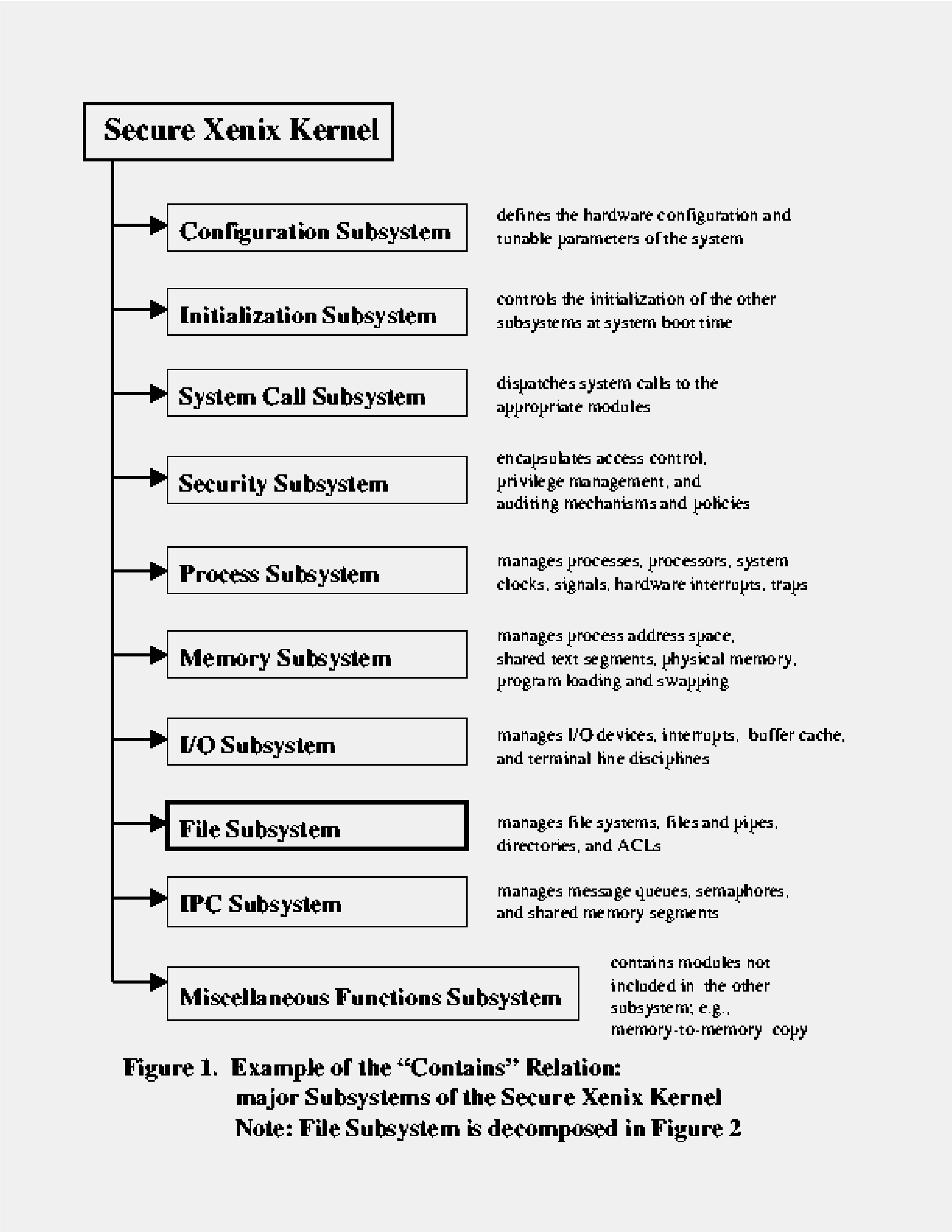

Figure B.1: Example of the *Contains* Relation

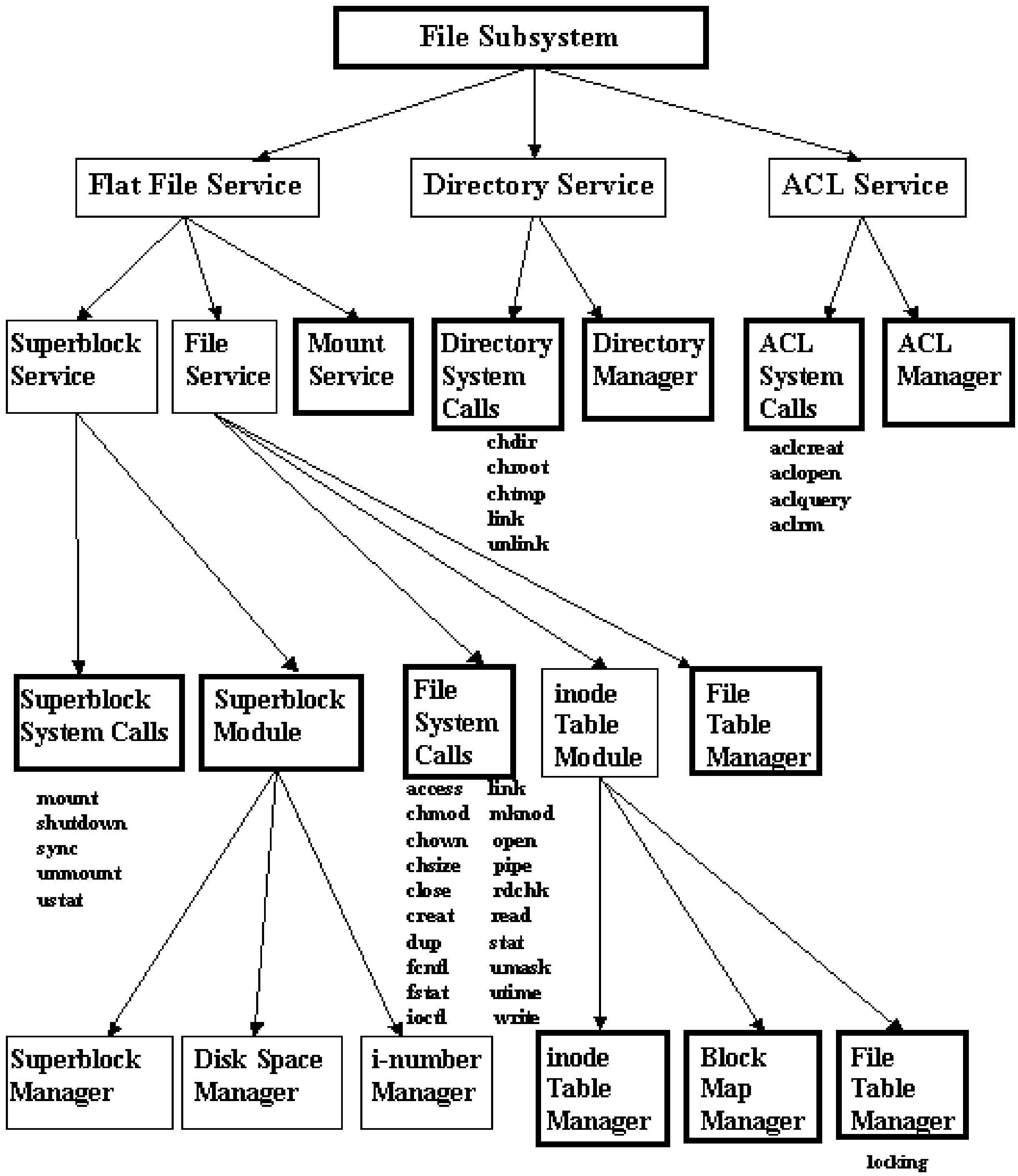

Figure 2. Example of the "Contains" Relation and Module Hierarchy: decomposition of File Subsystem
Note: a darkened box shows a file of source code

Figure B.2: Example of the *Contains* Relation and Module Hierarchy

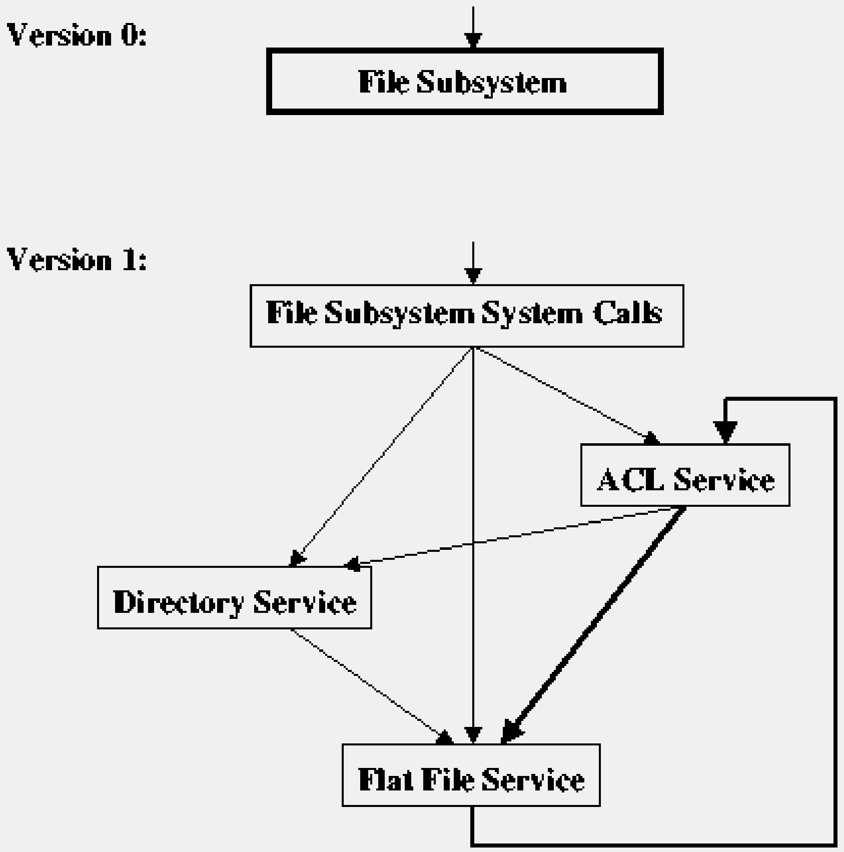

Figure 3. Example of Refining the "Uses" Relation (Versions 0,1) within the File Subsystem of Secure Xenix Kernel
Note: arrows denote the uses relationships.

Figure B.3: Example of Refining the *Uses* Relation 1

Version 2:

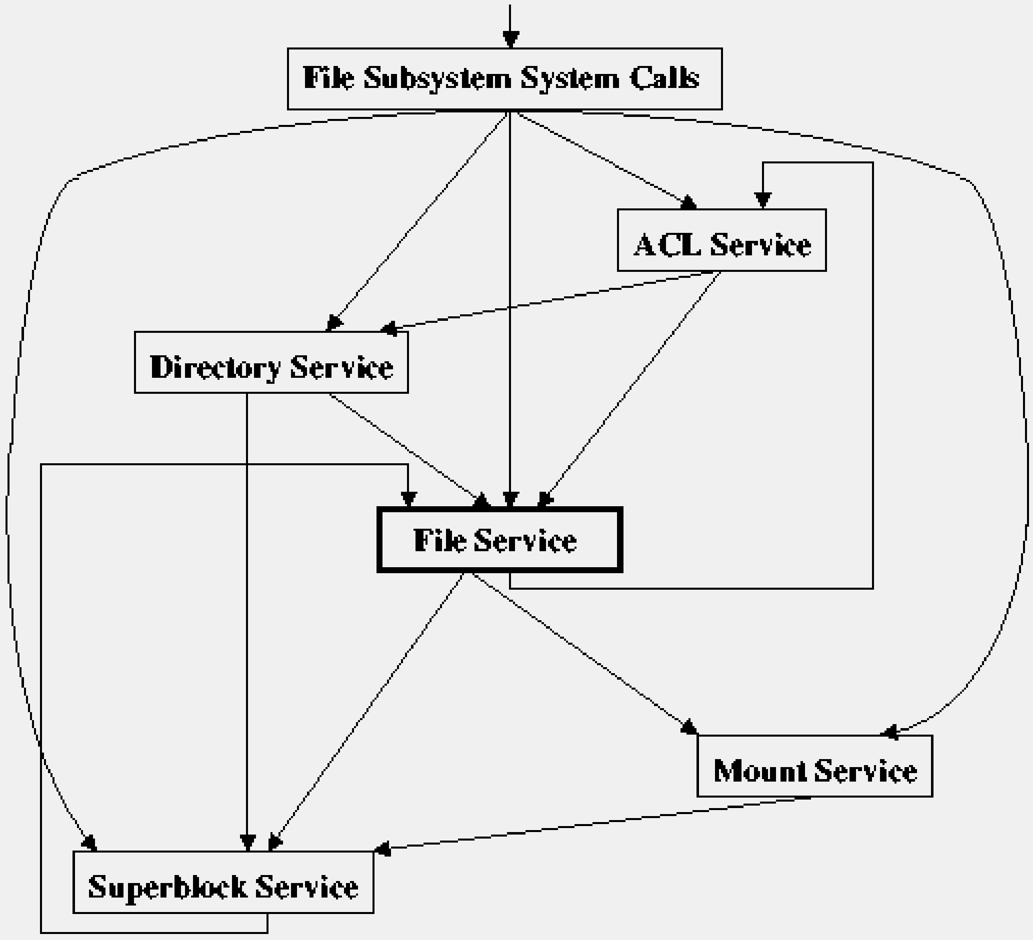

Figure 4. Example of Refining the "Uses" Relation (Version 2) within the File Subsystem of Secure Xenix Kernel Note: arrows denote the "uses" relationships.

Figure B.4: Example of Refining the *Uses* Relation 2

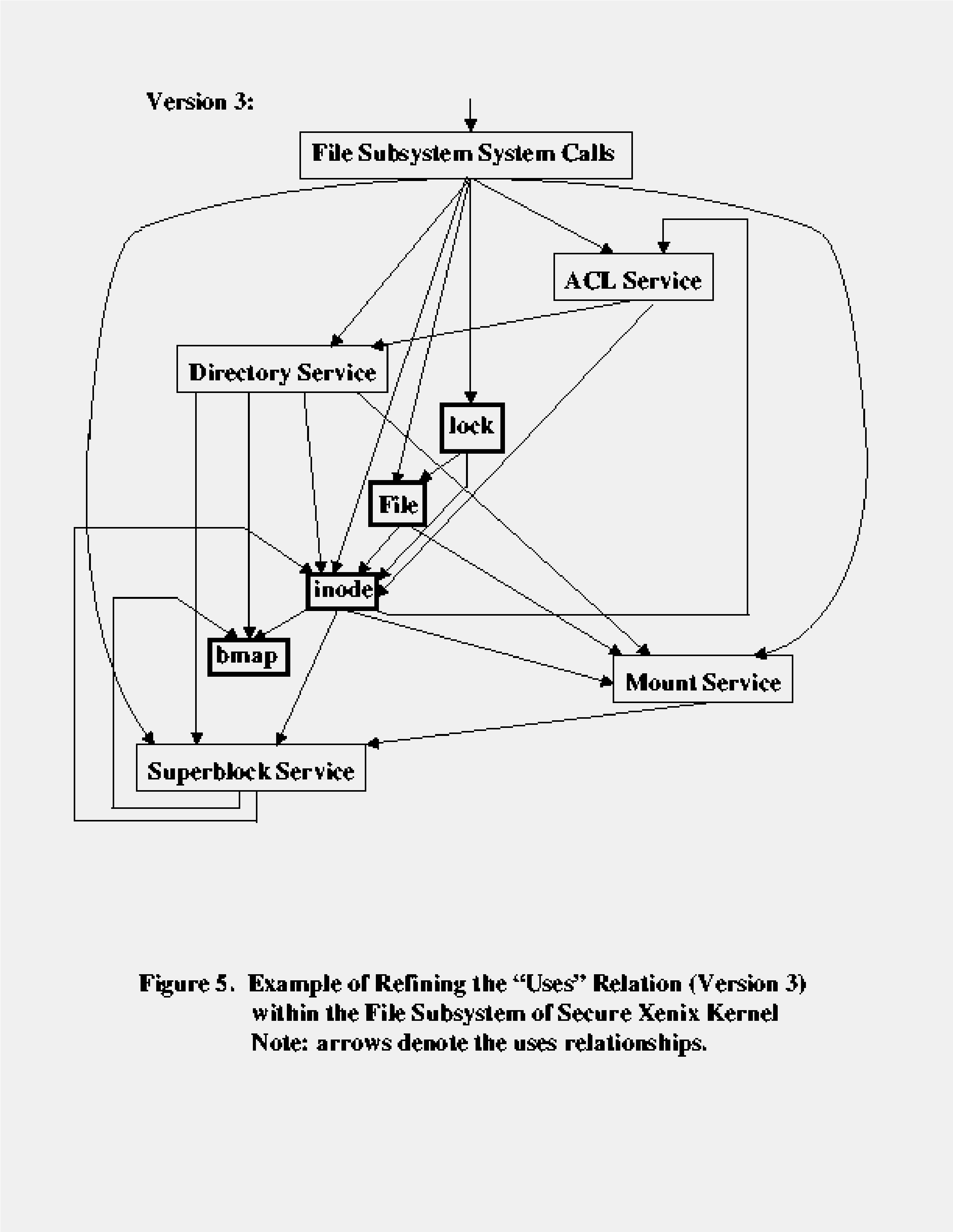

Figure B.5: Example of Refining the *Uses* Relation 3

Bibliography

[1] M. Abadi, A. Banerjee, N. Heintze, and J.G. Riecke. A core calculus of dependency. In *POPL '99, Proceedings of the 26th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 20-22 1999.

[2] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The Spi calculus. Technical report, Digital Equipment Corporation, SRC Research Report 149, Palo Alto, California, January 1998.

[3] 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, Berlin, Lecture Notes in Computer Science, vol. 230.

[4] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. Technical report, Digital Equipment Corporation, SRC Research Report, Palo Alto, California, June 1994.

[5] R.P. Abbott et al. Security analysis and enhancements of computer operating systems. Technical report, National Bureau of Standards, 1974. Order No. S-413558-74.

[6] H. Abelson, R. Anderson, S.M. Bellovin, J. Benaloh, M. Blaze, W. Diffie, J. Gilmore, P.G. Neumann, R.L. Rivest, J.I. Schiller, and B. Schneier. The risks of key recovery, key escrow, and trusted third-party encryption. *(http://www.cdt.org/crypto/risks98/)*, June 1998. This is a reissue of the May 27, 1997 report, with a new preface evaluating what happened in the intervening year.

[7] M.D. Abrams and M.V. Joyce. Composition of trusted IT systems. Technical report, MITRE, September 1992. Draft.

[8] N. Abramson and F.F. Kuo (editors.). *Computer-Communication Networks*. Prentice-Hall, 1971.

[9] J. Adamek. *Foundations of Coding: Theory and Applications of Error-Correcting Codes with an Introduction to Cryptography and Information Theory*. Wiley-Interscience, 1991.

[10] M. Adler. Tradeoffs in probabilistic packet marking for ip traceback. In *Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing*, pages 407–418, 2002.

[11] P.E. Agre and M. Rotenberg, editors. *Technology and Privacy: The New Landscape*. MIT Press, Cambridge, Massachusetts, 1997.

[12] J.H. An, Y. Dodis, and T. Rabin. On the security of joint signature and encryption. In *Advances in Cryptology, EUROCRYPT 2002, Amsterdam, The Netherlands, Springer-Verlag, Berlin, Lecture Notes in Computer Science*, pages 83–107, May 2002.

[13] R. Anderson and M. Kuhn. Tamper resistance — a cautionary note. In *Proceedings of the Second Usenix Workshop on Electronic Commerce*, pages 1–11. USENIX, November 1996.

[14] R.J. Anderson. *Security Engineering: A guide to Building Dependable Distributed Systems*. John Wiley and Sons, New York, 2001.

[15] T. Anderson and J.C. Knight. A framework for software fault tolerance in real-time systems. *IEEE Transactions on Software Engineering*, SE-9(3):355–364, May 1983.

[16] T. Anderson and P.A. Lee. *Fault-Tolerance: Principles and Practice*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1981.

[17] A.W. Appel and D.B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science vol. 528*, pages 1–26, Berlin, 1991. Springer-Verlag.

[18] W.A. Arbaugh, D.J. Farber, and J.M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 Symposium on Security and Privacy*, pages 65–71, Oakland, California, May 1997. IEEE Computer Society.

[19] W.A. Arbaugh, A.D. Keromytis, D.J. Farber, and J.M. Smith. Automated recovery in a secure bootstrap process. In *Proceedings of the 1998 Network and Distributed System Security Symposium*, San Diego, California, March 1998. Internet Society.

[20] K. Ashcraft and D. Engler. Detecting lots of security holes using system-specific static analysis. In *Proceedings of the 2002 Symposium on Security and Privacy*, pages 143–159, Oakland, California, May 2002. IEEE Computer Society.

[21] D. Asonov and R. Agrawal. Keyboard acoustic emanations. In *Proceedings of the 2004 Symposium on Security and Privacy*, pages 3–11, Oakland, California, May 2003. IEEE Computer Society.

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

[23] Numerous authors. Automated software engineering (special section). *ERCIM News*, (58):12–51, July 2004.

[24] A. Avižienis and J-C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.

[25] A. Avižienis and J. C. Laprie, editors. *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, Santa Barbara, California, August 1989. Springer-Verlag, Vienna, Austria.

[26] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January-March 2004.

[27] M. Backes and B. Pfitzmann. A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. In *23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Mumbai, India, December 2003.

[28] M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library with nested operations. In *Tenth ACM Conference on Computer and Communications Security*, Washington, D.C., October 2003. ACM.

[29] P. Baran. Reliable digital communications systems using unreliable network repeater nodes. Technical Report P-1995, The RAND Corporaton, May 27 1960.

[30] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, Reading, Massachusetts, 2003. reviewed in RISKS-23.01.

[31] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, Massachusetts, 1998.

[32] L. Bauer, A.W. Appel, and E.W. Felten. Mechanisms for secure modular programming in Java. *Software–Practice and Experience*, 33:461–480, 2003.

[33] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, 1999. (http://www.extremeprogramming.org).

[34] R. Bejtlich. *The Tao of Network Security Monitoring*. Addison-Wesley, Reading, Massachusetts, 2004.

[35] 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, Massachusetts, March 1976.

[36] L.A. Benzinger, G.W. Dinolt, and M.G. Yatabe. Combining components and policies. In *Proceedings of the Computer Security Foundations Workshop VII, J. Guttman, editor*, June 1994.

[37] L.A. Benzinger, G.W. Dinolt, and M.G. Yatabe. Final report: A distributed system multiple security policy model. Technical report, Loral Western Development Laboratories, report WDL-TR00777, San Jose, California, October 1994.

[38] P.L. Bernstein. *Against the Gods: The Remarkable Story of Risk.* John Wiley & Sons, New York, 1996.

[39] 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.

[40] 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, California, April 1983. IEEE Computer Society. (Abstract only).

[41] W.R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5(4):519–30, December 1989.

[42] W.R. Bevier, W.A. Hunt, Jr., J S. Moore, and W.D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

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

[44] R. Bisbey II, J. Carlstedt, and D. Chase. Data dependency analysis. Technical Report ISI/SR-76-45, USC Information Sciences Institute (ISI), Marina Del Rey, California, February 1976.

[45] R. Bisbey II and D. Hollingworth. Protection analysis: Project final report. Technical report, USC Information Sciences Institute (ISI), Marina Del Rey, California, 1978.

[46] R. Bisbey II, G. Popek, and J. Carlstedt. Protection errors in operating systems: Inconsistency of a single data value over time. Technical Report ISI/SR-75-4, USC Information Sciences Institute (ISI), Marina Del Rey, California, December 1975.

[47] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, Reading, Massachusetts, 2002.

[48] M. Bishop. *Introduction to Computer Security*. Addison-Wesley, Reading, Massachusetts, 2004.

[49] B. Blanc. GATeL: Automatic test generation from Lustre descriptions. *ERCIM News*, (58):29–30, July 2004.

[50] M. Blume and A.W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847, 1999.

[51] 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. Springer-Verlag, Berlin, LCNS 839, 1994.

[52] Defense Science Board. Protecting the homeland, volume ii. Technical report, Defense Science Board Task Force on Defensive Information Operations 2000 Summer Study, March 2001.

[53] 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.

[54] D. Boneh, R.A. DeMillo, and R.J. Lipton. On the importance of checking cryptographic protocols for faults. *Journal of Cryptology*, 14(2):101–119, 1997.

[55] P. Boudra, Jr. Minutes of the meetings of the system composition working group, volume 1. Technical report, National Security Agency, Information Systems Security Organization, Office of Infosec Systems Engineering, S9 Technical Report 6-92, Library No. S-239, 646, October 1992. For Official Use Only.

[56] P. Boudra, Jr. Report on rules of system composition: Principles of secure system design. Technical report, National Security Agency, Information Systems Security Organization, Office of Infosec Systems Engineering, I9 Technical Report 1-93, Library No. S-240, 330, March 1993. For Official Use Only.

[57] R.S. Boyer, B. Elspas, and K.N. Levitt. SELECT: A formal system for testing and debugging programs by symbolic execution. In *Proc. Int. Conf. Reliable Software*, pages 234–244. IEEE, IEEE, April 1975.

[58] R.S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[59] K.H. Britton and D.L. Parnas. A-7e software module guide. Technical report, NRL Memorandum Report 4702, Naval Research Laboratory, Washington, D.C., December 1981.

[60] J.E. Brunelle and D.E. Eckhardt, Jr. Fault-tolerant software: An experiment with the SIFT operating system. In *Proceedings of the Fifth AIAA Computers in Aerospace Conference*, pages 355–360, October 1985.

[61] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[62] R.W. Butler. An elementary tutorial on formal specification and verification using PVS. Technical report, NASA Langley Research Center, Hampton, Virginia, June 1993.

[63] R.W. Butler, D.L. Palumbo, and S.C. Johnson. Application of a clock synchronization validation methodology to the SIFT computer system. In *Digest of Papers, FTCS 15*, pages 194–199, Ann Arbor, Michigan, June 1985. IEEE Computer Society.

[64] Canadian Systems Security Centre, Communications Security Establishment, Government of Canada. *Canadian Trusted Computer Product Evaluation Criteria*, December 1990. Final Draft, version 2.0.

[65] G. Canfora, A. Cimitile, M. Munro, and C. Taylor. Extracting abstract data types from C programs: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 200–209, September 1993.

[66] 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, California, April 1986. IEEE Computer Society.

[67] J. Carlstedt. Protection errors in operating systems: Validation of critical conditions. Technical Report ISI/SR-76-5, USC Information Sciences Institute (ISI), Marina Del Rey, California, May 1976.

[68] J. Carlstedt, R. Bisbey II, and G. Popek. Pattern-directed protection evaluation. Technical Report ISI/SR-75-31, USC Information Sciences Institute (ISI), Marina Del Rey, California, June 1975.

[69] A. Chander, D. Dean, and J.C. Mitchell. A state-transition model of trust management. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 27–43, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society Technical Committee on Security and Privacy.

[70] A. Chander, D. Dean, and J.C. Mitchell. Deconstructing trust management. In *Proceedings of the 2002 Workshop on Issues in the Theory of Security*, Portland, Oregon, January 2002. IFIP Working Group 1.7.

[71] A. Chander, D. Dean, and J.C. Mitchell. A distributed high assurance reference monitor. In *Proceedings of the Seventh Information Security Conference Lecture Notes in Computer Science vol. 3225*, pages 231–244, Berlin, September 2004. Springer-Verlag.

[72] A. Chander, D. Dean, and J.C. Mitchell. Reconstructing trust management. *Journal of Computer Security*, 12(1):131–164, January 2004.

[73] D. Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security and Privacy*, 2(1):38–47, January-February 2004.

[74] H. Chen. *Lightweight Model Checking for Improving Software Security*. PhD thesis, University of California, Berkeley, 2004. http://www.cs.ucdavis.edu/˜hchen/paper/phddis.ps.

[75] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of code. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 171–185, San Diego, California, February 2004. Internet Society.

[76] H. Chen and J. Shapiro. Using build-integrated static checking to preserve correctness invariants. In *Proceedings of the Eleventh ACM Conference on Computer and Communications Security (CCS)*, Washington, D.C., November 2004.

[77] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Ninth ACM Conference on Computer and Communications Security*, Washington, D.C., November 2002. ACM.

[78] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security 2002*, pages 171–190, San Francisco, California, August 2002. USENIX.

[79] B.V. Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 Symposium on Security and Privacy*, pages 160–173, Oakland, California, May 2002. IEEE Computer Society.

[80] W.R. Cheswick, S.M. Bellovin, and A.D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker, Second Edition*. Addison-Wesley, Reading, Massachusetts, 2003.

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

[82] 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, California, April 1987. IEEE Computer Society.

[83] D.D. Clark et al. *Computers at Risk: Safe Computing in the Information Age*. National Research Council, National Academy Press, 2101 Constitution Ave., Washington, D.C., 5 December 1990. Final report of the System Security Study Committee.

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

[85] F.J. Corbató, J. Saltzer, and C.T. Clingen. Multics: The first seven years. In *Proceedings of the Spring Joint Computer Conference*, volume 40, Montvale, New Jersey, 1972. AFIPS Press.

[86] 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.

[87] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[88] I. Crnkovic and M. Larsson. Classification of quality attributes for predictability in component-based systems. In *Workshop on Architecting Dependable Systems (DSN WADS 2004)*, Florence, Italy, June 2004. http://www.cs.kent.ac.uk/events/conf/2004/wads/DSN-WADS2004/indexProgDSN2004.html.

[89] M. Curtin. *Developing Trust: Online Security and Privacy*. Apress, Berkeley, California, and Springer-Verlag, Berlin, 2002.

[90] M. Cusumano, A. MacCormack, C.F. Kemerer, and W. Crandall. A global survey of software development practices. Technical report, MIT Sloan School of Management, Cambridge, Massachusetts, June 2003.

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

[92] 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.

[93] A. Datta, R. Küsters, J.C. Mitchell, A. Ramanathan, and V. Shmatikov. Unifying equivalence-based definitions of protocol security. In *Proceedings of the ACM SIGPLAN and IFIP WG 1.7 Fourth Workshop on Issues in the Theory of Security*, Oakland, California, April 2004. IEEE Computer Society.

[94] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, New York, NY, 2001. Cambridge Tracts in Theoretical Computer Science no. 54.

[95] D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Computer Science Department, Princeton University, January 1999. (http://www.cs.princeton.edu/sip/pub/ddean-dissertation.php3).

[96] D. Dean. The impact of programming language theory on computer security. In *Proceedings of the Mathematical Foundations of Programming Semantics (MFPS)*, New Orleans, Louisiana, March 2002. Slides at http://www.csl.sri.com/neumann/ddean-MFPS02.ppt.

[97] D. Dean, M. Franklin, and A. Stubblefield. An algebraic approach to ip traceback. *ACM Transactions on Information and System Security*, 5(2):119–137, May 2002.

[98] D. Dean and D. Wagner. Intrusion detection via static analysis. In *Proceedings of the 2001 Symposium on Security and Privacy*, Oakland, California, May 2001. IEEE Computer Society.

[99] G. Denker and J. Millen. CAPSL integrated protocol environment. In *DARPA Information Survivability Conference (DISCEX 2000)*, pages 207–221. IEEE Computer Society, 2000.

[100] D.E. Denning, S.G. Akl, M. Heckman, T.F. Lunt, M. Morgenstern, P.G. Neumann, and R.R. Schell. Views for multilevel database security. *IEEE Transactions on Software Engineering*, 13(2), February 1987.

[101] D.E. Denning, P.G. Neumann, and Donn B. Parker. Social aspects of computer security. In *Proceedings of the 10th National Computer Security Conference*, September 1987.

[102] Y. Desmedt, Y. Frankel, and M. Yung. Multi-receiver/multi-sender network security: Efficient authenticated multicast/feedback. In *Proceedings of IEEE INFOCOM*. IEEE, 1992.

[103] Y. Deswarte, L. Blain, and J.-C. Fabre. Intrusion tolerance in distributed computing systems. In *Proceedings of the 1991 Symposium on Research in Security and Privacy*, pages 110–121, Oakland, California, April 1991. IEEE Computer Society.

[104] W. Diffie and S. Landau. *Privacy on the Line: The Politics of Wiretapping and Encryption*. MIT Press, 1998.

[105] E.W. Dijkstra. Co-operating sequential processes. In *Programming Languages, F. Genuys (editor)*, pages 43–112. Academic Press, 1968.

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

[107] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[108] G.W. Dinolt and J.C. Williams. A Graph-Theoretic Formulation of Multilevel Secure Distributed Systems: An overview. In *1987 IEEE Symposium on Security and Privacy*, pages 99–103, 1730 Massachusetts Avenue, N.W., Washington, D.C. 20036-1903, April 1987. The Computer Society of the IEEE, IEEE Computer Society Press.

[109] B.L. DiVito and L.W. Roberts. Using formal methods to assist in the requirements analysis of the space shuttle GPS change request. Technical Report NASA Contractor Report 4652, NASA Langley Research Center, Hampton, Virginia, August 1996.

[110] S. Dolev. *Self-Stabilization*. MIT Press, Cambridge, Massachusetts, 2000.

[111] B. Dutertre, V. Crettaz, and V. Stavridou. Intrusion-tolerant enclaves. In *Proceedings of the 2002 Symposium on Security and Privacy*, pages 216–224, Oakland, California, May 2002. IEEE Computer Society.

[112] C.M. Ellison et al. SPKI certificate theory. Technical report, Internet Engineering Task Force, September 1999. http://www.ietf.org/rfc/rfc2693.txt).

[113] D.R. Engler. The Exokernel Operating System Architecture. Technical report, Ph.D. Thesis, M.I.T., Cambridge, Massachusetts, October 1998.

[114] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *Operating Systems Review*, 29:251–266, December 1995. Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP '95).

[115] V.D. Gligor et al. Design and implementation of Secure Xenix[TM]. In *Proceedings of the 2004 Symposium on Security and Privacy*, Oakland, California, April 1986. IEEE Computer Society. also in *IEEE Transactions on Software Engineering,* vol. SE-13, 2, February 1987, 208–221.

[116] European Communities Commission. *Information Technology Security Evaluation Criteria (ITSEC), Provisional Harmonised Criteria (of France, Germany, the Netherlands, and the United Kingdom)*, June 1991. Version 1.2. Available from the Office for Official Publications of the European Communities, L-2985 Luxembourg, item CD-71-91-502-EN-C. Also available from U.K. CLEF, CESG Room 2/0805, Fiddlers Green Lane, Cheltenham U.K. GLOS GL52 5AJ, or GSA/GISA, Am Nippenkreuz 19, D 5300 Bonn 2, Germany.

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

[118] A.W. Faughn. Interoperability: Is it achievable? Technical report, Harvard University PIRP report, 2001.

[119] R.J. Feiertag, K.N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Proceedings of the Sixth ACM Symposium on Operating System Principles*, pages 57–65, November 1977.

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

[121] 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, Berlin, 11 May 1990.

[122] J. Feller, B. Fitzgerald, S.A. Hissam, and K.R. Lakhani, editors. *Perspectives on Free and Open Source Software*. MIT Press, Cambridge, Massachusetts, 2005.

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

[124] J.-M. Fray, Y. Deswarte, and D. Powell. Intrusion tolerance using fine-grain fragmentation-scattering. In *Proceedings of the 1986 Symposium on Security and Privacy*, pages 194–201, Oakland, California, April 1986. IEEE Computer Society.

[125] C. Gacek and C. Jones. Dependability issues in open source software. Technical report, Department of Computing Science, Dependable Interdisciplinary Research Collaboration, University of Newcastle upon Tyne, Newcastle, England, 2001. Final report for PA5, part of ongoing related work.

[126] C. Gacek, T. Lawrie, and B. Arief. The many meanings of open source. Technical report, Department of Computing Science, University of Newcastle upon Tyne, Newcastle, England, August 2001. Technical Report CS-TR-737.

[127] M. Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Company, New York, 1988.

[128] 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.

[129] S.L. Gerhart and L. Yelowitz. Observations of fallibility in modern programming methodologies. *IEEE Transactions on Software Engineering*, SE-2(3):195–207, September 1976.

[130] J.T. Giffin, S. Jha, and B.P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security 2002*, pages 61–79, San Francisco, California, August 2002. USENIX.

[131] E. Gilbert, J. MacWilliams, and N. Sloane. Codes which detect deception. *Bell System Technical Journal*, 53(3):405–424, 1974.

[132] J.F. Girard and R. Koschke. Finding components in a hierarchy of modules: A step towards architectural understanding. In *Proceedings of the International Conference on Software Maintenance*, pages 72–81, October 1997.

[133] V.D. Gligor. A note on the denial-of-service problem. In *Proceedings of the 1983 Symposium on Security and Privacy*, pages 139–149, Oakland, California, April 1983. IEEE Computer Society.

[134] V.D. Gligor and S.I. Gavrila. Application-oriented security policies and their composition. In *Proceedings of the 1998 Workshop on Security Paradigms*, Cambridge, England, 1998.

[135] V.D. Gligor, S.I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of the 1998 Symposium on Security and Privacy*, Oakland, California, May 1998. IEEE Computer Society.

[136] R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using Galois lattices. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93), SIGPLAN Notices, 28, 10*, pages 394–410, 1993.

[137] R. Godin, H. Mili, G.W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of class hierarchies based on concept (Galois) lattices. *Theory and Practice of Object Systems*, 4(2):117–134, 1998.

[138] W. Goerigk. Compiler verification revisited. In M. Kaufmann, P. Maniolis, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000. Chapter 15.

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

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

[141] 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, California, April 1984. IEEE Computer Society.

[142] A. Goldberg. A specification of Java loading and bytecode verification. In *Fifth ACM Conference on Computer and Communications Security*, pages 49–58, San Francisco, California, November 1998. ACM SIGSAC.

[143] 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.

[144] L. Gong. An overview of Enclaves 1.0. Technical report, SRI International, Menlo Park, California, SRI-CSL-96-01, January 1996. (http://www.csl.sri.com/papers/346/).

[145] L. Gong. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, Massachusetts, 1999.

[146] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[147] L. Gong and X. Qian. The complexibility and composability of secure interoperation. In *Proceedings of the 1994 Symposium on Research in Security and Privacy*, pages 190–200, Oakland, California, May 1994. IEEE Computer Society.

[148] L. Gong and R. Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998.

[149] G. Goth. Richard Clarke talks cybersecurity and JELL-O. *IEEE Security and Privacy*, 2(3):11–15, May-June 2004.

[150] R.M. Graham. Protection in an information processing utility. *Communications of the ACM*, 11(5), May 1968.

[151] C. Gunter, S. Weeks, and A. Wright. Models and languages for digital rights. In *Proceedings of the 2001 Hawaii Intenational Conference on Systems Science*, Honolulu, Hawaii, March 2001. (http://www.star-lab.com/tr/star-tr-01-04.html).

[152] V. Guruswami and M. Sudan. List decoding algorithms for certain contatenated codes. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, pages 181–190, April 2000.

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

[154] 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.

[155] 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.

[156] J.T. Haigh et al. Assured service concepts and models, final technical report, volume 1: Summary. Technical report, Secure Computing Technology Corporation, July 1991.

[157] S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. In *Fifth ACM Conference on Computer and Communications Security*, pages 122–131, San Francisco, California, November 1998. ACM SIGSAC.

[158] R.W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29:147–60, 1950.

[159] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[160] J. Hemenway and D. Gambel. Issues in the specification of composite trustworthy systems. In *Fourth Annual Canadian Computer Security Symposium*, May 1992.

[161] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan Kaufmann, 1996.

[162] H.M. Hinton. Composing partially-specified systems. In *Proceedings of the 1998 Symposium on Security and Privacy*, Oakland, California, May 1998. IEEE Computer Society.

[163] S. Hissam, C.B. Weinstock, D. Plakosh, and J. Asundi. Perspectives on open source software. Technical report, Carnegie-Mellon Software Engineering Institute, Pittsburgh, Pennsylvania 15213-3890, November 2001. CMU/SEI-2001-TR-019 (http://www.sei.cmu.edu/publications/pubweb.html).

[164] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10), October 1974.

[165] D. Hollingworth and R. Bisbey II. Protection errors in operating systems: Allocation/deallocation residuals. Technical report, USC Information Sciences Institute (ISI), Marina Del Rey, California, June 1976.

[166] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium, Notes in Computer Science 16*, pages 171–187. Springer-Verlag, Berlin, 1974.

[167] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40, 1952.

[168] D.A. Huffman. Canonical forms for information-lossless finite-state machines. *IRE Transactions on Circuit Theory (special supplement) and IRE Transactions on Information Theory (special supplement)*, CT-6 and IT-5:41–59, May 1959. A slightly revised version appeared in E.F. Moore, Editor, *Sequential Machines: Selected Papers*, Addison-Wesley, Reading, Massachusetts.

[169] C. Hunt. *TCP/IP Network Administration, 3rd Edition*. O'Reilly & Associates, Sebastopol, California, 2002.

[170] W.A. Hunt Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, December 1989.

[171] IEEE. Standard specifications for public key cryptography. Technical report, IEEE Standards Department, 445 Hoes Lane, P.O. Box 1331, Piscataway, New Jersey 08855-1331, 2000 and ongoing. (http://grouper.ieee.org/groups/1363/).

[172] International Standards Organization. *The Common Criteria for Information Technology Security Evaluation, Version 2.1, ISO 15408*. ISO/NIST/CCIB, 19 September 2000. (http://csrc.nist.gov/cc).

[173] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 20.

[174] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Reading, Massachusetts, 1999.

[175] R. Jagannathan. Transparent multiprocessing in the presence of fail-stop faults. In *Proceedings of the 3rd Workshop on Large-Grain Parallelism*, Pittsburgh, Pennsylvania, October 1989.

[176] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In *Advanced Topics in Dataflow Computing and Multithreading (edited by L. Bic, J-L. Gaudiot, and G. Gao)*. IEEE Computer Society, April 1995.

[177] R. Jagannathan and C. Dodd. GLU programmer's guide v0.9. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, November 1994. CSL Technical Report CSL-94-06.

[178] R. Jagannathan and A.A. Faustini. The GLU programming language. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, November 1990. CSL Technical Report CSL-90-11.

[179] P.A. Janson. Using type extension to organize virtual memory mechanisms. *ACM Operating Systems Review*, 15(4):6–38, October 1981.

[180] S. Jha, O. Sheyner, and J. Wing. Two formal analyses of attack graphs. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 49–64, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society Technical Committee on Security and Privacy.

[181] 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.

[182] C. Jones. Providing a formal basis for dependability notions. Technical report, Department of Computing Science, Dependable Interdisciplinary Research Collaboration, University of Newcastle upon Tyne, Newcastle, England, 2002. UNU/IIST Anniversary Colloquium.

[183] M.F. Kaashoek and A.S. Tanenbaum. Fault tolerance using group communication. *ACM SIGOPS Operating Systems Review*, 25(2):71–74, April 1991.

[184] 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, California, January 1994.

[185] R.Y. Kain. *Computer Architecture: Software and Hardware*. Prentice-Hall, 1988.

[186] 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.

[187] P.A. Karger and H. Kurth. Increased information flow needs for high-assurance composite evaluations. In *Proceedings of the Second International Information Assurance Workshop (IWIA 2004)*, pages 129–140, Charlotte, North Carolina, May 2004. IEEE Computer Society.

[188] P.A. Karger and R.R. Schell. Multics security evaluation: Vulnerability analysis. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC), Classic Papers section*, Las Vegas, Nevada, December 2002. Originally available as U.S. Air Force report ESD-TR-74-193, Vol. II, Hanscomb Air Force Base, Massachusetts.

[189] P.A. Karger and R.R. Schell. Thirty years later: Lessons from the Multics security evaluation. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC), Classic Papers section*, Las Vegas, Nevada, December 2002. http://www.acsac.org/ .

[190] M Kaufmann, J S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishing, Norwell, Massachusetts, 2000.

[191] S. Keung and L. Gong. Enclaves in Java: APIs and Implementations. Technical Report SRI-CSL-96-07, SRI International, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park, California 94025, July 1996.

[192] P. Kocher. Cryptanalysis of Diffie-Hellman, RSA, DSS, and other systems using timing attacks (extended abstract). Technical report, Cryptography Research Inc., 607 Market St, San Francisco, California 94105, December 7 1995.

[193] P.C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science, Advances in Cryptology, Proceedings of Crypto '96*, pages 104–113, Santa Barbara, California, August 1996.

[194] T. Kohno, A. Stubblefield, A.D. Rubin, and D.S. Wallach. Analysis of an electronic voting system. In *Proceedings of the 2004 Symposium on Security and Privacy*, pages 27–40, Oakland, California, May 2004. IEEE Computer Society.

[195] H. Kopetz. Composability in the time-triggered architecture. In *Proceedings of the SAE World Congress*, pages 1–8, Detroit, Michigan, 2000. SAE Press.

[196] M. Kuijper and J.W. Polderman. Reed-solomon list decoding from a system-theoretic perspective. *IEEE Transactions on Information Theory*, 40(2):259–271, February 2004.

[197] L. Lamport. A simple approach to specifying concurrent program systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[198] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[199] B.W. Lampson. Software components: Only the giants survive. In *Computer Systems: papers for Roger Needham, K. Spark-Jones and A. Herbert (editors)*, pages 113–120. Microsoft Research, Cambridge, U.K., February 2003.

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

[201] C.E. Landwehr, A.R. Bull, J.P. McDermott, and W.S. Choi. A taxonomy of computer program security flaws, with examples. Technical report, Center for Secure Information Technology, Information Technology Division, Naval Research Laboratory, Washington, D.C., November 1993.

[202] J.C. Laprie, editor. *Dependability: A Unifying Concept for Reliable Computing and Fault Tolerance*. Springer-Verlag, 1990.

[203] E.S. Lee, P.I.P. Boulton, B.W. Thompson, and R.E. Soper. Composable trusted systems. Technical report, Computer Systems Research Institute, University of Toronto, Technical Report CSRI-272, May 1992.

[204] E.S. Lee, P.I.P. Boulton, B.W. Thomson, and R.E. Soper. Composable trusted systems. Technical report, Computer Systems Research Institute, University of Toronto, Ontario, 31 May 1992. CSRI-272.

[205] K.N. Levitt, S. Crocker, and D. Craigen, editors. VERkshop III: Verification workshop. *ACM SIGSOFT Software Engineering Notes*, 10(4):1–136, August 1985.

[206] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the International Conference on Software Engineering*, pages 349–359, 1997.

[207] U. Lindqvist and P.A. Porras. Detecting computer and network misuse through the Production-Based Expert System Toolset (P-BEST). In *Proceedings of the 1999 Symposium on Security and Privacy*, Oakland, California, May 1999. IEEE Computer Society.

[208] U. Lindqvist and P.A. Porras. eXpert-BSM: A host-based intrusion-detection solution for Sun Solaris. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, New Orleans, Louisiana, 10–14 December 2001.

[209] 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, California, 26–28 April 1982.

[210] S.B. Lipner. Security and source code access: Issues and realities. In *Proceedings of the 2000 Symposium on Security and Privacy*, pages 124–125, Oakland, California, May 2000. IEEE Computer Society.

[211] P.E Livadas and T. Johnson. A new approach to finding objects in programs. *Software Maintenance: Research and Practice*, 6:249–260, 1994.

[212] M. Lubaszewski and B. Courtois. A reliable fail-safe system. *IEEE Transactions on Computers*, C-47(2):236–241, February 1998.

[213] 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, California, April 1988. IEEE Computer Society.

[214] T.F. Lunt and R.A. Whitehurst. The SeaView formal top level specifications and proofs. Final report, Computer Science Laboratory, SRI International, Menlo Park, California, January/February 1989. Volumes 3A and 3B of "Secure Distributed Data Views," SRI Project 1143.

[215] S. Mancoridis and R.C. Holt. Recovering the structure of software systems using tube graph interconnection clustering. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, 1996.

[216] S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organization of source code. In *Proceedings of the International Workshop on Program Comprehension*, pages 42–52, 1998.

[217] S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the International Conference on Software Maintenance*, pages 50–59, 1999.

[218] 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.

[219] H. Mantel. Preserving information flow properties under refinement. In *Proceedings of the 2001 Symposium on Security and Privacy*, pages 78–91, Oakland, California, May 2001. IEEE Computer Society.

[220] H. Mantel. On the composition of secure systems. In *Proceedings of the 2002 Symposium on Security and Privacy*, pages 88–101, Oakland, California, May 2002. IEEE Computer Society.

[221] 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.

[222] 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, California, April 1987. IEEE Computer Society.

[223] D. McCullough. Noninterference and composability of security properties. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 177–186, Oakland, California, April 1988. IEEE Computer Society.

[224] D. McCullough. Ulysses security properties modeling environment: The theory of security. Technical report, Odyssey Research Associates, Ithaca, New York, July 1988.

[225] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6), June 1990.

[226] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. In *Proceedings of the 2002 Symposium on Security and Privacy*, pages 73–87, Oakland, California, May 2002. IEEE Computer Society.

[227] G. McGraw. Will openish source really improve security? In *Proceedings of the 2000 Symposium on Security and Privacy*, pages 128–129, Oakland, California, May 2000. IEEE Computer Society.

[228] G. McGraw. Software security. *IEEE Security and Privacy*, 2(2):80–83, March-April 2004.

[229] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, Reading, Massachusetts, 1996.

[230] 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, California, May 1994. IEEE Computer Society.

[231] P.M. Melliar-Smith and L.E. Moser. Surviving network partitioning. *Computer*, 31(3):62–68, March 1998.

[232] 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.

[233] R. Mercuri. *Electronic Vote Tabulation Checks and Balances*. PhD thesis, Department of Computer Science, University of Pennsylvania, 2001. (http://www.notablesoftware.com/evote.html).

[234] R. Mercuri. A better ballot box: New electronic voting systems pose risks as well as solutions. *IEEE Spectrum*, pages 46–50, October 2002.

[235] R. Mercuri and P.G. Neumann. Verification for electronic balloting systems. In *Secure Electronic Voting, Advances in Information Security, Volume 7*. Kluwer Academic Publishers, Boston, Massachusetts, 2002.

[236] S. Micali. Fair public-key cryptosystems. In *Advances in Cryptology: Proceedings of CRYPTO '92 (E.F. Brickell, editor)*, pages 512–517. Springer-Verlag, Berlin, LCNS 740, 1992.

[237] J. Millen and G. Denker. CAPSL and MuCAPSL. *Journal of Telecommunications and Information Technology*, (4):16–27, 2002.

[238] J.K. Millen. Hookup security for synchronous machines. In *Proceedings of the IEEE Computer Security Foundations Workshop VII*, pages 2–10, Franconia, New Hampshire, June 1994. IEEE Computer Society.

[239] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1997.

[240] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[241] E.F. Moore. Gedanken experiments on sequential machines. In *Automata Studies, Annals of Mathematical Studies, 34, Princeton University Press, 1956*, pages 129–153, 1956. C.E. Shannon and J. McCarthy, editors.

[242] 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.

[243] J S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, December 1989.

[244] J S. Moore. System verification. *Journal of Automated Reasoning*, 5(4):409–410, December 1989.

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

[246] 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.

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

[248] NASA Conference Publication 2377. *Peer Review of a Formal Verification/Design Proof Methodology*, July 1983.

[249] NCSC. *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC)*. National Computer Security Center, December 1985. DOD-5200.28-STD, Orange Book.

[250] G.C. Necula. *Compiling with Proofs*. PhD thesis, Computer Science Department, Carnegie-Mellon University, 1998.

[251] B.J. Nelson. Remote procedure call. Technical report, Research Report CSL-79-9, XEROX Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California, May 1981.

[252] P.G. Neumann. Efficient error-limiting variable-length codes. *IRE Transactions on Information Theory*, IT-8:292–304, July 1962.

[253] P.G. Neumann. On a class of efficient error-limiting variable-length codes. *IRE Transactions on Information Theory*, IT-8:S260–266, September 1962.

[254] P.G. Neumann. Error-limiting coding using information-lossless sequential machines. *IEEE Transactions on Information Theory*, IT-10:108–115, April 1964.

[255] P.G. Neumann. The role of motherhood in the pop art of system programming. In *Proceedings of the ACM Second Symposium on Operating Systems Principles, Princeton, New Jersey*, pages 13–18. ACM, October 1969.

[256] P.G. Neumann. System design for computer networks. In *Computer-Communication Networks (Chapter 2)*, pages 29–81. Prentice-Hall, 1971. N. Abramson and F.F. Kuo (editors).

[257] 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, D.C., 1–4 October 1990. NIST/NCSC.

[258] P.G. Neumann. Can systems be trustworthy with software-implemented crypto? Technical report, Final Report, Project 6402, SRI International, Menlo Park, California, October 1994. For Official Use Only, NOFORN.

[259] P.G. Neumann. Architectures and formal representations for secure systems. Technical report, Final Report, Project 6401, SRI International, Menlo Park, California, October 1995. CSL report 96-05.

[260] P.G. Neumann. *Computer-Related Risks*. ACM Press, New York, and Addison-Wesley, Reading, Massachusetts, 1995.

[261] P.G. Neumann. Practical architectures for survivable systems and networks. Technical report, Final Report, Phase One, Project 1688, SRI International, Menlo Park, California, January 1999. `http://www.csl.sri.com/neumann/arl-one.html`, also available in .ps and .pdf form.

[262] P.G. Neumann. Certitude and rectitude. In *Proceedings of the 2000 International Conference on Requirements Engineering*, page 153, Schaumberg, Illinois, June 2000. IEEE Computer Society.

[263] P.G. Neumann. The potentials of open-box source code in developing robust systems. In *Proceedings of the NATO Conference on Commercial Off-The-Shelf Products in Defence Applications: The Ruthless Pursuit of COTS*, Brussels, Belgium, April 2000. NATO.

[264] P.G. Neumann. Practical architectures for survivable systems and networks. Technical report, Final Report, Phase Two, Project 1688, SRI International, Menlo Park, California, June 2000. (http://www.csl.sri.com/neumann/survivability.html).

[265] P.G. Neumann. Robust nonproprietary software. In *Proceedings of the 2000 Symposium on Security and Privacy*, pages 122–123, Oakland, California, May 2000. IEEE Computer Society. (http://www.csl.sri.com/neumann/ieee00.ps and http://www.csl.sri.com/neumann/ieee00.pdf).

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

[267] P.G. Neumann. Illustrative risks to the public in the use of computer systems and related technology, index to RISKS cases. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 2004. The most recent version is available online in html form for browsing at http://www.csl.sri.com/neumann/illustrative.html, and also in .ps and .pdf form for printing in a much denser format.

[268] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer

Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[269] P.G. Neumann and R.J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, pages 208–216, Las Vegas, Nevada, December 2003. IEEE Computer Society. http://www.acsac.org/ and http://www.csl.sri.com/neumann/psos03.pdf.

[270] P.G. Neumann, J. Goldberg, K.N. Levitt, and J.H. Wensley. A study of fault-tolerant computing. Final report for ARPA, AD 766 974, Stanford Research Institute, Menlo Park, California, July 1973.

[271] P.G. Neumann and D.B. Parker. A summary of computer misuse techniques. In *Proceedings of the Twelfth National Computer Security Conference*, pages 396–407, Baltimore, Maryland, 10–13 October 1989. NIST/NCSC.

[272] P.G. Neumann and P.A. Porras. Experience with EMERALD to date. In *Proceedings of the First USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, Santa Clara, California, April 1999. USENIX. http://www.csl.sri.com/neumann/det99.html.

[273] 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, California, June 1992. Issued as Rome Laboratory report RL-TR-92-152, Rome Laboratory C3AB, Griffiss AFB NY 13441-5700. For Official Use Only.

[274] P.G. Neumann and T.R.N. Rao. Error correction codes for byte-organized arithmetic processors. *IEEE Transactions on Computers*, C-24(3):226–232, March 1975.

[275] P.G. Neumann, editor. VERkshop I: Verification Workshop. *ACM SIGSOFT Software Engineering Notes*, 5(3):4–47, July 1980.

[276] P.G. Neumann, editor. VERkshop II: Verification Workshop. *ACM SIGSOFT Software Engineering Notes*, 6(3):1–63, July 1981.

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

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

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

[280] J.M. Park, E.K.P. Chong, and H.J. Siegel. Efficient multicast packet authentication using signature amortization. In *Proceedings of the 2002 Symposium on Security and Privacy*, pages 227–240, Oakland, California, May 2002. IEEE Computer Society.

[281] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), December 1972.

[282] D.L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5), May 1972.

[283] D.L. Parnas. On a "buzzword": Hierarchical structure. In *Information Processing 74 (Proceedings of the IFIP Congress 1974)*, volume Software, pages 336–339. North-Holland, Amsterdam, 1974.

[284] 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.

[285] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[286] D.L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[287] D.L. Parnas. Mathematical descriptions and specification of software. In *Proceedings of the IFIP World Congress 1994, Volume I*, pages 354–359. IFIP, August 1994.

[288] D.L. Parnas. Software engineering: An unconsummated marriage. *Communications of the ACM*, 40(9):128, September 1997. *Inside Risks* column.

[289] D.L. Parnas. Computer science and software engineering: Filing for divorce? *Communications of the ACM*, 41(8), August 1998. *Inside Risks* column.

[290] 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.

[291] 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.

[292] 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.

[293] 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.

[294] 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.

[295] 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.

[296] D.L. Parnas, A.J. van Schouwen, and S.P. Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, June 1990.

[297] 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.

[298] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, Second Edition*. Morgan Kaufmann, 1997.

[299] R. Perlman. *Network Layer Protocols with Byzantine Robustness*. PhD thesis, MIT, Cambridge, Massachusetts, 1988.

[300] H. Petersen and M. Michels. On signature schemes with threshold verification detecting malicious verifiers. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science, Security Protocols, Proceedings of 5th International Workshop*, pages 67–77, Paris, France, April 1997.

[301] L. Peterson and D. Clark. The Internet: An experiment that escaped from the lab. In *Computer Science: Reflections on the Field, Reflections from the Field*, pages 129–133. National Research Council, National Academy Press, 500 Fifth Ave., Washington, D.C. 20001, 2004.

[302] W.W. Peterson and E.J. Weldon, Jr. *Error-Correcting Codes, 2nd ed.* MIT Press, Cambridge, Massachusetts, 1972.

[303] C.P. Pfleeger. *Security in Computing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[304] P.A. Porras and P.G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the Nineteenth National Computer Security Conference*, pages 353–365, Baltimore, Maryland, 22-25 October 1997. NIST/NCSC.

[305] P.A. Porras, K. Nitz, U. Lindqvist, M. Fong, and P.G. Neumann. Discerning attacker intent. Technical report, Computer Science Laboratory, SRI International, Project 10779, Menlo Park, California, April 2003.

[306] P.A. Porras and A. Valdes. Live traffic analysis of TCP/IP gateways. In *Proceedings of the Symposium on Network and Distributed System Security*. Internet Society, March 1998.

[307] N.E. Proctor. The restricted access processor: An example of formal verification. In *Proceedings of the 1985 Symposium on Security and Privacy*, pages 49–55, Oakland, California, April 1985. IEEE Computer Society.

[308] 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. (http://www.csl.sri.com/neumann/ncs92.html).

[309] B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors. *Predictably Dependable Computing Systems*. Basic Research Series. Springer-Verlag, Berlin, 1995.

[310] T.R.N. Rao. *Error-Control Coding for Computer Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[311] V. Ratan, K. Partridge, J. Reese, and N. Leveson. Safety analysis tools for requirements specification. In *Proceedings of the Eleventh Annual Conference on Computer Assurance, COMPASS '96*, pages 149–160. IEEE Computer Society, 1996.

[312] M. Raynal. A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In *Proceedings of the 1997 High-Assurance Systems Engineering Workshop*, pages 209–214, Washington, D.C., August 1997. IEEE Computer Society.

[313] M. Reiter and K. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[314] J.H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, Cambridge, U.K., 1999.

[315] R. Rivest and B. Lampson. SDSI – a simple distributed security infrastructure. Technical report, MIT Laboratory for Computer Science, 2000. Version 2.0 is available online (http://theory.lcs.mit.edu/˜cis/sdsi.html) along with other documentation and source code.

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

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

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

[319] 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.

[320] E. Rosen. Vulnerabilities of network control protocols. *ACM SIGSOFT Software Engineering Notes*, 6(1):6–8, January 1981.

[321] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. Technical report, NASA Langley Research Center, June 1999. Contractor Report CR-1999-209347; also issued as FAA DOT/FAA/AR-99/58.

[322] J. Rushby. Modular certification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, June 2002.

[323] 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.

[324] 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.

[325] J.M. Rushby. Composing trustworthy systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, July 1991.

[326] 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.

[327] J.M. Rushby and B. Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, July 1983.

[328] J.M. Rushby and B. Randell. A distributed secure system. Technical Report 182, Computing Laboratory, University of Newcastle upon Tyne, May 1983.

[329] 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, California, April 1983. IEEE Computer Society.

[330] 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.

[331] 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, California, February 1989. Also available as NASA Contractor Report 4239.

[332] 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.

[333] J.H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

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

[335] 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, editor, *Advances in Computer System Security*, Vol. 3, Artech House, Dedham, Massachusetts, 1988.

[336] W.L. Schiller. The design and specification of a security kernel for the PDP-11/45. Technical Report MTR-2934, Mitre Corporation, Bedford, Massachusetts, March 1975.

[337] F.B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, New York, August 1987.

[338] F.B. Schneider. Open source in security: Visiting the bizarre. In *Proceedings of the 2000 Symposium on Security and Privacy*, pages 126–127, Oakland, California, May 2000. IEEE Computer Society.

[339] F.B. Schneider, editor. Research to support robust cyber defense. Technical report, Study Committee for J. Lala, DARPA, May 2000. Slides only.

[340] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C: Second Edition*. John Wiley and Sons, New York, 1996.

[341] B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley and Sons, New York, 2000.

[342] B. Schneier and D. Banisar. *The Electronic Privacy Papers*. John Wiley and Sons, New York, 1997.

[343] M.D. Schroeder. Cooperation of mutually suspicious subsystems in a computer utility. Technical report, Ph.D. Thesis, M.I.T., Cambridge, Massachusetts, September 1972.

[344] 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).

[345] M.D. Schroeder and J.H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3), March 1972.

[346] R.W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the International Conference On Software Engineering*, pages 83–92, 1991.

[347] Secure Computing Technology Center. LOCK formal top level specification, volumes 1-6. Technical report, SCTC, 1988.

[348] Secure Computing Technology Center. LOCK software B-specification, vol. 2. Technical report, SCTC, 1988.

[349] A. Shamir and E. Tromer. Acoustic cryptanalysis: On nosy people and noisy machines. *preliminary proof-of-concept presentation*, 2004.

[350] D. Shands, E. Wu, J. Horning, and S. Weeks. Spice: Configurationa synthesis for policies enforcement. Technical report, MacAfee Research Technical Report 04-018, June 2004.

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

[352] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2003 Symposium on Security and Privacy*, pages 273–284, Oakland, California, May 2003. IEEE Computer Society.

[353] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, SE-25(6):749–768, 1999.

[354] N.J.A. Sloane and F.J. MacWilliams. *The Theory of Error-Correcting Codes, 9th reprint*. North-Holland, 1998.

[355] M.A. Smith. Portals: Toward an application framework for interoperability. *Communications of the ACM*, 47(10):93–97, October 2004.

[356] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *IEEE Transactions on Software Engineering and Methodology*, 5(2):146–189, 1996.

[357] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the International Symposium on Foundations of Software Engineering*, 1998.

[358] G. Snider and J. Hays. The modix kernel. In *1989 Winter USENIX Conference Proceedings*, pages 377–392, San Diego, California, February 1989.

[359] I. Sommerville. *Software Engineering*. Addison-Wesley, Reading, Massachusetts, 2001. Sixth Edition.

[360] D. Song, D. Zuckerman, and J.D. Tygar. Expander graphs for digital stream authentication and robust overlay networks. In *Proceedings of the 2002 Symposium on Security and Privacy*, pages 258–270, Oakland, California, May 2002. IEEE Computer Society.

[361] 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.

[362] J. Staddon, S. Miner, M. Franklin, D. Balfanz, M. Malkin, and D. Dean. Self-healing key distribution with revocation. In *Proceedings of the 2002 Symposium on Security and Privacy*, pages 241–257, Oakland, California, May 2002. IEEE Computer Society.

[363] D.I. Sutherland. A model of information flow. In *Proceedings of the Ninth National Computer Security Conference*, pages 175–183, September 1986.

[364] K.L. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[365] M. Tinto. The design and evaluation of INFOSEC systems: The computer security contribution to the composition discussion. Technical report, National Computer Security Center, June 1992. C Technical Report 32-92.

[366] P. Tonella. Concept analysis for module restructuring. *IEEE Transactions on Software Engineering*, SE-27(4):351–363, 2001.

[367] I.L. Traiger, J. Gray, C.A. Galtieri, and B.G. Lindsay. Transactions and consistency in distributed database systems. *ACM TODS*, 7(3):323–342, September 1982.

[368] Unspecified. Composability constraints of multilevel systems. Technical report, Integrated Computer Systems, Inc., 215 South Rutgers Ave., Oak Ridge, Tennessee, June 1994.

[369] USGAO. Defense acquisitions: Knowledge of software suppliers needed to manage risks. Technical report, U.S. General Accounting Office, GAO-04-078, Washington, D.C., May 2004.

[370] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, pages 43–98, Princeton University, Princeton, New Jersey, 1956.

[371] D. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. PhD thesis, Division of Computer Science, University of California, Berkeley, December 2000. (http://www.cs.berkeley.edu/˜daw).

[372] W.H. Ware. A retrospective of the criteria movement. In *Proceedings of the Eighteenth National Information Systems Security Conference*, pages 582–588, Baltimore, Maryland, 10–13 October 1995. NIST/NCSC.

[373] P. Wayner. *Translucent Databases*. Flyzone Press, Baltimore, Maryland, 2002.

[374] S. Weeks. Understanding trust management systems. In *Proceedings of the 2001 Symposium on Security and Privacy*, Oakland, California, May 2001. IEEE Computer Society. (http://www.star-lab.com/tr/star-tr-01-02.html).

[375] L. Weinstein. The devil you know. *Communications of the ACM*, 46(12):144, December 2003.

[376] L. Weinstein. TRIPOLI: An Empowered E-Mail Environment. Technical report, People for Internet Responsibility, January 2004.

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

[378] 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, California, June 1982.

[379] D.A. Wheeler. *Secure Programming for Linux and Unix HOWTO*. 2003.

[380] D.A. Wheeler. Secure programmer: Minimizing privileges; taking the fangs out of bugs. May 2004.

[381] I. White. Wrapping the COTS dilemma. In *Proceedings of the NATO Conference on Commercial Off-The-Shelf Products in Defence Applications: The Ruthless Pursuit of COTS*, Brussels, Belgium, April 2000. NATO.

[382] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley, Reading, Massachusetts, 1995.

[383] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2):28–34, February 1973.

[384] A. Yeh, D. Harris, and H. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Proceedings of the Working Conference on Reverse Engineering*, pages 227–236, 1995.

[385] W.D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989.

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

[387] 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, California, April 1988. IEEE Computer Society. Also in *IEEE Transactions on Software Engineering,* SE-16, 12, June 1990, 581–592).

[388] 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.

[389] A. Zakinthinos and E.S. Lee. Composing secure systems that have emergent properties. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 117–122, Rockport, Massachusetts, June 1998.

[390] P.R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, Massachusetts, 1995.

[391] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: a translation validator for optimizing compilers. In *Electronic Notes in Theoretical Computer Science*, 2002. Preliminary version at www.cs.nyu.edu/˜zuck/pubs/, final version at www.elsevier.ai/locate/entcs.

Index