COMPUTER SECURITY

Art and Science

Matt Bishop

Library of Congress Cataloging-in-Publication Data

Bishop, Matt.
Computer security : art and science / Matt Bishop.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-44099-7 (alk. paper)
1. Computer security. I. Title.

QA76.9.A25 B56 2002
005.8—dc21 2002026219

ISBN 0-201-44099-7
Text printed on recycled paper
3 4 5 6 7 8 9 10—CRW—0807060504
Fifth printing, September 2004

Preface

HORTENSIO: Madam, before you touch the instrument
To learn the order of my fingering,
I must begin with rudiments of art
To teach you gamouth in a briefer sort,
More pleasant, pithy and effectual,
Than hath been taught by any of my trade;
And there it is in writing, fairly drawn.
—*The Taming of the Shrew*, III, i, 62–68.

On September 11, 2001, terrorists seized control of four airplanes. Three were flown into buildings, and a fourth crashed, with catastrophic loss of life. In the aftermath, the security and reliability of many aspects of society drew renewed scrutiny. One of these aspects was the widespread use of computers and their interconnecting networks.

The issue is not new. In 1988, approximately 5,000 computers throughout the Internet were rendered unusable within 4 hours by a program called a *worm* [432].[1] While the spread, and the effects, of this program alarmed computer scientists, most people were not worried because the worm did not affect their lives or their ability to do their jobs. In 1993, more users of computer systems were alerted to such dangers when a set of programs called *sniffers* were placed on many computers run by network service providers and recorded login names and passwords [374].

After an attack on Tsutomu Shimomura's computer system, and the fascinating way Shimomura followed the attacker's trail, which led to his arrest [914], the public's interest and apprehension were finally aroused. Computers were now vulnerable. Their once reassuring protections were now viewed as flimsy.

Several films explored these concerns. Movies such as *War Games* and *Hackers* provided images of people who can, at will, wander throughout computers and networks, maliciously or frivolously corrupting or destroying information it may have taken millions of dollars to amass. (Reality intruded on *Hackers* when the World Wide Web page set up by MGM/United Artists was quickly altered to present an irreverent commentary on the movie and to suggest that viewers see *The Net*

[1] Section 22.4 discusses computer worms.

instead. Paramount Pictures denied doing this [448].) Another film, *Sneakers*, presented a picture of those who test the security of computer (and other) systems for their owners and for the government.

Goals

This book has three goals. The first is to show the importance of theory to practice and of practice to theory. All too often, practitioners regard theory as irrelevant and theoreticians think of practice as trivial. In reality, theory and practice are symbiotic. For example, the theory of covert channels, in which the goal is to limit the ability of processes to communicate through shared resources, provides a mechanism for evaluating the effectiveness of mechanisms that confine processes, such as sandboxes and firewalls. Similarly, business practices in the commercial world led to the development of several security policy models such as the Clark-Wilson model and the Chinese Wall model. These models in turn help the designers of security policies better understand and evaluate the mechanisms and procedures needed to secure their sites.

The second goal is to emphasize that computer security and cryptography are different. Although cryptography is an essential component of computer security, it is by no means the only component. Cryptography provides a mechanism for performing specific functions, such as preventing unauthorized people from reading and altering messages on a network. However, unless developers understand the context in which they are using cryptography, and unless the assumptions underlying the protocol and the cryptographic mechanisms apply to the context, the cryptography may not add to the security of the system. The canonical example is the use of cryptography to secure communications between two low-security systems. If only trusted users can access the two systems, cryptography protects messages in transit. But if untrusted users can access either system (through authorized accounts or, more likely, by breaking in), the cryptography is not sufficient to protect the messages. The attackers can read the messages at either endpoint.

The third goal is to demonstrate that computer security is not just a science but also an art. It is an art because no system can be considered secure without an examination of how it is to be used. The definition of a "secure computer" necessitates a statement of requirements and an expression of those requirements in the form of authorized actions and authorized users. (A computer engaged in work at a university may be considered "secure" for the purposes of the work done at the university. When moved to a military installation, that same system may not provide sufficient control to be deemed "secure" for the purposes of the work done at that installation.) How will people, as well as other computers, interact with the computer system? How clear and restrictive an interface can a designer create without rendering the system unusable while trying to prevent unauthorized use or access to the data or resources on the system?

Just as an artist paints his view of the world onto canvas, so does a designer of security features articulate his view of the world of human/machine interaction in the security policy and mechanisms of the system. Two designers may use entirely different designs to achieve the same creation, just as two artists may use different subjects to achieve the same concept.

Computer security is also a science. Its theory is based on mathematical constructions, analyses, and proofs. Its systems are built in accordance with the accepted practices of engineering. It uses inductive and deductive reasoning to examine the security of systems from key axioms and to discover underlying principles. These scientific principles can then be applied to untraditional situations and new theories, policies, and mechanisms.

Philosophy

Key to understanding the problems that exist in computer security is a recognition that the problems are not new. They are old problems, dating from the beginning of computer security (and, in fact, arising from parallel problems in the noncomputer world). But the locus has changed as the field of computing has changed. Before the mid-1980s, mainframe and mid-level computers dominated the market, and computer security problems and solutions were phrased in terms of securing files or processes on a single system. With the rise of networking and the Internet, the arena has changed. Workstations and servers, and the networking infrastructure that connects them, now dominate the market. Computer security problems and solutions now focus on a networked environment. However, if the workstations and servers, and the supporting network infrastructure, are viewed as a single system, the models, theories, and problem statements developed for systems before the mid-1980s apply equally well to current systems.

As an example, consider the issue of assurance. In the early period, assurance arose in several ways: formal methods and proofs of correctness, validation of policy to requirements, and acquisition of data and programs from trusted sources, to name a few. Those providing assurance analyzed a single system, the code on it, and the sources (vendors and users) from which the code could be acquired to ensure that either the sources could be trusted or the programs could be confined adequately to do minimal damage. In the later period, the same basic principles and techniques apply, except that the scope of some has been greatly expanded (from a single system and a small set of vendors to the world-wide Internet). The work on proof-carrying code, an exciting development in which the proof that a downloadable program module satisfies a stated policy is incorporated into the program itself,[2] is an example of this expansion. It extends the notion of a proof of consistency with a stated policy. It advances the technology of the earlier period into the later period. But in order to

[2] Section 22.7.5.1 discusses proof-carrying code.

understand it properly, one must understand the ideas underlying the concept of proof-carrying code, and these ideas lie in the earlier period.

As another example, consider Saltzer and Schroeder's principles of secure design.[3] Enunciated in 1975, they promote simplicity, confinement, and understanding. When security mechanisms grow too complex, attackers can evade or bypass them. Many programmers and vendors are learning this when attackers break into their systems and servers. The argument that the principles are old, and somehow outdated, rings hollow when the result of their violation is a nonsecure system.

The work from the earlier period is sometimes cast in terms of systems that no longer exist and that differ in many ways from modern systems. This does not vitiate the ideas and concepts, which also underlie the work done today. Once these ideas and concepts are properly understood, applying them in a multiplicity of environments becomes possible. Furthermore, the current mechanisms and technologies will become obsolete and of historical interest themselves as new forms of computing arise, but the underlying principles will live on, to underlie the next generation—indeed the next era—of computing.

The philosophy of this book is that certain key concepts underlie all of computer security, and that the study of all parts of computer security enriches the understanding of all parts. Moreover, critical to an understanding of the applications of security-related technologies and methodologies is an understanding of the theory underlying those applications.

Advances in the theory of computer protection have illuminated the foundations of security systems. Issues of abstract modeling, and modeling to meet specific environments, lead to systems designed to achieve a specific and rewarding goal. Theorems about composability of policies[4] and the undecidability of the general security question[5] have indicated the limits of what can be done. Much work and effort are continuing to extend the borders of those limits.

Application of these results has improved the quality of the security of the systems being protected. However, the issue is how compatibly the assumptions of the model (and theory) conform to the environment to which the theory is applied. Although our knowledge of how to apply these abstractions is continually increasing, we still have difficulty correctly transposing the relevant information from a realistic setting to one in which analyses can then proceed. Such abstraction often eliminates vital information. The omitted data may pertain to security in nonobvious ways. Without this information, the analysis is flawed.

The practitioner needs to know both the theoretical and practical aspects of the art and science of computer security. The theory demonstrates what is possible. The practical makes known what is feasible. The theoretician needs to understand the constraints under which these theories are used, how their results are translated into practical tools and methods, and how realistic are the assumptions underlying the theories. *Computer Security: Art and Science* tries to meet these needs.

[3] Chapter 13 discusses these principles.

[4] See Chapter 8, "Noninterference and Policy Composition."

[5] See Section 3.2, "Basic Results."

Unfortunately, no single work can cover all aspects of computer security, so this book focuses on those parts that are, in the author's opinion, most fundamental and most pervasive. The mechanisms exemplify the applications of these principles.

Organization

The organization of this book reflects its philosophy. It begins with mathematical fundamentals and principles that provide boundaries within which security can be modeled and analyzed effectively. The mathematics provides a framework for expressing and analyzing the requirements of the security of a system. These policies constrain what is allowed and what is not allowed. Mechanisms provide the ability to implement these policies. The degree to which the mechanisms correctly implement the policies, and indeed the degree to which the policies themselves meet the requirements of the organizations using the system, are questions of assurance. Exploiting failures in policy, in implementation, and in assurance comes next, as well as mechanisms for providing information on the attack. The book concludes with the applications of both theory and policy focused on realistic situations. This natural progression emphasizes the development and application of the principles existent in computer security.

Part 1, "Introduction," describes what computer security is all about and explores the problems and challenges to be faced. It sets the context for the remainder of the book.

Part 2, "Foundations," deals with basic questions such as how "security" can be clearly and functionally defined, whether or not it is realistic, and whether or not it is decidable. If it is decidable, under what conditions is it decidable, and if not, how must the definition be bounded in order to make it decidable?

Part 3, "Policy," probes the relationship between policy and security. The definition of "security" depends on policy. In Part 3 we examine several types of policies, including the ever-present fundamental questions of trust, analysis of policies, and the use of policies to constrain operations and transitions.

Part 4, "Implementation I: Cryptography," discusses cryptography and its role in security. It focuses on applications and discusses issues such as key management and escrow, key distribution, and how cryptosystems are used in networks. A quick study of authentication completes Part 4.

Part 5, "Implementation II: Systems," considers how to implement the requirements imposed by policies using system-oriented techniques. Certain design principles are fundamental to effective security mechanisms. Policies define who can act and how they can act, and so identity is a critical aspect of implementation. Mechanisms implementing access control and flow control enforce various aspects of policies.

Part 6, "Assurance," presents methodologies and technologies for ascertaining how well a system, or a product, meets its goals. After setting the background, to explain exactly what "assurance" is, the art of building systems to meet varying levels of assurance is discussed. Formal verification methods play a role. Part 6 shows how the progression of standards has enhanced our understanding of assurance techniques.

Part 7, "Special Topics," discusses some miscellaneous aspects of computer security. Malicious logic thwarts many mechanisms. Despite our best efforts at high assurance, systems today are replete with vulnerabilities. Why? How can a system be analyzed to detect vulnerabilities? What models might help us improve the state of the art? Given these security holes, how can we detect attackers who exploit them? A discussion of auditing flows naturally into a discussion of intrusion detection—a detection method for such attacks.

Part 8, "Practicum," presents examples of how to apply the principles discussed throughout the book. It begins with networks and proceeds to systems, users, and programs. Each chapter states a desired policy and shows how to translate that policy into a set of mechanisms and procedures that support the policy. Part 8 tries to demonstrate that the material covered elsewhere can be, and should be, used in practice.

Each chapter in this book ends with a summary, descriptions of some research issues, and some suggestions for further reading. The summary highlights the important ideas in the chapter. The research issues are current "hot topics" or are topics that may prove to be fertile ground for advancing the state of the art and science of computer security. Interested readers who wish to pursue the topics in any chapter in more depth can go to some of the suggested readings. They expand on the material in the chapter or present other interesting avenues.

Roadmap

This book is both a reference book and a textbook. Its audience is undergraduate and graduate students as well as practitioners. This section offers some suggestions on approaching the book.

Dependencies

Chapter 1 is fundamental to the rest of the book and should be read first. After that, however, the reader need not follow the chapters in order. Some of the dependencies among chapters are as follows.

Chapter 3 depends on Chapter 2 and requires a fair degree of mathematical maturity. Chapter 2, on the other hand, does not. The material in Chapter 3 is for the most part not used elsewhere (although the existence of the first section's key result, the undecidability theorem, is mentioned repeatedly). It can be safely skipped if the interests of the reader lie elsewhere.

The chapters in Part 3 build on one another. The formalisms in Chapter 5 are called on in Chapters 19 and 20, but nowhere else. Unless the reader intends to delve into the sections on theorem proving and formal mappings, the formalisms may be skipped. The material in Chapter 8 requires a degree of mathematical maturity, and this material is used sparingly elsewhere. Like Chapter 3, Chapter 8 can be skipped by the reader whose interests lie elsewhere.

Chapters 9, 10, and 11 also build on one another in order. A reader who has encountered basic cryptography will have an easier time with the material than one who has not, but the chapters do not demand the level of mathematical experience that Chapters 3 and 8 require. Chapter 12 does not require material from Chapter 10 or Chapter 11, but it does require material from Chapter 9.

Chapter 13 is required for all of Part 5. A reader who has studied operating systems at the undergraduate level will have no trouble with Chapter 15. Chapter 14 uses the material in Chapter 11; Chapter 16 builds on material in Chapters 5, 13, and 15; and Chapter 17 uses material in Chapters 4, 13, and 16.

Chapter 18 relies on information in Chapter 4. Chapter 19 builds on Chapters 5, 13, 15, and 18. Chapter 20 presents highly mathematical concepts and uses material from Chapters 18 and 19. Chapter 21 is based on material in Chapters 5, 18, and 19; it does not require Chapter 20. For all of Part 5, a knowledge of software engineering is very helpful.

Chapter 22 draws on ideas and information in Chapters 5, 6, 9, 13, 15, and 17 (and for Section 22.6, the reader should read Section 3.1). Chapter 23 is self-contained, although it implicitly uses many ideas from assurance. It also assumes a good working knowledge of compilers, operating systems, and in some cases networks. Many of the flaws are drawn from versions of the UNIX operating system, or from Windows systems, and so a working knowledge of either or both systems will make some of the material easier to understand. Chapter 24 uses information from Chapter 4, and Chapter 25 uses material from Chapter 24.

The practicum chapters are self-contained and do not require any material beyond Chapter 1. However, they point out relevant material in other sections that augments the information and (we hope) the reader's understanding of that information.

Background

The material in this book is at the advanced undergraduate level. Throughout, we assume that the reader is familiar with the basics of compilers and computer architecture (such as the use of the program stack) and operating systems. The reader should also be comfortable with modular arithmetic (for the material on cryptography). Some material, such as that on formal methods (Chapter 20) and the mathematical theory of computer security (Chapter 3 and the formal presentation of policy models), requires considerable mathematical maturity. Other specific recommended background is presented in the preceding section. Part 9, "End Matter," contains material that will be helpful to readers with backgrounds that lack some of the recommended material.

Examples are drawn from many systems. Many come from the UNIX operating system or variations of it (such as Linux). Others come from the Windows family of systems. Familiarity with these systems will help the reader understand many examples easily and quickly.

Undergraduate Level

An undergraduate class typically focuses on applications of theory and how students can use the material. The specific arrangement and selection of material depends on the focus of the class, but all classes should cover some basic material—notably that in Chapters 1, 9, and 13, as well as the notion of an access control matrix, which is discussed in Sections 2.1 and 2.2.

Presentation of real problems and solutions often engages undergraduate students more effecively than presentation of abstractions. The special topics and the practicum provide a wealth of practical problems and ways to deal with them. This leads naturally to the deeper issues of policy, cryptography, noncryptographic mechanisms, and assurance. The following are sections appropriate for nonmathematical undergraduate courses in these topics.

- *Policy:* Sections 4.1 through 4.4 describe the notion of policy. The instructor should select one or two examples from Sections 5.1, 5.2.1, 6.2, 6.4, 7.1.1, and 7.2, which describe several policy models informally. Section 7.4 discusses role-based access control.
- *Cryptography:* Key distribution is discussed in Sections 10.1 and 10.2, and a common form of public key infrastructures (called PKIs) is discussed in Section 10.4.2. Section 11.1 points out common errors in using cryptography. Section 11.3 shows how cryptography is used in networks, and the instructor should use one of the protocols in Section 11.4 as an example. Chapter 12 offers a look at various forms of authentication, including noncryptographic methods.
- *Noncryptographic mechanisms:* Identity is the basis for many access control mechanisms. Sections 14.1 through 14.4 discuss identity on a system, and Section 14.6 discusses identity and anonymity on the Web. Sections 15.1 and 15.2 explore two mechanisms for controlling access to files, and Section 15.4 discusses the ring-based mechanism underlying the notion of multiple levels of privilege. If desired, the instructor can cover sandboxes by using Sections 17.1 and 17.2, but because Section 17.2 uses material from Sections 4.5 and 4.5.1, the instructor will need to go over those sections as well.
- *Assurance:* Chapter 18 provides a basic introduction to the often overlooked topic of assurance.

Graduate Level

A typical introductory graduate class can focus more deeply on the subject than can an undergraduate class. Like an undergraduate class, a graduate class should cover Chapters 1, 9, and 13. Also important are the undecidability results in Sections 3.1 and 3.2, which require that Chapter 2 be covered. Beyond that, the instructor can choose from a variety of topics and present them to whatever depth is appropriate. The following are sections suitable for graduate study.

- *Policy models:* Part 3 covers many common policy models both informally and formally. The formal description is much easier to understand once the informal description is understood, so in all cases both should be covered. The controversy in Section 5.4 is particularly illuminating to students who have not considered the role of policy and the nature of a policy. Chapter 8 is a highly formal discussion of the foundations of policy and is appropriate for students with experience in formal mathematics. Students without such a background will find it quite difficult.
- *Cryptography:* Part 4 focuses on the applications of cryptography, not on cryptography's mathematical underpinnings.[6] It discusses areas of interest critical to the use of cryptography, such as key management and some basic cryptographic protocols used in networking.
- *Noncryptographic mechanisms:* Issues of identity and certification are complex and generally poorly understood. Section 14.5 covers these problems. Combining this with the discussion of identity on the Web (Section 14.6) raises issues of trust and naming. Chapters 16 and 17 explore issues of information flow and confining that flow.
- *Assurance:* Traditionally, assurance is taught as formal methods, and Chapter 20 serves this purpose. In practice, however, assurance is more often accomplished by using structured processes and techniques and informal but rigorous arguments of justification, mappings, and analysis. Chapter 19 emphasizes these topics. Chapter 21 discusses evaluation standards and relies heavily on the material in Chapters 18 and 19 and some of the ideas in Chapter 20.
- *Miscellaneous Topics:* Section 22.6 presents a proof that the generic problem of determining if a generic program is a computer virus is in fact undecidable. The theory of penetration studies in Section 23.2, and the more formal approach in Section 23.5, illuminate the analysis of systems for vulnerabilities. If the instructor chooses to cover intrusion detection (Chapter 25) in depth, it should be understood that this discussion draws heavily on the material on auditing (Chapter 24).
- *Practicum:* The practicum (Part 8) ties the material in the earlier part of the book to real-world examples and emphasizes the applications of the theory and methodologies discussed earlier.

[6] The interested reader will find a number of books covering aspects of this subject [240, 590, 695, 702, 888, 897, 998].

Practitioners

Practitioners in the field of computer security will find much to interest them. The table of contents and the index will help them locate specific topics. A more general approach is to start with Chapter 1 and then proceed to Part 8, the practicum. Each chapter has references to other sections of the text that explain the underpinnings of the material. This will lead the reader to a deeper understanding of the reasons for the policies, settings, configurations, and advice in the practicum. This approach also allows readers to focus on those topics that are of most interest to them.

Special Acknowledgment

Elisabeth Sullivan contributed the assurance part of this book. She wrote several drafts, all of which reflect her extensive knowledge and experience in that aspect of computer security. I am particularly grateful to her for contributing her real-world knowledge of how assurance is managed. Too often, books recount the mathematics of assurance without recognizing that other aspects are equally important and more widely used. These other aspects shine through in the assurance section, thanks to Liz. As if that were not enough, she made several suggestions that improved the policy part of this book. I will always be grateful for her contribution, her humor, and especially her friendship.

Acknowledgments

Many others contributed to this book in various ways. Special thanks to Steven Alexander, Jim Alves-Foss, Bill Arbaugh, Andrew Arcilla, Alex Aris, Rebecca Bace, Belinda Bashore, Vladimir Berman, Ziad El Bizri, Logan Browne, Terry Brugger, Serdar Cabuk, Raymond Centeno, Lisa Clark, Michael Clifford, Christopher Clifton, Dan Coming, Kay Connelly, Crispin Cowan, Tom Daniels, Dimitri DeFigueiredo, Joseph-Patrick Dib, Jeremy Frank, Robert Fourney, Martin Gagne, Ron Gove, James Hinde, Xuxian Jiang, Jesper Johansson, Mark Jones, Calvin Ko, Mark-Neil Ledesma, Ken Levine, Karl Levitt, Yunhua Lu, Gary McGraw, Alexander Meau, Nasir Memon, Mark Morrissey, Ather Nawaz, Iulian Neamtiu, Kimberly Nico, Stephen Northcutt, Rafael Obelheiro, Josko Orsulic, Holly Pang, Ryan Poling, Sung Park, Ashwini Raina, Jorge Ramos, Brennen Reynolds, Peter Rozental, Christoph Schuba, David Shambroom, Jonathan Shapiro, Clay Shields, Sriram Srinivasan, Mahesh V. Tripunitara, Tom Walcott, James Walden, Dan Watson, Guido Wedig, Chris Wee, Patrick Wheeler, Paul Williams, Bonnie Xu, Xiaoduan Ye, Lara Whelan, John Zachary, Aleksandr Zingorenko, and to everyone in my computer security classes, who (knowingly or unknowingly) helped me develop and test this material.

The Addison-Wesley folks, Kathleen Billus, Susannah Buzard, Bernie Gaffney, Amy Fleischer, Helen Goldstein, Tom Stone, Asdis Thorsteinsson, and most especially my editor, Peter Gordon, were incredibly patient and helpful, despite fears that this book would never materialize. The fact that it did so is in great measure attributable to their hard work and encouragement. I also thank the production people at Argosy, especially Beatriz Valdés and Craig Kirkpatrick, for their wonderful work.

Dorothy Denning, my advisor in graduate school, guided me through the maze of computer security when I was just beginning. Peter Denning, Barry Leiner, Karl Levitt, Peter Neumann, Marvin Schaefer, Larry Snyder, and several others influenced my approach to the subject. I hope this work reflects in some small way what they gave to me and passes a modicum of it along to my readers.

I also thank my parents, Leonard Bishop and Linda Allen. My father, a writer, gave me some useful tips on writing, which I tried to follow. My mother, a literary agent, helped me understand the process of getting the book published, and supported me throughout.

Finally, I would like to thank my family for their support throughout the writing. Sometimes they wondered if I would ever finish. My wife Holly and our children Steven, David, and Caroline were very patient and understanding and made sure I had time to work on the book. Our oldest daughter Heidi and her husband Mike also provided much love and encouragement and the most wonderful distraction: our grandson—Skyler. To all, my love and gratitude.

Part 1

Introduction

Writers say "To write a good book, first tell them what you are going to tell them, then tell them, then tell them what you told them." This is the "what we're going to tell you" part.

Chapter 1, "An Overview of Computer Security," presents the underpinnings of computer security and an overview of the important issues to place them in context. It begins with a discussion of what computer security is and how threats are connected to security services. The combination of desired services makes up a policy, and mechanisms enforce the policy. All rely on underlying assumptions, and the systems built on top of these assumptions lead to issues of assurance. Finally, the operational and human factors affect the mechanisms used as well as the policy.

Chapter 1
An Overview of Computer Security

ANTONIO: Whereof what's past is prologue, what to come
In yours and my discharge.
—*The Tempest*, II, i, 257–258.

This chapter presents the basic concepts of computer security. The remainder of the book will elaborate on these concepts in order to reveal the logic underlying the principles of these concepts.

We begin with basic security-related services that protect against threats to the security of the system. The next section discusses security policies that identify the threats and define the requirements for ensuring a secure system. Security mechanisms detect and prevent attacks and recover from those that succeed. Analyzing the security of a system requires an understanding of the mechanisms that enforce the security policy. It also requires a knowledge of the related assumptions and trust, which lead to the threats and the degree to which they may be realized. Such knowledge allows one to design better mechanisms and policies to neutralize these threats. This process leads to risk analysis. Human beings are the weakest link in the security mechanisms of any system. Therefore, policies and procedures must take people into account. This chapter discusses each of these topics.

1.1 The Basic Components

Computer security rests on confidentiality, integrity, and availability. The interpretations of these three aspects vary, as do the contexts in which they arise. The interpretation of an aspect in a given environment is dictated by the needs of the individuals, customs, and laws of the particular organization.

1.1.1 Confidentiality

Confidentiality is the concealment of information or resources. The need for keeping information secret arises from the use of computers in sensitive fields such as government and industry. For example, military and civilian institutions in the government often restrict access to information to those who need that information. The first formal work in computer security was motivated by the military's attempt to implement controls to enforce a "need to know" principle. This principle also applies to industrial firms, which keep their proprietary designs secure lest their competitors try to steal the designs. As a further example, all types of institutions keep personnel records secret.

Access control mechanisms support confidentiality. One access control mechanism for preserving confidentiality is cryptography, which scrambles data to make it incomprehensible. A *cryptographic key* controls access to the unscrambled data, but then the cryptographic key itself becomes another datum to be protected.

EXAMPLE: Enciphering an income tax return will prevent anyone from reading it. If the owner needs to see the return, it must be deciphered. Only the possessor of the cryptographic key can enter it into a deciphering program. However, if someone else can read the key when it is entered into the program, the confidentiality of the tax return has been compromised.

Other system-dependent mechanisms can prevent processes from illicitly accessing information. Unlike enciphered data, however, data protected only by these controls can be read when the controls fail or are bypassed. Then their advantage is offset by a corresponding disadvantage. They can protect the secrecy of data more completely than cryptography, but if they fail or are evaded, the data becomes visible.

Confidentiality also applies to the existence of data, which is sometimes more revealing than the data itself. The precise number of people who distrust a politician may be less important than knowing that such a poll was taken by the politician's staff. How a particular government agency harassed citizens in its country may be less important than knowing that such harassment occurred. Access control mechanisms sometimes conceal the mere existence of data, lest the existence itself reveal information that should be protected.

Resource hiding is another important aspect of confidentiality. Sites often wish to conceal their configuration as well as what systems they are using; organizations may not wish others to know about specific equipment (because it could be used without authorization or in inappropriate ways), and a company renting time from a service provider may not want others to know what resources it is using. Access control mechanisms provide these capabilities as well.

All the mechanisms that enforce confidentiality require supporting services from the system. The assumption is that the security services can rely on the kernel, and other agents, to supply correct data. Thus, assumptions and trust underlie confidentiality mechanisms.

1.1.2 Integrity

Integrity refers to the trustworthiness of data or resources, and it is usually phrased in terms of preventing improper or unauthorized change. Integrity includes data integrity (the content of the information) and origin integrity (the source of the data, often called *authentication*). The source of the information may bear on its accuracy and credibility and on the trust that people place in the information.This dichotomy illustrates the principle that the aspect of integrity known as credibility is central to the proper functioning of a system. We will return to this issue when discussing malicious logic.

EXAMPLE: A newspaper may print information obtained from a leak at the White House but attribute it to the wrong source. The information is printed as received (preserving data integrity), but its source is incorrect (corrupting origin integrity).

Integrity mechanisms fall into two classes: *prevention* mechanisms and *detection* mechanisms.

Prevention mechanisms seek to maintain the integrity of the data by blocking any unauthorized attempts to change the data or any attempts to change the data in unauthorized ways. The distinction between these two types of attempts is important. The former occurs when a user tries to change data which she has no authority to change. The latter occurs when a user authorized to make certain changes in the data tries to change the data in other ways. For example, suppose an accounting system is on a computer. Someone breaks into the system and tries to modify the accounting data. Then an unauthorized user has tried to violate the integrity of the accounting database. But if an accountant hired by the firm to maintain its books tries to embezzle money by sending it overseas and hiding the transactions, a user (the accountant) has tried to change data (the accounting data) in unauthorized ways (by moving it to a Swiss bank account). Adequate authentication and access controls will generally stop the break-in from the outside, but preventing the second type of attempt requires very different controls.

Detection mechanisms do not try to prevent violations of integrity; they simply report that the data's integrity is no longer trustworthy. Detection mechanisms may analyze system events (user or system actions) to detect problems or (more commonly) may analyze the data itself to see if required or expected constraints still hold. The mechanisms may report the actual cause of the integrity violation (a specific part of a file was altered), or they may simply report that the file is now corrupt.

Working with integrity is very different from working with confidentiality. With confidentiality, the data is either compromised or it is not, but integrity includes both the correctness and the trustworthiness of the data. The origin of the data (how and from whom it was obtained), how well the data was protected before it arrived at the current machine, and how well the data is protected on the current machine all affect the integrity of the data. Thus, evaluating integrity is often very difficult, because it relies on assumptions about the source of the data and about trust in that source—two underpinnings of security that are often overlooked.

1.1.3 Availability

Availability refers to the ability to use the information or resource desired. Availability is an important aspect of reliability as well as of system design because an unavailable system is at least as bad as no system at all. The aspect of availability that is relevant to security is that someone may deliberately arrange to deny access to data or to a service by making it unavailable. System designs usually assume a statistical model to analyze expected patterns of use, and mechanisms ensure availability when that statistical model holds. Someone may be able to manipulate use (or parameters that control use, such as network traffic) so that the assumptions of the statistical model are no longer valid. This means that the mechanisms for keeping the resource or data available are working in an environment for which they were not designed. As a result, they will often fail.

EXAMPLE: Suppose Anne has compromised a bank's secondary system server, which supplies bank account balances. When anyone else asks that server for information, Anne can supply any information she desires. Merchants validate checks by contacting the bank's primary balance server. If a merchant gets no response, the secondary server will be asked to supply the data. Anne's colleague prevents merchants from contacting the primary balance server, so all merchant queries go to the secondary server. Anne will never have a check turned down, regardless of her actual account balance. Notice that if the bank had only one server (the primary one), this scheme would not work. The merchant would be unable to validate the check.

Attempts to block availability, called *denial of service attacks*, can be the most difficult to detect, because the analyst must determine if the unusual access patterns are attributable to deliberate manipulation of resources or of environment. Complicating this determination is the nature of statistical models. Even if the model accurately describes the environment, atypical events simply contribute to the nature of the statistics. A deliberate attempt to make a resource unavailable may simply look like, or be, an atypical event. In some environments, it may not even appear atypical.

1.2 Threats

A *threat* is a potential violation of security. The violation need not actually occur for there to be a threat. The fact that the violation *might* occur means that those actions that could cause it to occur must be guarded against (or prepared for). Those actions are called *attacks*. Those who execute such actions, or cause them to be executed, are called *attackers*.

The three security services—confidentiality, integrity, and availability—counter threats to the security of a system. Shirey [916] divides threats into four broad classes: *disclosure*, or unauthorized access to information; *deception*, or acceptance of false data; *disruption*, or interruption or prevention of correct operation; and *usurpation*, or unauthorized control of some part of a system. These four broad classes encompass many common threats. Because the threats are ubiquitous, an introductory discussion of each one will present issues that recur throughout the study of computer security.

Snooping, the unauthorized interception of information, is a form of disclosure. It is passive, suggesting simply that some entity is listening to (or reading) communications or browsing through files or system information. *Wiretapping*, or *passive wiretapping*, is a form of snooping in which a network is monitored. (It is called "wiretapping" because of the "wires" that compose the network, although the term is used even if no physical wiring is involved.) Confidentiality services counter this threat.

Modification or *alteration*, an unauthorized change of information, covers three classes of threats. The goal may be deception, in which some entity relies on the modified data to determine which action to take, or in which incorrect information is accepted as correct and is released. If the modified data controls the operation of the system, the threats of disruption and usurpation arise. Unlike snooping, modification is active; it results from an entity changing information. *Active wiretapping* is a form of modification in which data moving across a network is altered; the term "active" distinguishes it from snooping ("passive" wiretapping). An example is the *man-in-the-middle* attack, in which an intruder reads messages from the sender and sends (possibly modified) versions to the recipient, in hopes that the recipient and sender will not realize the presence of the intermediary. Integrity services counter this threat.

Masquerading or *spoofing*, an impersonation of one entity by another, is a form of both deception and usurpation. It lures a victim into believing that the entity with which it is communicating is a different entity. For example, if a user tries to log into a computer across the Internet but instead reaches another computer that claims to be the desired one, the user has been spoofed. Similarly, if a user tries to read a file, but an attacker has arranged for the user to be given a different file, another spoof has taken place. This may be a passive attack (in which the user does not attempt to authenticate the recipient, but merely accesses it), but it is usually an active attack (in which the masquerader issues responses to mislead the user about its identity). Although primarily deception, it is often used to usurp control of a system by an attacker impersonating an authorized manager or controller. Integrity services (called "authentication services" in this context) counter this threat.

Some forms of masquerading may be allowed. *Delegation* occurs when one entity authorizes a second entity to perform functions on its behalf. The distinctions between delegation and masquerading are important. If Susan delegates to Thomas the authority to act on her behalf, she is giving permission for him to perform specific actions as though she were performing them herself. All parties are aware of the delegation. Thomas will not pretend to be Susan; rather, he will say, "I am Thomas combined site should be. The inconsistency often manifests itself as a security breach. For example, if proprietary documents were given to a university, the policy of confidentiality in the corporation would conflict with the more open policies of most universities. The university and the company must develop a mutual security policy that meets both their needs in order to produce a consistent policy. When the two sites communicate through an independent third party, such as an Internet Service Provider, the complexity of the situation grows rapidly.

1.3.1 Goals of Security

Given a security policy's specification of "secure" and "nonsecure" actions, these security mechanisms can prevent the attack, detect the attack, or recover from the attack. The strategies may be used together or separately.

Prevention means that an attack will fail. For example, if one attempts to break into a host over the Internet and that host is not connected to the Internet, the attack has been prevented. Typically, prevention involves implementation of mechanisms that users cannot override and that are trusted to be implemented in a correct, unalterable way, so that the attacker cannot defeat the mechanism by changing it. Preventative mechanisms often are very cumbersome and interfere with system use to the point that they hinder normal use of the system. But some simple preventative mechanisms, such as passwords (which aim to prevent unauthorized users from accessing the system), have become widely accepted. Prevention mechanisms can prevent compromise of parts of the system; once in place, the resource protected by the mechanism need not be monitored for security problems, at least in theory.

Detection is most useful when an attack cannot be prevented, but it can also indicate the effectiveness of preventative measures. Detection mechanisms accept that an attack will occur; the goal is to determine that an attack is underway, or has occurred, and report it. The attack may be monitored, however, to provide data about its nature, severity, and results. Typical detection mechanisms monitor various aspects of the system, looking for actions or information indicating an attack. A good example of such a mechanism is one that gives a warning when a user enters an incorrect password three times. The login may continue, but an error message in a system log reports the unusually high number of mistyped passwords. Detection mechanisms do not prevent compromise of parts of the system, which is a serious drawback. The resource protected by the detection mechanism is continuously or periodically monitored for security problems.

Recovery has two forms. The first is to stop an attack and to assess and repair any damage caused by that attack. As an example, if the attacker deletes a file, one recovery mechanism would be to restore the file from backup tapes. In practice, recovery is far more complex, because the nature of each attack is unique. Thus, the type and extent of any damage can be difficult to characterize completely. Moreover, the attacker may return, so recovery involves identification and fixing of the vulnerabilities used by the attacker to enter the system. In some cases, retaliation (by attacking the attacker's system or taking legal steps to hold the attacker accountable) is part of recovery. In all these cases, the system's functioning is inhibited by the attack. By definition, recovery requires resumption of correct operation.

In a second form of recovery, the system continues to function correctly while an attack is underway. This type of recovery is quite difficult to implement because of the complexity of computer systems. It draws on techniques of fault tolerance as well as techniques of security and is typically used in safety-critical systems. It differs from the first form of recovery, because at no point does the system function incorrectly. However, the system may disable nonessential functionality. Of course, this type of recovery is often implemented in a weaker form whereby the system detects incorrect functioning automatically and then corrects (or attempts to correct) the error.

1.4 Assumptions and Trust

How do we determine if the policy correctly describes the required level and type of security for the site? This question lies at the heart of all security, computer and otherwise. Security rests on assumptions specific to the type of security required and the environment in which it is to be employed.

EXAMPLE: Opening a door lock requires a key. The assumption is that the lock is secure against lock picking. This assumption is treated as an axiom and is made because most people would require a key to open a door lock. A good lock picker, however, can open a lock without a key. Hence, in an environment with a skilled, untrustworthy lock picker, the assumption is wrong and the consequence invalid.

If the lock picker is trustworthy, the assumption is valid. The term "trustworthy" implies that the lock picker will not pick a lock unless the owner of the lock authorizes the lock picking. This is another example of the role of trust. A well-defined exception to the rules provides a "back door" through which the security mechanism (the locks) can be bypassed. The trust resides in the belief that this back door will not be used except as specified by the policy. If it is used, the trust has been misplaced and the security mechanism (the lock) provides no security.

Like the lock example, a policy consists of a set of axioms that the policy makers believe can be enforced. Designers of policies always make two assumptions. First, the policy correctly and unambiguously partitions the set of system states into "secure" and "nonsecure" states. Second, the security mechanisms prevent the system from entering a "nonsecure" state. If either assumption is erroneous, the system will be nonsecure.

These two assumptions are fundamentally different. The first assumption asserts that the policy is a correct description of what constitutes a "secure" system. For example, a bank's policy may state that officers of the bank are authorized to shift money among accounts. If a bank officer puts $100,000 in his account, has the bank's security been violated? Given the aforementioned policy statement, no, because the officer was authorized to move the money. In the "real world," that action would constitute embezzlement, something any bank would consider a security violation.

The second assumption says that the security policy can be enforced by security mechanisms. These mechanisms are either *secure*, *precise*, or *broad*. Let P be the set of all possible states. Let Q be the set of secure states (as specified by the security policy). Let the security mechanisms restrict the system to some set of states R (thus, $R \subseteq P$). Then we have the following definition.

Definition 1–3. A security mechanism is *secure* if $R \subseteq Q$; it is *precise* if $R = Q$; and it is *broad* if there are states r such that $r \in R$ and $r \notin Q$.

Ideally, the union of all security mechanisms active on a system would produce a single precise mechanism (that is, $R = Q$). In practice, security mechanisms are broad; they allow the system to enter nonsecure states. We will revisit this topic when we explore policy formulation in more detail.

Trusting that mechanisms work requires several assumptions.

1. Each mechanism is designed to implement one or more parts of the security policy.
2. The union of the mechanisms implements all aspects of the security policy.
3. The mechanisms are implemented correctly.
4. The mechanisms are installed and administered correctly.

Because of the importance and complexity of trust and of assumptions, we will revisit this topic repeatedly and in various guises throughout this book.

1.5 Assurance

Trust cannot be quantified precisely. System specification, design, and implementation can provide a basis for determining "how much" to trust a system. This aspect of trust is called *assurance*. It is an attempt to provide a basis for bolstering (or substantiating or specifying) how much one can trust a system.

EXAMPLE: In the United States, aspirin from a nationally known and reputable manufacturer, delivered to the drugstore in a safety-sealed container, and sold with the seal still in place, is considered trustworthy by most people. The bases for that trust are as follows.

- The testing and certification of the drug (aspirin) by the Food and Drug Administration. The FDA has jurisdiction over many types of medicines and allows medicines to be marketed only if they meet certain clinical standards of usefulness.

- The manufacturing standards of the company and the precautions it takes to ensure that the drug is not contaminated. National and state regulatory commissions and groups ensure that the manufacture of the drug meets specific acceptable standards.
- The safety seal on the bottle. To insert dangerous chemicals into a safety-sealed bottle without damaging the seal is very difficult.

The three technologies (certification, manufacturing standards, and preventative sealing) provide some degree of assurance that the aspirin is not contaminated. The degree of trust the purchaser has in the purity of the aspirin is a result of these three processes.

In the 1980s, drug manufacturers met two of the criteria above, but none used safety seals.[1] A series of "drug scares" arose when a well-known manufacturer's medicines were contaminated after manufacture but before purchase. The manufacturer promptly introduced safety seals to assure its customers that the medicine in the container was the same as when it was shipped from the manufacturing plants.

Assurance in the computer world is similar. It requires specific steps to ensure that the computer will function properly. The sequence of steps includes detailed specifications of the desired (or undesirable) behavior; an analysis of the design of the hardware, software, and other components to show that the system will not violate the specifications; and arguments or proofs that the implementation, operating procedures, and maintenance procedures will produce the desired behavior.

Definition 1–4. A system is said to *satisfy* a specification if the specification correctly states how the system will function.

This definition also applies to design and implementation satisfying a specification.

1.5.1 Specification

A *specification* is a (formal or informal) statement of the desired functioning of the system. It can be highly mathematical, using any of several languages defined for that purpose. It can also be informal, using, for example, English to describe what the system should do under certain conditions. The specification can be low-level, combining program code with logical and temporal relationships to specify ordering of events. The defining quality is a statement of what the system is allowed to do or what it is not allowed to do.

[1] Many used childproof caps, but they prevented only young children (and some adults) from opening the bottles. They were not designed to protect the medicine from malicious adults.

EXAMPLE: A company is purchasing a new computer for internal use. They need to trust the system to be invulnerable to attack over the Internet. One of their (English) specifications would read "The system cannot be attacked over the Internet."

Specifications are used not merely in security but also in systems designed for safety, such as medical technology. They constrain such systems from performing acts that could cause harm. A system that regulates traffic lights must ensure that pairs of lights facing the same way turn red, green, and yellow at the same time and that at most one set of lights facing cross streets at an intersection is green.

A major part of the derivation of specifications is determination of the set of requirements relevant to the system's planned use. Section 1.6 discusses the relationship of requirements to security.

1.5.2 Design

The *design* of a system translates the specifications into components that will implement them. The design is said to *satisfy* the specifications if, under all relevant circumstances, the design will not permit the system to violate those specifications.

EXAMPLE: A design of the computer system for the company mentioned above had no network interface cards, no modem cards, and no network drivers in the kernel. This design satisfied the specification because the system would not connect to the Internet. Hence it could not be attacked over the Internet.

An analyst can determine whether a design satisfies a set of specifications in several ways. If the specifications and designs are expressed in terms of mathematics, the analyst must show that the design formulations are consistent with the specifications. Although much of the work can be done mechanically, a human must still perform some analyses and modify components of the design that violate specifications (or, in some cases, components that cannot be shown to satisfy the specifications). If the specifications and design do not use mathematics, then a convincing and compelling argument should be made. Most often, the specifications are nebulous and the arguments are half-hearted and unconvincing or provide only partial coverage. The design depends on assumptions about what the specifications mean. This leads to vulnerabilities, as we will see.

1.5.3 Implementation

Given a design, the implementation creates a system that satisfies that design. If the design also satisfies the specifications, then by transitivity the implementation will also satisfy the specifications.

The difficulty at this step is the complexity of proving that a program correctly implements the design and, in turn, the specifications.

for confidentiality of the salaries in the database. The officers of the company must decide the financial cost to the company should the salaries be disclosed, including potential loss from lawsuits (if any); changes in policies, procedures, and personnel; and the effect on future business. These are all business-related judgments, and determining their value is part of what company officers are paid to do.

Overlapping benefits are also a consideration. Suppose the integrity protection mechanism can be augmented very quickly and cheaply to provide confidentiality. Then the cost of providing confidentiality is much lower. This shows that evaluating the cost of a particular security service depends on the mechanism chosen to implement it and on the mechanisms chosen to implement other security services. The cost-benefit analysis should take into account as many mechanisms as possible. Adding security mechanisms to an existing system is often more expensive (and, incidentally, less effective) than designing them into the system in the first place.

1.6.2 Risk Analysis

To determine whether an asset should be protected, and to what level, requires analysis of the potential threats against that asset and the likelihood that they will materialize. The level of protection is a function of the probability of an attack occurring and the effects of the attack should it succeed. If an attack is unlikely, protecting against it has a lower priority than protecting against a likely one. If the unlikely attack would cause long delays in the company's production of widgets but the likely attack would be only a nuisance, then more effort should be put into preventing the unlikely attack. The situations between these extreme cases are far more subjective.

Let's revisit our company with the salary database that transmits salary information over a network to a second computer that prints employees' checks. The data is stored on the database system and then moved over the network to the second system. Hence, the risk of unauthorized changes in the data occurs in three places: on the database system, on the network, and on the printing system. If the network is a local (company-wide) one and no wide area networks are accessible, the threat of attackers entering the systems is confined to untrustworthy internal personnel. If, however, the network is connected to the Internet, the risk of geographically distant attackers attempting to intrude is substantial enough to warrant consideration.

This example illustrates some finer points of risk analysis. First, risk is a function of environment. Attackers from a foreign country are not a threat to the company when the computer is not connected to the Internet. If foreign attackers wanted to break into the system, they would need physically to enter the company (and would cease to be "foreign" because they would then be "local"). But if the computer is connected to the Internet, foreign attackers become a threat because they can attack over the Internet. An additional, less tangible issue is the faith in the company. If the company is not able to meet its payroll because it does not know *whom* it is to pay, the company will lose the faith of its employees. It may be unable to hire anyone, because the people hired would not be sure they would get paid. Investors would not fund the company because of the likelihood of lawsuits by unpaid employees. The risk arises from the environments in which the company functions.

Second, the risks change with time. If a company's network is not connected to the Internet, there seems to be no risk of attacks from other hosts on the Internet. However, despite any policies to the contrary, someone could connect a modem to one of the company computers and connect to the Internet through the modem. Should this happen, any risk analysis predicated on isolation from the Internet would no longer be accurate. Although policies can forbid the connection of such a modem and procedures can be put in place to make such connection difficult, unless the responsible parties can guarantee that no such modem will ever be installed, the risks can change.

Third, many risks are quite remote but still exist. In the modem example, the company has sought to minimize the risk of an Internet connection. Hence, this risk is "acceptable" but not nonexistent. As a practical matter, one does not worry about acceptable risks; instead, one worries that the risk will become unacceptable.

Finally, the problem of "analysis paralysis" refers to making risk analyses with no effort to act on those analyses. To change the example slightly, suppose the company performs a risk analysis. The executives decide that they are not sure if all risks have been found, so they order a second study to verify the first. They reconcile the studies then wait for some time to act on these analyses. At that point, the security officers raise the objection that the conditions in the workplace are no longer those that held when the original risk analyses were done. The analysis is repeated. But the company cannot decide how to ameliorate the risks, so it waits until a plan of action can be developed, and the process continues. The point is that the company is paralyzed and cannot act on the risks it faces.

1.6.3 Laws and Customs

Laws restrict the availability and use of technology and affect procedural controls. Hence, any policy and any selection of mechanisms must take into account legal considerations.

EXAMPLE: Until the year 2000, the United States controlled the export of cryptographic hardware and software (considered munitions under United States law). If a U.S. software company worked with a computer manufacturer in London, the U.S. company could not send cryptographic software to the manufacturer. The U.S. company first would have to obtain a license to export the software from the United States. Any security policy that depended on the London manufacturer using that cryptographic software would need to take this into account.

EXAMPLE: Suppose the law makes it illegal to read a user's file without the user's permission. An attacker breaks into the system and begins to download users' files. If the system administrators notice this and observe what the attacker is reading, they will be reading the victim's files without his permission and therefore will be violating the law themselves. For this reason, most sites require users to give (implicit or explicit) permission for system administrators to read their files. In some jurisdictions, an explicit exception allows system administrators to access information on their systems without permission in order to protect the quality of service provided or to prevent damage to their systems.

Complicating this issue are situations involving the laws of multiple jurisdictions—especially foreign ones.

EXAMPLE: In the 1990s, the laws involving the use of cryptography in France were very different from those in the United States. The laws of France required companies sending enciphered data out of the country to register their cryptographic keys with the government. Security procedures involving the transmission of enciphered data from a company in the United States to a branch office in France had to take these differences into account.

EXAMPLE: If a policy called for prosecution of attackers and intruders came from Russia to a system in the United States, prosecution would involve asking the United States authorities to extradite the alleged attackers from Russia. This undoubtedly would involve court testimony from company personnel involved in handling the intrusion, possibly trips to Russia, and more court time once the extradition was completed. The cost of prosecuting the attackers might be considerably higher than the company would be willing (or able) to pay.

Laws are not the only constraints on policies and selection of mechanisms. Society distinguishes between *legal* and *acceptable* practices. It may be legal for a company to require all its employees to provide DNA samples for authentication purposes, but it is not socially acceptable. Requiring the use of social security numbers as passwords is legal (unless the computer is one owned by the U.S. government) but also unacceptable. These practices provide security but at an unacceptable cost, and they encourage users to evade or otherwise overcome the security mechanisms.

The issue that laws and customs raise is the issue of psychological acceptability. A security mechanism that would put users and administrators at legal risk would place a burden on these people that few would be willing to bear; thus, such a mechanism would not be used. An unused mechanism is worse than a nonexistent one, because it gives a false impression that a security service is available. Hence, users may rely on that service to protect their data, when in reality their data is unprotected.

1.7 Human Issues

Implementing computer security controls is complex, and in a large organization procedural controls often become vague or cumbersome. Regardless of the strength of the technical controls, if nontechnical considerations affect their implementation and use, the effect on security can be severe. Moreover, if configured or used incorrectly, even the best security control is useless at best and dangerous at worst. Thus, the designers, implementers, and maintainers of security controls are essential to the correct operation of those controls.

1.7.1 Organizational Problems

Security provides no direct financial rewards to the user. It limits losses, but it also requires the expenditure of resources that could be used elsewhere. Unless losses occur, organizations often believe they are wasting effort related to security. After a loss, the value of these controls suddenly becomes appreciated. Furthermore, security controls often add complexity to otherwise simple operations. For example, if concluding a stock trade takes two minutes without security controls and three minutes with security controls, adding those controls results in a 50% loss of productivity.

Losses occur when security protections are in place, but such losses are expected to be less than they would have been without the security mechanisms. The key question is whether such a loss, combined with the resulting loss in productivity, would be greater than a financial loss or loss of confidence should one of the nonsecured transactions suffer a breach of security.

Compounding this problem is the question of who is responsible for the security of the company's computers. The power to implement appropriate controls must reside with those who are responsible; the consequence of not doing so is that the people who can most clearly see the need for security measures, and who are responsible for implementing them, will be unable to do so. This is simply sound business practice; responsibility without power causes problems in any organization, just as does power without responsibility.

Once clear chains of responsibility and power have been established, the need for security can compete on an equal footing with other needs of the organization. The most common problem a security manager faces is the lack of people trained in the area of computer security. Another common problem is that knowledgeable people are overloaded with work. At many organizations, the "security administrator" is also involved in system administration, development, or some other secondary function. In fact, the security aspect of the job is often secondary. The problem is that indications of security problems often are not obvious and require time and skill to spot. Preparation for an attack makes dealing with it less chaotic, but such preparation takes enough time and requires enough attention so that treating it as a secondary aspect of a job means that it will not be performed well, with the expected consequences.

Lack of resources is another common problem. Securing a system requires resources as well as people. It requires time to design a configuration that will provide an adequate level of security, to implement the configuration, and to administer the system. It requires money to purchase products that are needed to build an adequate security system or to pay someone else to design and implement security measures. It requires computer resources to implement and execute the security mechanisms and procedures. It requires training to ensure that employees understand how to use the security tools, how to interpret the results, and how to implement the nontechnical aspects of the security policy.

1.7.2 People Problems

The heart of any security system is people. This is particularly true in computer security, which deals mainly with technological controls that can usually be bypassed by human intervention. For example, a computer system authenticates a user by asking that user for a secret code; if the correct secret code is supplied, the computer assumes that the user is authorized to use the system. If an authorized user tells another person his secret code, the unauthorized user can masquerade as the authorized user with significantly less likelihood of detection.

People who have some motive to attack an organization and are not authorized to use that organization's systems are called *outsiders* and can pose a serious threat. Experts agree, however, that a far more dangerous threat comes from disgruntled employees and other *insiders* who are authorized to use the computers. Insiders typically know the organization of the company's systems and what procedures the operators and users follow and often know enough passwords to bypass many security controls that would detect an attack launched by an outsider. Insider *misuse* of authorized privileges is a very difficult problem to solve.

Untrained personnel also pose a threat to system security. As an example, one operator did not realize that the contents of backup tapes needed to be verified before the tapes were stored. When attackers deleted several critical system files, she discovered that none of the backup tapes could be read.

System administrators who misread the output of security mechanisms, or do not analyze that output, contribute to the probability of successful attacks against their systems. Similarly, administrators who misconfigure security-related features of a system can weaken the site security. Users can also weaken site security by misusing security mechanisms (such as selecting passwords that are easy to guess).

Lack of training need not be in the technical arena. Many successful break-ins have arisen from the art of *social engineering*. If operators will change passwords based on telephone requests, all an attacker needs to do is to determine the name of someone who uses the computer. A common tactic is to pick someone fairly far above the operator (such as a vice president of the company) and to feign an emergency (such as calling at night and saying that a report to the president of the company is due the next morning) so that the operator will be reluctant to refuse the request. Once the password has been changed to one that the attacker knows, he can simply log in as a normal user. Social engineering attacks are remarkably successful and often devastating.

The problem of misconfiguration is aggravated by the complexity of many security-related configuration files. For instance, a typographical error can disable key protection features. Even worse, software does not always work as advertised.

This chapter has laid the foundation for what follows. All aspects of computer security begin with the nature of threats and countering security services. In future chapters, we will build on these basic concepts.

1.10 Research Issues

Future chapters will explore research issues in the technical realm. However, other, nontechnical issues affect the needs and requirements for technical solutions, and research into these issues helps guide research into technical areas.

A key question is how to quantify risk. The research issue is how to determine the effects of a system's vulnerabilities on its security. For example, if a system can be compromised in any of 50 ways, how can a company compare the costs of the procedures (technical and otherwise) needed to prevent the compromises with the costs of detecting the compromises, countering them, and recovering from them? Many methods assign weights to the various factors, but these methods are *ad hoc*. A rigorous technique for determining appropriate weights has yet to be found.

The relationships of computer security to the political, social, and economic aspects of the world are not well understood. How does the ubiquity of the Internet change a country's borders? If someone starts at a computer in France, transits networks that cross Switzerland, Germany, Poland, Norway, Sweden, and Finland, and launches an attack on a computer in Russia, who has jurisdiction? How can a country limit the economic damage caused by an attack on its computer networks? How can attacks be traced to their human origins?

This chapter has also raised many technical questions. Research issues arising from them will be explored in future chapters.

1.11 Further Reading

Risk analysis arises in a variety of contexts. Molak [725] presents essays on risk management and analysis in a variety of fields. Laudan [610] provides an enjoyable introduction to the subject. Neumann [772] discusses the risks of technology and recent problems. Software safety (Leveson [622]) requires an understanding of the risks posed in the environment. Peterson [804] discusses many programming errors in a readable way. All provide insights into the problems that arise in a variety of environments.

Many authors recount stories of security incidents. The earliest, Parker's wonderful book [799], discusses motives and personalities as well as technical details. Stoll recounts the technical details of uncovering an espionage ring that began as the result of a 75¢ accounting error [973, 975]. Hafner and Markoff describe the same episode in a study of "cyberpunks" [432]. The Internet worm [322, 432, 845, 953] brought the problem of computer security into popular view. Numerous other incidents [374, 432, 642, 914, 931, 968] have heightened public awareness of the problem.

Several books [59, 61, 824, 891] discuss computer security for the layperson. These works tend to focus on attacks that are visible or affect the end user (such as pornography, theft of credit card information, and deception). They are worth reading for those who wish to understand the results of failures in computer security.

1.12 Exercises

1. Classify each of the following as a violation of confidentiality, of integrity, of availability, or of some combination thereof.
    a. John copies Mary's homework.
    b. Paul crashes Linda's system.
    c. Carol changes the amount of Angelo's check from $100 to $1,000.
    d. Gina forges Roger's signature on a deed.
    e. Rhonda registers the domain name "AddisonWesley.com" and refuses to let the publishing house buy or use that domain name.
    f. Jonah obtains Peter's credit card number and has the credit card company cancel the card and replace it with another card bearing a different account number.
    g. Henry spoofs Julie's IP address to gain access to her computer.
2. Identify mechanisms for implementing the following. State what policy or policies they might be enforcing.
    a. A password changing program will reject passwords that are less than five characters long or that are found in the dictionary.
    b. Only students in a computer science class will be given accounts on the department's computer system.
    c. The login program will disallow logins of any students who enter their passwords incorrectly three times.
    d. The permissions of the file containing Carol's homework will prevent Robert from cheating and copying it.
    e. When World Wide Web traffic climbs to more than 80% of the network's capacity, systems will disallow any further communications to or from Web servers.
    f. Annie, a systems analyst, will be able to detect a student using a program to scan her system for vulnerabilities.

   g. A program used to submit homework will turn itself off just after the due date.

3. The aphorism "security through obscurity" suggests that hiding information provides some level of security. Give an example of a situation in which hiding information does not add appreciably to the security of a system. Then give an example of a situation in which it does.
4. Give an example of a situation in which a compromise of confidentiality leads to a compromise in integrity.
5. Show that the three security services—confidentiality, integrity, and availability—are sufficient to deal with the threats of disclosure, disruption, deception, and usurpation.
6. In addition to mathematical and informal statements of policy, policies can be implicit (not stated). Why might this be done? Might it occur with informally stated policies? What problems can this cause?
7. For each of the following statements, give an example of a situation in which the statement is true.
   a. Prevention is more important than detection and recovery.
   b. Detection is more important than prevention and recovery.
   c. Recovery is more important than prevention and detection.
8. Is it possible to design and implement a system in which *no* assumptions about trust are made? Why or why not?
9. Policy restricts the use of electronic mail on a particular system to faculty and staff. Students cannot send or receive electronic mail on that host. Classify the following mechanisms as secure, precise, or broad.
   a. The electronic mail sending and receiving programs are disabled.
   b. As each letter is sent or received, the system looks up the sender (or recipient) in a database. If that party is listed as faculty or staff, the mail is processed. Otherwise, it is rejected. (Assume that the database entries are correct.)
   c. The electronic mail sending programs ask the user if he or she is a student. If so, the mail is refused. The electronic mail receiving programs are disabled.
10. Consider a very high-assurance system developed for the military. The system has a set of specifications, and both the design and implementation have been proven to satisfy the specifications. What questions should school administrators ask when deciding whether to purchase such a system for their school's use?
11. How do laws protecting privacy impact the ability of system administrators to monitor user activity?

12. Computer viruses are programs that, among other actions, can delete files without a user's permission. A U.S. legislator wrote a law banning the deletion of any files from computer disks. What was the problem with this law from a computer security point of view? Specifically, state which security service would have been affected if the law had been passed.
13. Users often bring in programs or download programs from the Internet. Give an example of a site for which the benefits of allowing users to do this outweigh the dangers. Then give an example of a site for which the dangers of allowing users to do this outweigh the benefits.
14. A respected computer scientist has said that no computer can ever be made perfectly secure. Why might she have said this?
15. An organization makes each lead system administrator responsible for the security of the system he or she runs. However, the management determines what programs are to be on the system and how they are to be configured.
    a. Describe the security problem(s) that this division of power would create.
    b. How would you fix them?
16. The president of a large software development company has become concerned about competitors learning proprietary information. He is determined to stop them. Part of his security mechanism is to require all employees to report any contact with employees of the company's competitors, even if it is purely social. Do you believe this will have the desired effect? Why or why not?
17. The police and the public defender share a computer. What security problems does this present? Do you feel it is a reasonable cost-saving measure to have all public agencies share the same (set of) computers?
18. Companies usually restrict the use of electronic mail to company business but do allow minimal use for personal reasons.
    a. How might a company detect excessive personal use of electronic mail, other than by reading it? (*Hint*: Think about the personal use of a company telephone.)
    b. Intuitively, it seems reasonable to ban *all* personal use of electronic mail on company computers. Explain why most companies do not do this.
19. Argue for or against the following proposition. Ciphers that the government cannot cryptanalyze should be outlawed. How would your argument change if such ciphers could be used provided that the users registered the keys with the government?
20. For many years, industries and financial institutions hired people who broke into their systems once those people were released from prison. Now, such a conviction tends to prevent such people from being hired.

Why you think attitudes on this issue changed? Do you think they changed for the better or for the worse?

21. A graduate student accidentally releases a program that spreads from computer system to computer system. It deletes no files but requires much time to implement the necessary defenses. The graduate student is convicted. Despite demands that he be sent to prison for the maximum time possible (to make an example of him), the judge sentences him to pay a fine and perform community service. What factors do you believe caused the judge to hand down the sentence he did? What would you have done were you the judge, and what extra information would you have needed to make your decision?

Chapter 2
Access Control Matrix

GRANDPRÉ: Description cannot suit itself in words
To demonstrate the life of such a battle
In life so lifeless as it shows itself.
—*The Life of Henry the Fifth*, IV, ii, 53–55.

A *protection system* describes the conditions under which a system is secure. In this chapter, we present a classical formulation of a protection system. The *access control matrix model* arose both in operating systems research and in database research; it describes allowed accesses using a matrix.

2.1 Protection State

The *state* of a system is the collection of the current values of all memory locations, all secondary storage, and all registers and other components of the system. The subset of this collection that deals with protection is the *protection state* of the system. An *access control matrix* is one tool that can describe the current protection state.

Consider the set of possible protection states P. Some subset Q of P consists of exactly those states in which the system is authorized to reside. So, whenever the system state is in Q, the system is secure. When the current state is in $P - Q$[1], the system is not secure. Our interest in representing the state is to characterize those states in Q, and our interest in enforcing security is to ensure that the system state is always an element of Q. Characterizing the states in Q is the function of a *security policy*; preventing the system from entering a state in $P - Q$ is the function of a *security mechanism*. Recall from Definition 1–3 that a mechanism that enforces this restriction is *precise*.

The *access control matrix model* is the most precise model used to describe a protection state. It characterizes the rights of each *subject* (active entity, such as a process) with respect to every other entity. The description of elements of A form a *specification* against which the current state can be compared. Specifications

[1] The notation $P - Q$ means all elements of set P not in set Q.

take many forms, and different specification languages have been created to describe the characteristics of allowable states.

As the system changes, the protection state changes. When a command changes the state of the system, a *state transition* occurs. Very often, constraints on the set of allowed states use these transitions inductively; a set of authorized states is defined, and then a set of operations is allowed on the elements of that set. The result of transforming an authorized state with an operation allowed in that state is an authorized state. By induction, the system will always be in an authorized state. Hence, both states and state transitions are often constrained.

In practice, *any* operation on a real system causes multiple state transitions; the reading, loading, altering, and execution of any datum or instruction causes a transition. We are concerned only with those state transitions that affect the protection state of the system, so only transitions that alter the actions a subject is authorized to take are relevant. For example, a program that changes a variable to 0 does not (usually) alter the protection state. However, if the variable altered is one that affects the privileges of a process, then the program does alter the protection state and needs to be accounted for in the set of transitions.

2.2 Access Control Matrix Model

The simplest framework for describing a protection system is the *access control matrix model*, which describes the rights of users over files in a matrix. Butler Lampson first proposed this model in 1971 [608]; Graham and Denning [279, 413] refined it, and we will use their version.

The set of all protected entities (that is, entities that are relevant to the protection state of the system) is called the set of *objects* O. The set of *subjects* S is the set of active objects, such as processes and users. In the access control matrix model, the relationship between these entities is captured by a matrix A with *rights* drawn from a set of rights R in each entry $a[s, o]$, where $s \in S$, $o \in O$, and $a[s, o] \subseteq R$. The subject s has the set of rights $a[s, o]$ over the object o. The set of protection *states* of the system is represented by the triple (S, O, A). For example, Figure 2–1 shows the protection state of a system. Here, process 1 can read or write file 1 and can read file 2; process 2 can append to file 1 and read file 2. Process 1 can communicate with process 2 by writing to it, and process 2 can read from process 1. Each process owns itself and the file with the same number. Note that the processes themselves are treated as both subjects (rows) and objects (columns). This enables a process to be the target of operations as well as the operator.

Interpretation of the meaning of these rights varies from system to system. Reading from, writing to, and appending to files is usually clear enough, but what does "reading from" a process mean? Depending on the instantiation of the model, it could mean that the reader accepts messages from the process being read, or it could mean that the reader simply looks at the state of the process being read (as a debugger does,

	file 1	file 2	process 1	process 2
process 1	read, write, own	read	read, write, execute, own	write
process 2	append	read, own	read	read, write, execute, own

Figure 2–1 An access control matrix. The system has two processes and two files. The set of rights is {read, write, execute, append, own}.

for example). The meaning of the right may vary depending on the object involved. The point is that the access control matrix model is an *abstract* model of the protection state, and when one talks about the meaning of some particular access control matrix, one must always talk with respect to a particular implementation or system.

The *own* right is a distinguished right. In most systems, the creator of an object has special privileges: the ability to add and delete rights for other users (and for the owner). In the system shown in Figure 2–1, for example, process 1 could alter the contents of $A[x, \text{file } 1]$, where x is any subject.

EXAMPLE: The UNIX system defines the rights "read," "write," and "execute." When a process accesses a file, these terms mean what one would expect. When a process accesses a directory, "read" means to be able to list the contents of the directory; "write" means to be able to create, rename, or delete files or subdirectories in that directory; and "execute" means to be able to access files or subdirectories in that directory. When a process accesses another process, "read" means to be able to receive signals, "write" means to be able to send signals, and "execute" means to be able to execute the process as a subprocess.

Moreover, the superuser can access any (local) file regardless of the permissions the owner has granted. In effect, the superuser "owns" all objects on the system. Even in this case however, the interpretation of the rights is constrained. For example, the superuser cannot alter a directory using the system calls and commands that alter files. The superuser must use specific system calls and commands to create, rename, and delete files.

Although the "objects" involved in the access control matrix are normally thought of as files, devices, and processes, they could just as easily be messages sent between processes, or indeed systems themselves. Figure 2–2 shows an example access control matrix for three systems on a local area network (LAN). The rights correspond to various network protocols: *own* (the ability to add servers), *ftp* (the ability to access the system using the File Transfer Protocol, or FTP [815]), *nfs* (the ability to access file systems using the Network File System, or NFS, protocol [166, 981]), and *mail* (the ability to send and receive mail using the Simple Mail Transfer

host names	*telegraph*	*nob*	*toadflax*
telegraph	own	ftp	ftp
nob		ftp, nfs, mail, own	ftp, nfs, mail
toadflax		ftp, mail	ftp, nfs, mail, own

Figure 2–2 Rights on a LAN. The set of rights is {ftp, mail, nfs, own}.

Protocol, or SMTP [814]). The subject *telegraph* is a personal computer with an *ftp* client but no servers, so neither of the other systems can access it, but it can *ftp* to them. The subject *nob* is configured to provide NFS service to a set of clients that does not include the host toadflax, and both systems will exchange mail with any host and allow any host to use *ftp*.

At the micro level, access control matrices can model programming language accesses; in this case, the objects are the variables and the subjects are the procedures (or modules). Consider a program in which events must be synchronized. A module provides functions for incrementing (*inc_ctr*) and decrementing (*dec_ctr*) a counter private to that module. The routine *manager* calls these functions. The access control matrix is shown in Figure 2–3. Note that "+" and "–" are the rights, representing the ability to add and subtract, respectively, and *call* is the ability to invoke a procedure. The routine *manager* can call itself; presumably, it is recursive.

In the examples above, entries in the access control matrix are rights. However, they could as easily have been functions that determined the set of rights at any particular state based on other data, such as a history of prior accesses, the time of day, the rights another subject has over the object, and so forth. A common form of such a function is a locking function used to enforce the Bernstein conditions,[2] so when a process is writing to a file, other processes cannot access the file; but once the writing is done, the processes can access the file once again.

	counter	*inc_ctr*	*dec_ctr*	*manager*
inc_ctr	+			
dec_ctr	–			
manager		call	call	call

Figure 2–3 Rights in a program. The set of rights is {+, –, call}.

[2] The Bernstein conditions ensure that data is consistent. They state that any number of readers may access a datum simultaneously, but if a writer is accessing the datum, no other writers or any reader can access the datum until the current writing is complete [805].

2.2.1 Access Control by Boolean Expression Evaluation

Miller and Baldwin [715] use an access control matrix to control access to fields in a database. The values are determined by Boolean expressions. Their objects are records and fields; the subjects are users authorized to access the databases. Types of access are defined by the database and are called *verbs*; for example, the Structured Query Language (SQL) would have the verbs Insert and Update. Each *rule*, corresponding to a function, is associated with one or more verbs. Whenever a subject attempts to access an object using a right (verb) *r*, the Boolean expression (rule) associated with *r* is evaluated; if it is true, access is allowed, but if it is false, access is not allowed.

The Access Restriction Facility (ARF) program exemplifies this approach. It defines subjects as having attributes such as a name, a level, a role, membership in groups, and access to programs, but the user can assign any meaning desired to any attribute. For example:

name	**role**	**groups**	**programs**
matt	programmer	sys, hack	compilers, editors
holly	artist	user, creative	editors, paint, draw
heidi	chef, gardener	acct, creative	editors, kitchen

Verbs have a default rule, either "closed" (access denied unless explicitly granted; represented by the 0 rule) or "open" (access granted unless explicitly denied; represented by the 1 rule):

verb	**default rule**
read	1
write	0
paint	0
temp_ctl	0

Associated with each object is a set of verbs, and each (object, verb) pair has an associated rule:

name	**rules**
recipes	write: 'creative' in subject.group
overpass	write: 'artist' in subject.role or 'gardener' in subject.role
.shellrct	write: 'hack' in subject.group and time.hour < 4 and time.hour > 0
oven.dev	read: 0; temp_ctl: 'kitchen' in subject.programs and 'chef' in subject.role

commands, or *transformation procedures*, that update the access control matrix. The commands state which entry in the matrix is to be changed, and how; hence, the commands require parameters. Formally, let c_k be the kth command with formal parameters $p_{k,1}, ..., p_{k,m}$. Then the ith transition would be written as

$$X_i \vdash_{c_{i+1}(p_{i+1,1}, ..., p_{i+1,m})} X_{i+1}.$$

Note the similarity in notation between the use of the command and the state transition operations. This is deliberate. For every command, there is a sequence of state transition operations that takes the initial state X_i to the resulting state X_{i+1}. Using the command notation allows us to shorten the description of the transformation as well as list the parameters (subjects, objects, and entries) that affect the transformation operations.

We now focus on the commands themselves. Following Harrison, Ruzzo, and Ullman [450], we define a set of *primitive commands* that alter the access control matrix. In the following list, the protection state is (S, O, A) before the execution of each command and (S', O', A') after each command. The preconditions state the conditions needed for the primitive command to be executed, and the postconditions state the results.

1. Precondition: $s \notin S$
   Primitive command: **create subject** s
   Postconditions: $S' = S \cup \{ s \}, O' = O \cup \{ s \}$,
   $(\forall y \in O')[a'[s, y] = \varnothing], (\forall x \in S')[a'[x, s] = \varnothing]$,
   $(\forall x \in S)(\forall y \in O)[a'[x, y] = a[x, y]]$

   This primitive command creates a new subject s. Note that s must not exist as a subject *or an object* before this command is executed. This operation does not add any rights. It merely modifies the matrix.
2. Precondition: $o \notin O$
   Primitive command: **create object** o
   Postconditions: $S' = S, O' = O \cup \{ o \}$,
   $(\forall x \in S')[a'[x, o] = \varnothing], (\forall x \in S')(\forall y \in O)[a'[x, y] = a[x, y]]$

   This primitive command creates a new object o. Note that o must not exist before this command is executed. Like **create subject**, this operation does not add any rights. It merely modifies the matrix.
3. Precondition: $s \in S, o \in O$
   Primitive command: **enter** r **into** $a[s, o]$
   Postconditions: $S' = S, O' = O, a'[s, o] = a[s, o] \cup \{ r \}$,
   $(\forall x \in S')(\forall y \in O')[(x, y) \neq (s, o) \rightarrow a'[x, y] = a[x, y]]$

   This primitive command adds the right r to the cell $a[s, o]$. Note that $a[s, o]$ may already contain the right, in which case the effect of this primitive depends on the instantiation of the model (it may add another copy of the right or may do nothing).

4. Precondition: $s \in S, o \in O$
   Primitive command: **delete** r **from** $a[s, o]$
   Postconditions: $S' = S, O' = O, a'[s, o] = a[s, o] - \{ r \}$,
   $(\forall x \in S')(\forall y \in O')[(x, y) \neq (s, o) \rightarrow a'[x, y] = a[x, y]]$
   This primitive command deletes the right r from the cell $a[s, o]$. Note that $a[s, o]$ need not contain the right, in which case this operation has no effect.
5. Precondition: $s \in S$
   Primitive command: **destroy subject** s
   Postconditions: $S' = S - \{ s \}, O' = O - \{ o \}$,
   $(\forall y \in O')[a'[s, y] = \varnothing]$, $(\forall x \in S')[a'[x, s] = \varnothing]$,
   $(\forall x \in S')(\forall y \in O')[a'[x, y] = a[x, y]]$
   This primitive command deletes the subject s. The column and row for s in A are deleted also.
6. Precondition: $o \in O$
   Primitive command: **destroy object** o
   Postconditions: $S' = S, O' = O - \{ o \}$,
   $(\forall x \in S')[a'[x, o] = \varnothing]$, $(\forall x \in S')(\forall y \in O')[a'[x, y] = a[x, y]]$
   This primitive command deletes the object o. The column for o in A is deleted also.

These primitive operations can be combined into commands, during which multiple primitive operations may be executed.

EXAMPLE: In the UNIX system, if process p created a file f with owner read (r) and write (w) permission, the command capturing the resulting changes in the access control matrix would be

```
command create•file(p,f)
    create object f;
    enter own into a[p,f];
    enter r into a[p,f];
    enter w into a[p,f];
end
```

Suppose the process p wishes to create a new process q. The following command would capture the resulting changes in the access control matrix.

```
command spawn•process(p, q)
    create subject q;
    enter own into a[p, q];
    enter r into a[p, q];
    enter w into a[p, q];
```

```
        enter $r$ into $a[q, p]$;
        enter $w$ into $a[q, p]$;
    end
```

The *r* and *w* rights enable the parent and child to signal each other.

The system can update the matrix only by using defined commands; it cannot use the primitive commands directly. Of course, a command may invoke only a single primitive; such a command is called *mono-operational*.

EXAMPLE: The command

```
    command make•owner(p, f)
        enter own into $a[p, f]$;
    end
```

is a mono-operational command. It does not delete any existing owner rights. It merely adds *p* to the set of owners of *f*. Hence, *f* may have multiple owners after this command is executed.

2.3.1 Conditional Commands

The execution of some primitives requires that specific preconditions be satisfied. For example, suppose a process *p* wishes to give another process *q* the right to read a file *f*. In some systems, *p* must own *f*. The abstract command would be

```
    command grant•read•file•1(p, f, q)
        if own in $a[p, f]$
        then
            enter $r$ into $a[q, f]$;
    end
```

Any number of conditions may be placed together using **and**. For example, suppose a system has the distinguished right *c*. If a subject has the rights *r* and *c* over an object, it may give any other subject *r* rights over that object. Then

```
    command grant•read•file•2(p, f, q)
        if $r$ in $a[p, f]$ and $c$ in $a[p, f]$
        then
            enter $r$ into $a[q, f]$;
    end
```

Commands with one condition are called *monoconditional*. Commands with two conditions are called *biconditional*. The command *grant•read•file•1* is monoconditional, and the command *grant•read•file•2* is biconditional. Because both have one primitive command, both are mono-operational.

Note that all conditions are joined by **and**, and never by **or**. Because joining conditions with **or** is equivalent to two commands each with one of the conditions, the disjunction is unnecessary and thus is omitted. For example, suppose the right *a* enables one to grant the right *r* to another subject. To achieve the effect of a command equivalent to

```
    if own in a[p,f] or a in a[p,f]
    then
        enter r into a[q,f];
```

define the following two commands:

```
command grant•write•file•1(p, f, q)
    if own in a[p,f]
    then
        enter r into a[q,f];
end
command grant•write•file•2(p, f, q)
    if a in a[p,f]
    then
        enter r into a[q,f];
end
```

and then say

```
grant•write•file•1(p, f, q); grant•write•file•2(p, f, q);
```

Also, the negation of a condition is not permitted—that is, one cannot test for the *absence* of a right within a command by the condition

```
if r not in A[p,f]
```

This has some interesting consequences, which we will explore in the next chapter.

2.4 Copying, Owning, and the Attenuation of Privilege

Two specific rights are worth discussing. The first augments existing rights and is called the *copy flag*; the second is the *own right*. Both of these rights are related to the principle of attenuation of privilege, which essentially says that a subject may not give away rights it does not possess.

2.4.1 Copy Right

The *copy right* (often called the *grant right*) allows the possessor to grant rights to another. By the principle of attenuation, only those rights the grantor possesses may be copied. Whether the copier must surrender the right, or can simply pass it on, is specific to the system being modeled. This right is often considered a flag attached to other rights; in this case, it is known as the *copy flag*.

EXAMPLE: In Windows NT, the copy flag corresponds to the "P" (change permission) right.

EXAMPLE: System R is a relational database developed by the IBM Corporation. Its authorization model [337, 426] takes the database tables as objects to be protected. Each table is a separate object, even if the same records are used to construct the table (meaning that two different views of the same records are treated as two separate objects). The users who access the tables are the subjects. The database rights are *read* entries, which define new views on an existing table; *insert*, *delete*, and *update* entries in a table; and *drop* (to delete a table). Associated with each right is a *grant option*; if it is set, the possessor of the privilege can grant it to another. Here, the grant option corresponds to a copy flag.

EXAMPLE: Let c be the copy right, and suppose a subject p has r rights over an object f. Then the following command allows p to copy r over f to another subject q only if p has a copy right over f.

```
command grant•r(p, f, q)
    if r in a[p, f] and c in a[p, f]
    then
        enter r into a[q, f];
end
```

EXAMPLE: If p does not have c rights over f, this command will not copy the r rights to q.

2.4.2 Own Right

The *own right* is a special right that enables possessors to add or delete privileges for themselves. It also allows the possessor to grant rights to others, although to whom they can be granted may be system- or implementation-dependent. The owner of an object is usually the subject that created the object or a subject to which the creator gave ownership.

EXAMPLE: On UNIX systems, the owner may use the *chown*(1) command to change the permissions that others have over an object. The semantics of delegation of owner-

image
not
available

   a. Create the corresponding access control matrix.
   b. Cyndy gives Alice permission to read *cyndyrc*, and Alice removes Bob's ability to read *alicerc*. Show the new access control matrix.

2. In Miller and Baldwin's model (see Section 2.2.1), they restricted the functions that generated the access control matrix entries to working on objects, not on subjects. Thus, one could not base rights being granted on whether another subject possessed those rights. Why did they impose this restriction? Can you think of cases in which this restriction would cause problems?
3. The query-set-overlap mechanism requires a history of all queries to the database. Discuss the feasibility of this control. In particular, how will the size of the history affect the response of the mechanism.
4. Consider the set of rights {*read*, *write*, *execute*, *append*, *list*, *modify*, *own*}.
   a. Using the syntax in Section 2.3, write a command *delete_all_rights* (p, q, s). This command causes p to delete all rights the subject q has over an object s.
   b. Modify your command so that the deletion can occur only if p has *modify* rights over s.
   c. Modify your command so that the deletion can occur only if p has *modify* rights over s and q does *not* have *own* rights over s.
5. Let c be a copy flag and let a computer system have the same rights as in Exercise 4.
   a. Using the syntax in Section 2.3, write a command *copy_all_rights*(p, q, s) that copies all rights that p has over s to q.
   b. Modify your command so that only those rights with an associated copy flag are copied. The new copy should *not* have the copy flag.
   c. In part (b), what conceptually would be the effect of copying the copy flag along with the right?
6. This exercise asks you to consider the consequences of not applying the principle of attenuation of privilege to a computer system.
   a. What are the consequences of not applying the principle at all? In particular, what is the maximal set of rights that subjects within the system can acquire (possibly with the cooperation of other subjects)?
   b. Suppose attenuation of privilege applied only to *access* rights such as *read* and *write*, but not to rights such as *own* and *grant_rights*. Would this ameliorate the situation discussed in part (a)? Why or why not?

c. Consider a restricted form of attenuation, which works as follows. A subject q is attenuated by the maximal set of rights that q, or any of its ancestors, has. So, for example, if any ancestor of q has r permission over a file f, q can also r f. How does this affect the spread of rights throughout the access control matrix of the system? Develop an example matrix that includes the ancestor right, and illustrate your answer.

Chapter 3
Foundational Results

MARIA: Ay, but you must confine yourself
within the modest limits of order.
—*Twelfth Night*, I, iii, 8–9.

In 1976, Harrison, Ruzzo, and Ullman [450] proved that in the most general abstract case, the security of computer systems was undecidable and explored some of the limits of this result. In that same year, Jones, Lipton, and Snyder [527] presented a specific system in which security was not only decidable, but decidable in time linear with the size of the system. Minsky [718] suggested a third model to examine what made the general, abstract case undecidable but at least one specific case decidable. Sandhu [870] extended this model to examine the boundary even more closely.

These models explore the most basic question of the art and science of computer security: under what conditions can a generic algorithm determine whether a system is secure? Understanding models and the results derived from them lays the foundations for coping with limits in policy and policy composition as well as applying the theoretical work.

3.1 The General Question

Given a computer system, how can we determine if it is secure? More simply, is there a generic algorithm that allows us to determine whether a computer system is secure? If so, we could simply apply that algorithm to any system; although the algorithm might not tell us where the security problems were, it would tell us whether any existed.

The first question is the definition of "secure." What policy shall define "secure"? For a general result, the definition should be as broad as possible. We use the access control matrix to express our policy. However, we do not provide any special rights such as *copy* or *own*, and the principle of attenuation of privilege does not apply.

Let R be the set of generic (primitive) rights of the system.

Definition 3–1. When a generic right r is added to an element of the access control matrix not already containing r, that right is said to be *leaked*.

Our policy defines the authorized set of states A to be the set of states in which no command $c(x_1, ..., x_n)$ can leak r. This means that no generic rights can be added to the matrix.

We do not distinguish between the *leaking* of rights and an *authorized* transfer of rights. In our model, there is *no* authorized transfer of rights. (If we wish to allow such a transfer, we designate the subjects involved as "trusted." We then eliminate all trusted subjects from the matrix, because the security mechanisms no longer apply to them.)

Let a computer system begin in protection state s_0.

Definition 3–2. If a system can never leak the right r, the system (including the initial state s_0) is called *safe with respect to the right r*. If the system can leak the right r (enter an unauthorized state), it is called *unsafe with respect to the right r*.

We use these terms rather than *secure* and *nonsecure* because safety refers to the abstract model and security refers to the actual implementation. Thus, a secure system corresponds to a model safe with respect to all rights, but a model safe with respect to all rights does not ensure a secure system.

EXAMPLE: A computer system allows the network administrator to read all network traffic. It disallows all other users from reading this traffic. The system is designed in such a way that the network administrator cannot communicate with other users. Thus, there is no way for the right r of the network administrator over the network device to leak. This system is safe.

Unfortunately, the operating system has a flaw. If a user specifies a certain file name in a file deletion system call, that user can obtain access to any file on the system (bypassing all file system access controls). This is an implementation flaw, not a theoretical one. It also allows the user to read data from the network. So this system is not secure.

Our question (called the *safety question*) is: Does there exist an algorithm for determining whether a given protection system with initial state s_0 is safe with respect to a generic right r?

3.2 Basic Results

The simplest case is a system in which the commands are mono-operational (each consisting of a single primitive command). In such a system, the following theorem holds.

Theorem 3–1. [450] There exists an algorithm that will determine whether a given mono-operational protection system with initial state s_0 is safe with respect to a generic right *r*.

Proof Because all commands are mono-operational, we can identify each command by the type of primitive operation it invokes. Consider the minimal sequence of commands $c_1, ..., c_k$ needed to leak the right *r* from the system with initial state s_0.

Because no commands can test for the absence of rights in an access control matrix entry, we can omit the **delete** and **destroy** commands from the analysis. They do not affect the ability of a right to leak.

Now suppose that multiple **create** commands occurred during the sequence of commands, causing a leak. Subsequent commands check only for the presence of rights in an access control matrix element. They distinguish between different elements only by the presence (or lack of presence) of a particular right. Suppose that two subjects s_1 and s_2 are created and the rights in $A[s_1, o_1]$ and $A[s_2, o_2]$ are tested. The same test for $A[s_1, o_1]$ and $A[s_1, o_2] = A[s_1, o_2] \cup A[s_2, o_2]$ will produce the same result. Hence, all **create**s are unnecessary except possibly the first (and that only if there are no subjects initially), and any commands **enter**ing rights into the new subjects are rewritten to enter the new right into the lone created subject. Similarly, any tests for the presence of rights in the new subjects are rewritten to test for the presence of that right in an existing subject (or, if none initially, the first subject created).

Let $|S_0|$ be the number of subjects and $|O_0|$ the number of objects in the initial state. Let *n* be the number of generic rights. Then, in the worst case, one new subject must be created (one command), and the sequence of commands will enter every right into every element of the access control matrix. After the creation, there are $|S_0| + 1$ subjects and $|O_0| + 1$ objects, and $(|S_0| + 1)(|O_0| + 1)$ elements. Because there are *n* generic rights, this leads to $n(|S_0| + 1)(|O_0| + 1)$ commands. Hence, $k \leq n(|S_0| + 1)(|O_0| + 1) + 1$.

By enumerating all possible states we can determine whether the system is safe. Clearly, this may be computationally infeasible, especially if many subjects, objects, and rights are involved, but it is computable. (See Exercise 2.) Unfortunately, this result does not generalize to all protection systems.

Before proving this, let us review the notation for a *Turing machine*. A Turing machine *T* consists of a head and an infinite tape divided into cells numbered 1, 2, ..., from left to right. The machine also has a finite set of states *K* and a finite set of tape symbols *M*. The distinguished symbol $b \in M$ is a blank and appears on all the cells of the tape at the start of all computations; also, at that time *T* is in the initial state q_0.

The tape head occupies one square of the tape, and can read and write symbols on that cell of the tape, and can move into the cell to the left or right of the cell it currently occupies. The function $\delta: K \times M \rightarrow K \times M \times \{L, R\}$ describes the action of *T*. For example, let $p, q \in K$ and $A, B \in M$. Then, if $\delta(p, A) = (q, B, R)$, when *T* is in state *p* and the head rests on a cell with symbol *A*, the tape head changes the symbol

```
        enter $end$ into $a[s_{i+1}, s_{i+1}]$;
        delete $p$ from $a[s_i, s_i]$;
        delete $A$ from $a[s_i, s_i]$;
        enter $B$ into $a[s_i, s_i]$;
        enter $q$ into $a[s_{i+1}, s_{i+1}]$;
    end
```

Clearly, only one right in any of the access control matrices corresponds to a state, and there will be exactly one *end* right in the matrix (by the nature of the commands simulating Turing machine actions). Hence, in each configuration of the Turing machine, there is at most one applicable command. Thus, the protection system exactly simulates the Turing machine, given the representation above. Now, if the Turing machine enters state q_f, then the protection system has leaked the right q_f; otherwise, the protection system is safe for the generic right q_f. But whether the Turing machine will enter the (halting) state q_f is undecidable, so whether the protection system is safe must be undecidable also.

However, we can generate a list of all unsafe systems.

Theorem 3–3. [269] The set of unsafe systems is recursively enumerable.

Proof See Exercise 3.

Assume that the **create** primitive is disallowed. Clearly, the safety question is decidable (simply enumerate all possible sequences of commands from the given state; as no new subjects or objects are created, at some point no new rights can be added to any element of the access control matrix, so if the leak has not yet occurred, it cannot occur). Hence, we have the following theorem.

Theorem 3–4. [450] For protection systems without the **create** primitives, the question of safety is complete in **P-SPACE**.

Proof Consider a Turing machine bounded in polynomial space. A construction similar to that of Theorem 3–2 reduces that Turing machine in polynomial time to an access control matrix whose size is polynomial in the length of the Turing machine input.

If deleting the **create** primitives makes the safety question decidable, would deleting the **delete** and **destroy** primitives but not the **create** primitive also make the safety question decidable? Such systems are called *monotonic* because they only increase in size and complexity; they cannot decrease. But:

Theorem 3–5. [451] It is undecidable whether a given configuration of a given monotonic protection system is safe for a given generic right.

Restricting the number of conditions in the commands to two does not help:

Theorem 3–6. [451] The safety question for biconditional monotonic protection systems is undecidable.

But if at most one condition per command is allowed:

Theorem 3–7. [451] The safety question for monoconditional monotonic protection systems is decidable.

This can be made somewhat stronger:

Theorem 3–8. [451] The safety question for monoconditional protection systems with **create**, **enter**, and **delete** primitives (but no **destroy** primitive) is decidable.

Thus, the safety question is undecidable for generic protection models but is decidable if the protection system is restricted in some way. Two questions arise. First, given a *particular* system with specific rules for transformation, can we show that the safety question is decidable? Second, what are the weakest restrictions on a protection system that will make the safety question decidable in that system?

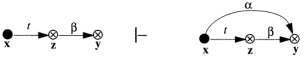
3.3 The Take-Grant Protection Model

Can the safety of a particular system, with specific rules, be established (or disproved)? The answer, not surprisingly, is yes. Such a system is the *Take-Grant Protection Model*.

The Take-Grant Protection Model represents a system as a directed graph. Vertices are either subjects (represented by ●) or objects (represented by ○). Vertices that may be either subjects or objects are represented by ⊗. Edges are labeled, and the label indicates the rights that the source vertex has over the destination vertex. Rights are elements of a predefined set *R*; *R* contains two distinguished rights: *t* (for *take*) and *g* (for *grant*).

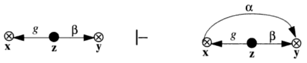
As the protection state of the system changes, so does the graph. The protection state (and therefore the graph) changes according to four *graph rewriting rules*:

Take rule: Let **x**, **y**, and **z** be three distinct vertices in a protection graph G_0, and let **x** be a subject. Let there be an edge from **x** to **z** labeled γ with $t \in \gamma$, an edge from **z** to **y** labeled β, and $\alpha \subseteq \beta$. Then the *take* rule defines a new graph G_1 by adding an edge to the protection graph from **x** to **y** labeled α. Graphically,



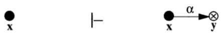
The rule is written "**x** takes (α to **y**) from **z**."

Grant rule: Let **x**, **y**, and **z** be three distinct vertices in a protection graph G_0, and let **z** be a subject. Let there be an edge from **z** to **x** labeled γ with $g \in \gamma$, an edge from **z** to **y** labeled β, and $\alpha \subseteq \beta$. Then the *grant* rule defines a new graph G_1 by adding an edge to the protection graph from **x** to **y** labeled α. Graphically,



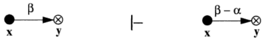
The rule is written "**z** grants (α to **y**) to **x**."

Create rule: Let **x** be any subject in a protection graph G_0 and let $\alpha \subseteq R$. Then *create* defines a new graph G_1 by adding a new vertex **y** to the graph and an edge from **x** to **y** labeled α. Graphically,



The rule is written "**x** creates (α to new vertex) **y**."

Remove rule: Let **x** and **y** be any distinct vertices in a protection graph G_1 such that **x** is a subject. Let there be an explicit edge from **x** to **y** labeled β, and let $\alpha \subseteq \beta$. Then *remove* defines a new graph G_1 by deleting the α labels from β. If β becomes empty as a result, the edge itself is deleted. Graphically,



The rule is written "**x** removes (α to) **y**."

Because these rules alter the state of the protection graph, they are called *de jure* ("by law" or "by right") rules.

We demonstrate that one configuration of a protection graph can be derived from another by applying the four rules above in succession. The symbol $\vdash$ means that the graph following it is produced by the action of a graph rewriting rule on the graph preceding it; and the symbol $\vdash^*$ represents a finite number of successive rule applications. Such a sequence of graph rewriting rules is called a *witness*. A witness is often demonstrated by listing the graph rewriting rules that make up the witness (usually with pictures).

3.3.1 Sharing of Rights

We first wish to determine if a given right α can be shared—that is, given a protection graph G_0, can a vertex **x** obtain α rights over another vertex **y**? More formally:

> **Definition 3–3.** The predicate *can•share*(α, **x**, **y**, G_0) is true for a set of rights α and two vertices **x** and **y** if and only if there exists a sequence of protection graphs $G_1, \ldots, G_n$ such that $G_0 \vdash^* G_n$ using only *de jure* rules and in G_n there is an edge from **x** to **y** labeled α.

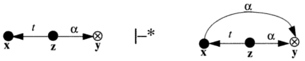
To establish the conditions under which this predicate will hold, we must define a few terms.

> **Definition 3–4.** A *tg-path* is a nonempty sequence $\mathbf{v}_0, \ldots, \mathbf{v}_n$ of distinct vertices such that for all i, $0 \le i < n$, $\mathbf{v}_i$ is connected to $\mathbf{v}_{i+1}$ by an edge (in either direction) with a label containing t or g.

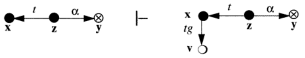
> **Definition 3–5.** Vertices are *tg-connected* if there is a *tg*-path between them.

We can now prove that any two subjects with a *tg*-path of length 1 can share rights. Four such paths are possible. The take and grant rules in the preceding section account for two of them. Lemmata 3–1 and 3–2 cover the other two cases.

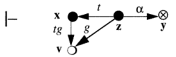
Lemma 3–1.



Proof **x** creates (*tg* to new vertex) **v**.

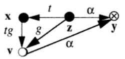


z takes (*g* to **v**) from **x**.



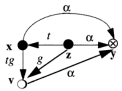
z grants (α to **y**) to **v**.

$\vdash$



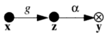
x takes (α to **y**) from **v**.

$\vdash$

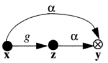


This sequence of rule applications adds an edge labeled α from **x** to **y**. A similar proof establishes the following lemma.

Lemma 3–2.



$\vdash^*$



Thus, the *take* and *grant* rules are symmetric if the vertices on the *tg*-path between **x** and **y** are subjects. This leads us to the following definition.

Definition 3–6. An *island* is a maximal *tg*-connected subject-only subgraph.

Because an island is a maximal *tg*-only subgraph, a straightforward inductive proof shows that any right possessed by any vertex in the island can be shared with any other vertex in the island.

Transferring rights between islands requires that a subject in one island be able to take the right from a vertex in the other island or that a subject be able to grant the right to an intermediate object from which another subject in the second island may take the right. This observation, coupled with the symmetry of take and grant, leads to a characterization of paths between islands along which rights can be transferred. To express it succinctly, we use the following notation. With each *tg*-path, associate one or more words over the alphabet in the obvious way. If the

b. For all pairs of vertices $\mathbf{x}_i$ and $\mathbf{x}_j$ in G with $\mathbf{x}_i$ having α rights over $\mathbf{x}_j$, perform "**v** grants (α to $\mathbf{x}_j$) to $\mathbf{x}_i$."
c. Let β be the set of rights labeling the edge from $\mathbf{x}_i$ and $\mathbf{x}_j$ in G (note that β may be empty). Perform "**v** removes (($\alpha \cup \{ g \}) - \beta$ to) $\mathbf{x}_j$."

The resulting graph G' is the desired graph G.
($\Leftarrow$). Let **v** be the initial subject, and let $G_0 \vdash^* G$. By inspection of the rules, G is finite, loop-free, and a directed graph; furthermore, it consists of subjects and objects only, and all edges are labeled with nonempty subsets of R.

Because no rule allows the deletion of vertices, **v** is in G. Because no rule allows an incoming edge to be added to a vertex without any incoming edges, and **v** has no incoming edges, it cannot be assigned any.

Corollary 3–2. [944] A k-component, n-edge protection graph can be constructed from t-rule applications, where $2(k-1) + n \le t \le 2(k-1) + 3n$.

Using the Take-Grant Protection Model, Snyder [943] showed how some common protection problems could be solved. For example, suppose two processes **p** and **q** communicate through a shared buffer **b** controlled by a trusted entity **s** (for example, an operating system). The configuration in Figure 3–2a shows the initial protection state of the system. Because **s** is a trusted entity, the assumption that it has g rights over **p** and **q** is reasonable. To create **b**, and to allow **p** and **q** to communicate through it, **s** does the following:

a. **s** creates ($\{r, w\}$ to new object) **b**.
b. **s** grants ($\{r, w\}$ to **b**) to **p.**
c. **s** grants ($\{r, w\}$ to **b**) to **q.**

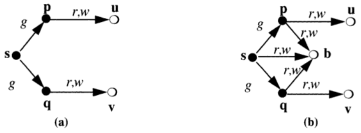


Figure 3–2 (a) The initial state of the system: s, a trusted entity, can grant rights to untrusted processes p and q. Each process p and q controls its own private information (here represented by files u and v). (b) The trusted entity has created a buffer b shared by the untrusted processes.

This creates the configuration in Figure 3–2(b). The communication channel is two-way; if it is to be one-way, the sender would have write rights and the receiver would have read rights. This configuration also captures the ability of the trusted entity to monitor the communication channel or interfere with it (by altering or creating messages)—a point we will explore in later sections.

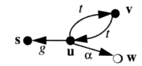
3.3.3 Theft in the Take-Grant Protection Model

The proof of the conditions necessary and sufficient for *can•share* requires that all subjects involved in the witness cooperate. This is unrealistic. If Professor Olson does not want any students to read her grade file, the notion of "sharing" fails to capture the unwillingness to grant access. This leads to a notion of *stealing*, in which no owner of any right over an object grants that right to another.

Definition 3–10. Let G_0 be a protection graph, let **x** and **y** be distinct vertices in G_0, and let α be a subset of a set of rights R. The predicate *can•steal*(α, **x**, **y**, G_0) is true when there is no edge from **x** to **y** labeled α in G_0 and there exists a sequence of protection graphs $G_1, \ldots, G_n$ for which the following hold simultaneously:

a. There is an edge from **x** to **y** labeled α in G_n.
b. There is a sequence of rule applications $\rho_1, \ldots, \rho_n$ such that $G_{i-1} \vdash G_i$ using ρ_i.
c. For all vertices **v** and **w** in G_{i-1}, $1 \leq i < n$, if there is an edge from **v** to **y** in G_0 labeled α, then ρ_i is not of the form "**v** grants (α to **y**) to **w**."

This definition disallows owners of α rights to **y** from transferring those rights. It does *not* disallow those owners from transferring other rights. Consider Figure 3–3. The given witness exhibits *can•steal*(α, **s**, **w**, G_0). In step (1), the owner of α rights to **w** grants other rights (specifically, *t* rights to **v**) to a different subject, **s**. Without this step, the theft cannot occur. The definition only forbids grants of the rights to be stolen. Other rights may be granted. One justification for this formulation is the ability of attackers to trick others into surrendering rights. While the owner of the target



(1) **u** grants (*t* to **v**) to **s**.
(2) **s** takes (*t* to **u**) from **v**.
(3) **s** takes (α to **w**) from **u**.

Figure 3–3 A witness to theft in which the owner, u, of the stolen right, α, grants other rights to another subject (*t* rights to v are granted to s).

right would be unlikely to grant that right, the owner might grant other rights. This models the Trojan horse (see Section 22.2), in which the owner of the rights is unaware she is giving them away.

Making the target of the theft a subject complicates this situation. According to Definition 3–10, the target may give away any rights as well. In this case, the owner is acting as a moderator between the target and the source and must restrain the transfer of the right through it. This models the case of mandatory access controls.

Theorem 3–12. [944] The predicate *can•steal*(α, **x**, **y**, G_0) is true if and only if the following hold simultaneously:

a. There is no edge from **x** to **y** labeled α in G_0.
b. There exists a subject vertex **x**´ such that **x**´ = **x** or **x**´ initially spans to **x**.
c. There exists a vertex **s** with an edge labeled α to **y** in G_0 and for which *can•share*(t, **x**, **s**, G_0) holds.

Proof (⇒). Assume that the three conditions above hold. If **x** is a subject, then **x** need merely obtain t rights to **s** and then use the take rule to obtain α rights to **y**. By definition, this satisfies *can•steal*(α, **x**, **y**, G_0).

Suppose **x** is an object. Then Theorem 3–10, *can•share*(t, **x**, **s**, G_0), implies that there exists a subject vertex **x**´ that *tg*-initially spans to **x** and for which the predicate *can•share*(t, **x**´, **s**, G_0) is true. Without loss of generality, assume that the *tg*-initial span is of length 1 and that **x**´ has t rights over **s** in G_0. If **x**´ does not have an edge labeled α to **y** in G_0, then **x**´ takes α rights to **y** and grants those rights to **x**, satisfying the definition. If **x**´ has an edge labeled α to **y** in G_0, then **x**´ will create a "surrogate" to which it can give take rights to **s**:

1. **x**´ creates (g to new subject) **x**˝.
2. **x**´ grants (t to **s**) to **x**˝.
3. **x**´ grants (g to **x**) to **x**˝.

Now **x**˝ has t rights over **s** and g rights over **x**, so the rule applications

1. **x**˝ takes (α to **y**) from **s**.
2. **x**˝ grants (α to **y**) to **x**.

satisfy the definition. Hence, *can•steal*(α, **x**, **y**, G_0) holds if the three conditions in the theorem hold.

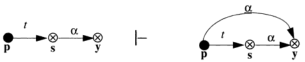
(⇐): Assume that *can•steal*(α, **x**, **y**, G_0) holds. Then condition (a) of the theorem holds directly from Definition 3–10.

Condition (a) of Definition 3–10 implies *can•share*(α, **x**, **y**, G_0). From condition (b) of Theorem 3–10, we immediately obtain condition (b) of this theorem.

Condition (a) of Theorem 3–10 ensures that the vertex **s** in condition (c) of this theorem exists.

We must show that *can•share*(*t*, **x**, **s**, G_0) holds. Let ρ be a sequence of rule applications. Consider the minimal length sequence of rule applications deriving G_n from G_0. Let *i* be the least index such that $G_{i-1} \vdash_{\rho i} G_i$ and such that there is an edge labeled α from some vertex **p** to **y** in G_i but not in G_{i-1}. Then G_i is the first graph in which an edge labeled α to **y** is added.

Obviously, ρ_i is not a remove rule. It cannot be a create rule, because **y** already existed. By condition (c) of Definition 3–10, and the choice of *i* ensuring that all vertices with α rights to **y** in G_i are also in G_0, ρ_i cannot be a grant rule. Hence, ρ_i must be a take rule of the form



for some vertex **s** in G_0. From this, *can•share*(*t*, **p**, **s**, G_0) holds. By condition (c) of Theorem 3–10, there is a subject **s**´ such that **s**´ = **s** or **s**´ terminally spans to **s**, and by condition (d), there exists a sequence of islands $I_1, \ldots, I_n$ such that **x**´ ∈ I_1 and **s**´ ∈ I_n.

If **s** is an object (and thus **s**´ ≠ **s**), consider two cases. If **s**´ and **p** are in the same island, then take **p** = **s**´. If they are in different islands, the derivation cannot be of minimal length; choose **s**´ in the same island to exhibit a shorter one. From this, the conditions of Theorem 3–10 have been met, and *can•share*(*t*, **x**, **s**, G_0) holds.

If **s** is a subject (**s**´ = **s**), then **p** ∈ I_n, and we must show that **p** ∈ G_0 for Theorem 3–10 to hold. If **p** ∉ G_0, then there is a subject **q** in one of the islands such that *can•share*(*t*, **q**, **s**, G_0) holds. (To see this, note that **s** ∈ G_0 and that none of the *de jure* rules add new labels to incoming edges on existing vertices.) Because **s** is an owner of α rights to **y** in G_0, we must derive a witness for this sharing in which **s** does not grant (α to **q**). If **s** and **q** are distinct, replace each rule application of the form

s grants (α to **y**) to **q**

with the sequence

p takes (α to **y**) from **s**
p takes (*g* to **q**) from **s**
p grants (α to **y**) to **q**

thereby transferring the right (α to **y**) to **q** without **s** granting. If **s** = **q**, then the first rule application in this sequence suffices.

Hence, there exists a witness to *can•share*(t, **q**, **s**, G_0) in which **s** does not grant (α to **y**). This completes the proof.

3.3.4 Conspiracy

The notion of theft introduced the issue of cooperation: which subjects are actors in a transfer of rights, and which are not? This raises the issue of the number of actors necessary to transfer a right. More formally, what is the minimum number of actors required to witness a given predicate *can•share*(α, **x**, **y**, G_0)?

Consider a subject vertex **y**. Then **y** can share rights from any vertex to which it terminally spans and can pass those rights to any vertex to which it initially spans.

Definition 3–11. The *access set A(y) with focus* **y** is the set of vertices **y**, all vertices **x** to which **y** initially spans, and all vertices **x´** to which **y** terminally spans.

Of course, a focus must be a subject.

Consider two access sets with different foci **y** and **y´** that have a vertex **z** in common. If **z** ∈ A(**y**) because **y** initially spans to **z**, and **z** ∈ A(**y´**) because **y´** initially spans to **z**, by the definition of initial span, no rights can be transferred between **y** and **y´** through **z**. A similar result holds if both **y** and **y´** terminally span to **z**. However, if one focus initially spans to **z** and the other terminally spans to **z**, rights can be transferred through **z**. Because we care about the transfer of rights, we identify a set of vertices that can be removed from the graph without affecting transfers:

Definition 3–12. The *deletion set* δ(**y**, **y´**) contains all vertices **z** in the set A(**y**) ∩ A(**y´**) for which (a) **y** initially spans to **z** and **y´** terminally spans to **z**, (b) **y** terminally spans to **z** and **y´** initially spans to **z**, (c) **z** = **y**, and (d) **z** = **y´**.

Given the deletion set, we construct an undirected graph, called the *conspiracy graph* and represented by H, from G_0:

1. For each subject vertex **x** in G_0, there is a corresponding vertex h(**x**) in H with the same label.
2. If δ(**y**, **y´**) ≠ ∅ in G_0, there is a line between h(**y**) and h(**y´**) in H.

The conspiracy graph represents the paths along which subjects can transfer rights. The paths are unidirectional because the rights can be transmitted in either direction. Furthermore, each vertex in H represents an access set focus in G_0.

The first paper introduced a model called the Schematic Send-Receive (SSR) Protection Model [869]. The Schematic Protection Model (SPM) [870] generalizes these results.

3.4.1 Schematic Protection Model

The key notion of the Schematic Protection Model, also called the SPM, is the *protection type*. This is a label for an entity that determines how control rights affect that entity. For example, if the Take-Grant Protection Model is viewed as an instance of a scheme under the SPM, the protection types are *subject* and *object* because the control rights *take*, *grant*, *create*, and *remove* affect subject entities differently than they do object entities. Moreover, under SPM, the protection type of an entity is set when the entity is created, and cannot change thereafter.

In SPM, a *ticket* is a description of a single right. An entity has a set of tickets (called a *domain*) that describe what rights it has over another entity. A ticket consists of an *entity name* and a *right symbol*; for example, the ticket **X**/r allows the possessor of the ticket to apply the right r to the entity **X**. Although a ticket may contain only one right, if an entity has multiple tickets **X**/r, **X**/s, and **X**/t, we abbreviate them by writing **X**/rst.

Rights are partitioned into a set of *inert rights* (*RI*) or *control rights* (*RC*). Applying an inert right does not alter the protection state of the system. For example, reading a file does not modify which entities have access to the document, so *read* is an inert right. But in the Take-Grant Protection Model, applying the take rule does change the protection state of the system (it gives a subject a new right over an object). Hence, the *take* right is a control right. SPM ignores the effect of applying inert rights, but not the effect of applying control rights.

The attribute c is a *copy flag*; every right r has an associated copyable right rc. A ticket with the copy flag can be copied to another domain. The notation $r{:}c$ means r or rc, with the understanding that all occurrences of $r{:}c$ are read as r or all are read as rc.

We partition the set of types T into a *subject set* TS and an *object set* TO. The type of an entity **X** is written $\tau(\mathbf{X})$. The type of a ticket **X**/$r{:}c$ is $\tau(\mathbf{X}/r{:}c)$, which is the same as $\tau(\mathbf{X})/r{:}c$. More formally, let E be the set of entities; then $\tau{:}E \to T$ and $\tau{:}E \times R \to T \times R$.

The manipulation of rights is controlled by two relationships: a *link predicate* and a *filter function*. Intuitively, the link predicate determines whether the source and target of the transfer are "connected" (in a mathematical sense), and the filter function determines whether the transfer is authorized.

3.4.1.1 Link Predicate

A link predicate is a relation between two subjects. It is local in the sense that its evaluation depends *only* on the tickets that the two subjects possess. Formally:

Definition 3–13. Let $dom(\mathbf{X})$ be the set of tickets that $\mathbf{X}$ possesses. A *link predicate* $link_i(\mathbf{X}, \mathbf{Y})$ is a conjunction or disjunction (but not a negation) of the following terms, for any right $z \in RC$.

1. $\mathbf{X}/z \in dom(\mathbf{X})$
2. $\mathbf{X}/z \in dom(\mathbf{Y})$
3. $\mathbf{Y}/z \in dom(\mathbf{X})$
4. $\mathbf{Y}/z \in dom(\mathbf{Y})$
5. **true**

A finite set of link predicates $\{ link_i \mid i = 1, \ldots, n \}$ is called a *scheme*. If only one link predicate is defined, we omit the subscript i.

EXAMPLE: The link predicate corresponding to the Take-Grant Protection Model rules *take* and *grant* is

$$link(\mathbf{X}, \mathbf{Y}) = \mathbf{Y}/g \in dom(\mathbf{X}) \vee \mathbf{X}/t \in dom(\mathbf{Y})$$

Here, **X** and **Y** are connected if **X** has g rights over **Y** or **Y** has t rights over **X**, which corresponds to the model in the preceding section.

EXAMPLE: The link predicate

$$link(\mathbf{X}, \mathbf{Y}) = \mathbf{X}/b \in dom(\mathbf{X})$$

connects **X** to every other entity **Y** provided that **X** has b rights over itself. With respect to networks, b would correspond to a broadcast right. However, **X** does not yet have the right to broadcast to all **Y** because predicates do not endow the ability to exercise that right. Similarly, the predicate

$$link(\mathbf{X}, \mathbf{Y}) = \mathbf{Y}/p \in dom(\mathbf{Y})$$

corresponds to a *pull* connection between all entities **X** and **Y**. Again, this is not sufficient for **Y** to exercise the pull right, but it is necessary.

EXAMPLE: The *universal link* depends on no entity's rights:

$$link(\mathbf{X}, \mathbf{Y}) = \mathbf{true}$$

This link holds even when **X** and **Y** have no tickets that refer to each other.

3.4.1.2 Filter Function

The filter functions impose conditions on when transfer of tickets can occur. Specifically, a filter function is a function f_i: $TS \times TS \rightarrow 2^{T \times R}$ that has as its range the set of copyable tickets. For a copy to occur, the ticket to be copied must be in the range of the appropriate filter function.

Combining this requirement with the others, a ticket **X**/*r*:*c* can be copied from *dom*(**Y**) to *dom*(**Z**) if and only if, for some *i*, the following are true:

1. $\mathbf{X}/rc \in dom(\mathbf{Y})$
2. $link_i(\mathbf{Y}, \mathbf{Z})$
3. $\tau(\mathbf{X})/r{:}c \in f_i(\tau(\mathbf{Y}), \tau(\mathbf{Z}))$

One filter function is defined for each *link* predicate. As with the *link* predicates, if there is only one filter function, we omit the subscripts.

EXAMPLE: Let $f(\tau(\mathbf{Y}), \tau(\mathbf{Z})) = T \times R$. Then any tickets are transferable, assuming that the other two conditions are met. However, if $f(\tau(\mathbf{Y}), \tau(\mathbf{Z})) = T \times RI$, then only inert rights are transferable; and if $f(\tau(\mathbf{Y}), \tau(\mathbf{Z})) = \varnothing$, no rights can be copied.

3.4.1.3 Putting It All Together

Let us take stock of these terms by considering two examples: an *owner-based policy* and the Take-Grant Protection Model.

In an owner-based policy, a subject **U** can authorize another subject **V** to access an object **F** if and only if **U** owns **F**. Here, the set of subjects is the set of users and the set of objects is the set of files. View these as types. Then:

$$TS = \{ \, user \, \}, TO = \{ \, file \, \}$$

In this model, ownership is best viewed as copy attributes—that is, if **U** owns **F**, all its tickets for **F** are copyable. Under this interpretation, the set of control rights is empty because no rights are required to alter the state of the protection graph. All rights are inert. For our example, assume that the *r* (read), *w* (write), *a* (append), and *x* (execute) rights are defined. Then:

$$RC = \varnothing, RI = \{ \, r{:}\text{c}, w{:}\text{c}, a{:}\text{c}, x{:}\text{c} \, \}$$

Because the owner can give the right to any other subject, there is a connection between each pair of subjects and the link predicate is always true:

$$link(\mathbf{U}, \mathbf{V}) = \textbf{true}$$

Finally, tickets can be copied across these connections:

$$f(user, user) = \{ \, file/r, file/w, file/a, file/x \, \}$$

EXAMPLE: Suppose a user Peter wishes to give another user Paul execute permissions over a file called *doom*. Then $\tau(\text{Peter}) = \mathit{user}$, $\tau(\mathit{doom}) = \mathit{file}$, and $\mathit{doom}/xc \in \mathit{dom}(\text{Peter})$. Because any user can give rights away to any other user, all users are "connected" in that sense, so $\mathit{link}(\text{Peter}, \text{Paul}) = \mathbf{true}$. Finally, because $\tau(\mathit{doom}) = \mathit{file}$, and $\tau(\text{Paul}) = \mathit{user}$, we have $\tau(\mathit{doom})/x \in f(\tau(\text{Peter}), \tau(\text{Paul}))$. Thus, Peter can copy the ticket *doom*/*x* to Paul.

The Take-Grant Protection Model can be formulated as an instance of SPM. The set of subjects and objects in the Take-Grant model corresponds to the set of subjects and objects in SPM:

$$TS = \{\ \mathit{subject}\ \},\ TO = \{\ \mathit{object}\ \}$$

The control rights are t (take) and g (grant), because applying them changes the protection state of the graph. All other rights are inert; for our example, we will take them to be r (read) and w (write). All rights can be copied (in fact, the Take-Grant Protection Model implicitly assumes this), so:

$$RC = \{\ tc, gc\ \},\ RI = \{\ rc, wc\ \}$$

Rights can be transferred along edges labeled t or g, meaning that one vertex on the edge has take or grant rights over the other. Let $\mathbf{p}$ and $\mathbf{q}$ be subjects. Then the link predicate is

$$\mathit{link}(\mathbf{p}, \mathbf{q}) = \mathbf{p}/t \in \mathit{dom}(\mathbf{q}) \lor \mathbf{q}/g \in \mathit{dom}(\mathbf{p})$$

Finally, any right can be transferred, so the filter function is simply

$$f(\mathit{subject}, \mathit{subject}) = \{\ \mathit{subject}, \mathit{object}\ \} \times \{\ tc, gc, rc, wc\ \}$$

We now explore how the transfer of tickets occurs in SPM.

3.4.1.4 Demand and Create Operations

The demand function $d{:}TS \rightarrow 2^{T \times R}$ authorizes a subject to demand a right from another entity. Let a and b be types. Then $a/r{:}c \in d(b)$ means that every subject of type b can demand a ticket $\mathbf{X}/r{:}c$ for all $\mathbf{X}$ such that $\tau(\mathbf{X}) = a$. This is a generalization of the *take* rule in the Take-Grant model. The *take* rule refers to an individual subject. The *demand* rule refers to all subjects of a particular type (here, of type b).

EXAMPLE: In the owner-based policy, no user can force another to give rights; hence, the range of the demand function is empty: $d(\mathit{user}) = \varnothing$. In the Take-Grant Protection Model, there is also no demand function. Although the *take* right is similar, to treat it as the demand right would require the creation of additional types to distinguish between those vertices directly connected by take edges to subjects and all other vertices. This complicates the system unnecessarily. Hence, $d(\mathit{subject}) = \varnothing$.

Sandhu [871] has demonstrated that a sophisticated construction eliminates the need for the demand operation. Thus, although the demand rule is present in SPM, that rule is omitted from the models that followed SPM.

Creating a new entity requires handling of not only the type of the new entity but also the tickets added by the creation. The type of the new entity is specified by the relation *can-create* (cc): $cc \subseteq TS \times T$; a subject of type a can create entities of type b if and only if $cc(a, b)$ holds.

In practice, the rule of *acyclic creates* limits the membership in this relation. Represent the types as vertices, and let a directed edge go from a to b if $cc(a, b)$. The relation cc is acyclic if this graph has no loops except possibly edges from one vertex to itself. Figure 3–5 gives an example of both cyclic and acyclic *can-create* relations. The rationale for this rule is to eliminate recursion in cc; if a subject of type a can create a subject of type b, none of the descendents of the subject can create a subject of type a. This simplifies the analysis without unduly limiting the applicability of the model.

Let **A** be a subject of type $a = \tau(\mathbf{A})$ and let **B** be an entity of type $b = \tau(\mathbf{B})$. The *create-rule* $cr(a, b)$ specifies the tickets introduced when a subject of type a creates an entity of type b.

If **B** is an object, the rule specifies the tickets for **B** to be placed in $dom(\mathbf{A})$ as a result of the creation. Only inert tickets can be created, so $cr(a, b) \subseteq \{ b/r{:}c \in RI \}$, and **A** gets $\mathbf{B}/r{:}c$ if and only if $b/r{:}c \in cr(a, b)$.

If **B** is a subject, the rule also specifies that the tickets for **A** be placed in $dom(\mathbf{B})$ as a result of the creation. Assume that types a and b are distinct. Let $cr_p(a, b)$ be the set of tickets the creation adds to $dom(\mathbf{A})$, and let $cr_c(a, b)$ be the set of tickets the creation adds to $dom(\mathbf{B})$. Then **A** gets the ticket $\mathbf{B}/r{:}c$ if $\mathbf{B}/r{:}c \in cr_p(a, b)$ and **B** gets the ticket $\mathbf{A}/r{:}c$ if $a/r{:}c \in cr_c(a, b)$. We write $cr(a, b) = \{ a/r{:}c, b/r{:}c \mid r{:}c \in R \}$. If the types a and b are not distinct, then do the types refer to the creator or the created? To avoid this ambiguity, if $a = b$, we define $self/r{:}c$ to be tickets for the creator and $a/r{:}c$ to be tickets for the created, and we say that $cr(a, a) = \{ a/r{:}c, self/r{:}c \mid r{:}c \in R \}$. $cr_p(a, b)$ and $cr_c(a, b)$ are subsets of $cr(a, a)$, as before.

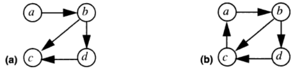


Figure 3–5 The rule of acyclic creates. (a) The *can-create* relation $cc = \{ (a, b), (b, c), (b, d), (d, c) \}$. Because there are no cycles in the graph, cc satisfies the rule of acyclic creates. (b) Same as (a), except that the *can-create* relation is $cc' = cc \cup \{ (c, a) \}$, which induces a cycle; hence, cc' does not follow the rule of acyclic creates.

Sandhu [870] has shown that the flow function requires $O(|T \times R|\ |SUB^h|^3)$, and hence the computation's time complexity is polynomial in the number of subjects in the system.

This definition allows us to sharpen our intuition of what a "maximal state" is (and will ultimately enable us to define that state formally). Intuitively, a maximal state maximizes flow between all pairs of subjects. Call the maximal state * and the flow function corresponding to this state $flow^*$; then if a ticket is in $flow^*(\mathbf{X}, \mathbf{Y})$, there exists a sequence of operations that can copy the ticket from **X** to **Y**. This brings up two questions. First, is a maximal state unique? Second, does every system have a maximal state?

We first formally define the notion of maximal state using a relation named $\leq_0$.

Definition 3–18. The relation $g \leq_0 h$ is true if and only if, for all pairs of subjects **X** and **Y** in SUB^0, $flow^g(\mathbf{X}, \mathbf{Y}) \subseteq flow^h(\mathbf{X}, \mathbf{Y})$. If $g \leq_0 h$ and $h \leq_0 g$, g and h are equivalent.

In other words, the relation $\leq_0$ induces a set of equivalence classes on the set of derivable states.

Definition 3–19. For a given system, a state m is maximal if and only if $h \leq_0 m$ for every derivable state h.

In a maximal state, the flow function contains all the tickets that can be transferred from one subject to another. Hence, all maximal states are in the same equivalence class and thus are equivalent. This answers our first question.

To show that every system has a maximal state, we first show that for any state in a finite collection of derivable states, there is a maximal state.

Lemma 3–3. Given an arbitrary finite collection H of derivable states, there exists a derivable state m such that, for all $h \in H$, $h \leq_0 m$.

Proof By induction on $|H|$.

Basis. Take $H = \varnothing$ and m to be the initial state. The claim is trivially true.

Induction hypothesis. The claim holds when $|H| = n$.

Induction step. Let $|H'| = n + 1$, where $H' = G \cup \{ h \}$; thus, $|G| = n$. Choose $g \in G$ such that, for every state $x \in G$, $x \leq_0 g$; such a state's existence is guaranteed by the induction hypothesis.

Consider the states g and h, defined above. Each of these states is established by a history. Let M be an interleaving of these histories that preserves the relative order of transitions with respect to g and h, and with only the first create operation of duplicate create operations in the two histories. Let M attain state m. If either $path^g(\mathbf{X}, \mathbf{Y})$ for $\mathbf{X}, \mathbf{Y} \in SUB^g$ or $path^h(\mathbf{X}, \mathbf{Y})$ for $\mathbf{X}, \mathbf{Y} \in SUB^h$, then $path^m(\mathbf{X}, \mathbf{Y})$, as g and h are ancestor states of m and SPM is monotonic.

Thus, $g \leq_0 m$ and $h \leq_0 m$, so m is a maximal state in H'. This concludes the induction step and the proof.

Take one state from each equivalence class of derivable states. To see that this is finite, consider each pair of subjects in SUB^0. The flow function's range is $2^{T \times R}$, so that function can take on at most $2^{|T \times R|}$ values. Given that there are $|SUB^0|^2$ pairs of subjects in the initial state, there can be at most $2^{|T \times R|}|SUB^0|^2$ distinct equivalence classes.

Theorem 3–15. There exists a maximal state * for every system.

Proof Take K to be the collection of derivable states that contains exactly one state from each equivalence class of derivable states. From above, this set is finite. The theorem follows from Lemma 3–3.

In this model, the safety question now becomes: *Is it possible to have a derivable state with* $\boldsymbol{X}/r{:}c$ *in* $dom(\boldsymbol{A})$*, or does there exist a subject* $\boldsymbol{B}$ *with ticket* $\boldsymbol{X}/rc$ *in the initial state or which can demand* $\boldsymbol{X}/rc$ *and* $\tau(\boldsymbol{X})/r{:}c$ *in* $flow^*(\boldsymbol{B}, \boldsymbol{A})$*?*

To answer this question, we need to construct a maximal state and test. Generally, this will require the creation of new subjects. In the general case, this is undecidable. But in special cases, this question is decidable. We now consider an important case—that of acyclic attenuating schemes—and determine how to construct the maximal state.

Consider a state h. Intuitively, generating a maximal state m from h will require all three types of operations. Define u to be a state corresponding to h but with a minimal number of new entities created such that m can be derived from u without any create operations. (That is, begin in state h. Use create operations to create as few new entities as possible such that state m can be derived from the new state *after* the entities are created. The state after the entities are created, but before any other operations occur, is u.) For example, if in the history from h to m, subject **X** creates two entities of type y, in u there would be only one entity of type y. That entity would act as a surrogate for the two entities that **X** created. Because m can be derived from u in polynomial time, if u can be created by adding to h a finite number of subjects, the safety question is decidable in polynomial time for such a system.

We now make this formal.

Definition 3–20. ([870], p.425) Given any initial state 0 of an acyclic attenuating scheme, the *fully unfolded state u* is the state derived by the following algorithm.

```
(* delete any loops so it's loop-free *)
cc′ = cc − { (a, a) | a ∈ TS }
(* mark all subjects as unfolded *)
for X ∈ SUB⁰ do
        folded = folded ∪ { X }
```

```
(* if anything is folded, it has to be unfolded *)
while folded ≠ ∅ do begin
        (* subject X is going to be unfolded *)
        folded = folded - { x }
        (* for each type X can create, create one entity of *)
        (* that type and mark it as folded; this will force *)
        (* the new entity to be unfolded *)
        for y ∈ TS do begin
            if cc´(τ(X), y) then
                X creates Y of type y
                (* system is in state g here *)
                if y ∈ SUB^g then
                    folded = folded ∪ { X }
        end
end
(* now account for the loops; the system is in state h here *)
for X ∈ SUB^h do
        if cc(τ(X), τ(X)) then
            X creates Y of type τ(X)
(* currently in desired state u *)
```

The **while** loop will terminate because the system is acyclic and attenuating, hence the types of the created entities must all be different—and *TS* is a finite set.

Definition 3–21. Given any initial state of an acyclic attenuating scheme, for every derivable state *h* define the *surrogate function* σ:$ENT^h \rightarrow ENT^h$ by

1. $\sigma(\mathbf{X}) = \mathbf{X}$ if **X** in ENT^0
2. $\sigma(\mathbf{X}) = \sigma(\mathbf{Y})$ if **Y** creates **X** and $\tau(\mathbf{Y}) = \tau(\mathbf{X})$
3. $\sigma(\mathbf{X}) = \tau(\mathbf{Y})$-surrogate of $\sigma(\mathbf{Y})$ if **Y** creates **X** and $\tau(\mathbf{Y}) \neq \tau(\mathbf{X})$

It is easy to show that $\tau(\sigma(\mathbf{A})) = \tau(\mathbf{A})$.

If $\tau(\mathbf{X}) = \tau(\mathbf{Y})$, then $\sigma(\mathbf{X}) = \sigma(\mathbf{Y})$. If $\tau(\mathbf{X}) \neq \tau(\mathbf{Y})$, then in the construction of *u*, $\sigma(\mathbf{X})$ creates $\sigma(\mathbf{Y})$ (see the while loop of Definition 3–20). Also, in this construction, $\sigma(\mathbf{X})$ creates entities $\mathbf{X}'$ of type $\tau(\mathbf{X}) = \tau(\sigma(\mathbf{X}))$ (see the last for loop of Definition 3–20). So, by Definition 3–14, we have the following lemma.

Lemma 3–4. For a system with an acyclic attenuating scheme, if **X** creates **Y**, then tickets that would be introduced by pretending that $\sigma(\mathbf{X})$ creates $\sigma(\mathbf{Y})$ are in $dom^u(\sigma(\mathbf{X}))$ and $dom^u(\sigma(\mathbf{Y}))$.

Now, let *H* be a legal history that derives a state *h* from the initial state of an acyclic attenuating system. Without loss of generality, we may assume that *H*'s operations are ordered such that all create operations come first, followed by all demand operations, followed by all copy operations. Replace the transitions in H as follows, while preserving their relative order.

1. Delete all create operations.
2. Replace "**X** demands **Y**/r:c" with "σ(**X**) demands σ(**Y**)/r:c."
3. Replace "**Z** copies **X**/r:c from **Y**" with "σ(**Z**) copies σ(**X**)/r:c from σ(**Y**)."

Call the new history G. Then:

Lemma 3–5. Every transition in G is legal, and if $\mathbf{X}/r{:}c \in dom^h(\mathbf{Y})$, then $\sigma(\mathbf{X})/r{:}c \in dom^g(\sigma(\mathbf{Y}))$.

Proof By induction on the number of copy operations in H.

Basis. Assume that H consists only of create and demand operations. Then G consists only of demand operations. By construction, and because σ preserves type, every demand operation in G is legal. Furthermore, $\mathbf{X}/r{:}c$ can appear in $dom^h(\mathbf{Y})$ in one of three ways. If $\mathbf{X}/r{:}c \in dom^0(\mathbf{Y})$, then $\mathbf{X}, \mathbf{Y} \in ENT^0$ and $\sigma(\mathbf{X})/r{:}c \in dom^g(\sigma(\mathbf{Y}))$ trivially holds. If a create operation in H put $\mathbf{X}/r{:}c \in dom^h(\mathbf{Y})$, $\sigma(\mathbf{X})/r{:}c \in dom^g(\sigma(\mathbf{Y}))$ by Lemma 3–4. And if a demand operation put $\mathbf{X}/r{:}c \in dom^h(\mathbf{Y})$, then $\sigma(\mathbf{X})/r{:}c \in dom^g(\sigma(\mathbf{Y}))$ follows from the corresponding demand operation in G. This establishes both parts of the claim.

Induction hypothesis. Assume that the claim holds for all histories with k copy operations, and consider a history H with $k + 1$ copy operations. Let H' be the initial sequence of H composed of k copy operations, and let h' be the state derived from H'.

Induction step. Let G' be the sequence of modified operations corresponding to H'. By the induction hypothesis, G' is a legal history. Let g' be the state derived from G'. Suppose the final operation of H is "**Z** copies **X**/r:c from **Y**." By construction of G, the final operation of G is "σ(**Z**) copies σ(**X**)/r:c from σ(**Y**)." Now, h differs from h' by at most $\mathbf{X}/r{:}c \in dom^h(\mathbf{Z})$. However, the construction causes the final operation of G to be $\sigma(\mathbf{X})/r{:}c \in dom^h(\sigma(\mathbf{Z}))$, proving the second part of the claim.

Because H' is legal, for H to be legal the following conditions must hold.

1. $\mathbf{X}/rc \in dom^{h'}(\mathbf{Y})$
2. $link_i^{h'}(\mathbf{Y}, \mathbf{Z})$
3. $\tau(\mathbf{X}/r{:}c) \in f_i(\tau(\mathbf{Y}), \tau(\mathbf{Z}))$

The induction hypothesis, the first two conditions above, and $\mathbf{X}/r{:}c \in dom^{h'}(\mathbf{Y})$ mean that $\sigma(\mathbf{X})/rc \in dom^{g'}(\sigma(\mathbf{Y}))$ and $link_i^{g'}(\sigma(\mathbf{Y}), \sigma(\mathbf{Z}))$. Because σ preserves type, the third condition and the induction hypothesis imply

$\tau(\sigma(\mathbf{X})/r{:}c) \in f_i(\tau(\sigma(\mathbf{Y})), \tau(\sigma(\mathbf{Z})))$. G' is legal, by the induction hypothesis; so, by these conditions, G is legal. This establishes the lemma.

Corollary 3–3. For every i, if $link_i^h(\mathbf{X}, \mathbf{Y})$, then $link_i^g(\sigma(\mathbf{X}), \sigma(\mathbf{Y}))$.

We can now present the following theorem.

Theorem 3–16. For a system with an acyclic attenuating scheme, for every history H that derives h from the initial state, there exists a history G without create operations that derives g from the fully unfolded state u such that

$$(\forall\, \mathbf{X}, \mathbf{Y} \in SUB^h)[flow^h(\mathbf{X}, \mathbf{Y}) \subseteq flow^g(\sigma(\mathbf{X}), \sigma(\mathbf{Y}))]$$

Proof It suffices to show that for every $path^h$ from $\mathbf{X}$ to $\mathbf{Y}$ there is a $path^g$ from $\sigma(\mathbf{X})$ to $\sigma(\mathbf{Y})$ for which $cap(path^h(\mathbf{X}, \mathbf{Y})) = cap(path^g(\sigma(\mathbf{X}), \sigma(\mathbf{Y}))$. Induct on the number of links.

Basis. Let the length of the $path^h$ from $\mathbf{X}$ to $\mathbf{Y}$ be 1. By Definition 3–16, then, $link_i^h(\mathbf{X}, \mathbf{Y})$, so $link_i^g(\sigma(\mathbf{X}), \sigma(\mathbf{Y}))$ by Corollary 3–3. Because σ preserves type, $cap(path^h(\mathbf{X}, \mathbf{Y})) = cap(path^g(\sigma(\mathbf{X}), \sigma(\mathbf{Y}))$, verifying the claim.

Induction hypothesis. Assume that the claim holds for every $path^h$ of length k.

Induction step. Consider a $path^h$ from $\mathbf{X}$ to $\mathbf{Y}$ of length $k + 1$. Then there exists an entity $\mathbf{Z}$ with a $path^h$ from $\mathbf{X}$ to $\mathbf{Z}$ of length k, and $link_j^h(\mathbf{Z}, \mathbf{Y})$. By the induction hypothesis, there is a $path^g$ from $\sigma(\mathbf{X})$ to $\sigma(\mathbf{Z})$ with the same capacity as the $path^h$ from $\mathbf{X}$ to $\mathbf{Z}$. By Corollary 3–3, we have $link_j^g(\sigma(\mathbf{Z}), \sigma(\mathbf{Y}))$. Because σ preserves type, there is a $path^g$ from $\mathbf{X}$ to $\mathbf{Y}$ with $cap(path^h(\mathbf{X}, \mathbf{Y})) = cap(path^g(\sigma(\mathbf{X}), \sigma(\mathbf{Y}))$, proving the induction step and therefore the theorem.

Thus, any history derived from an initial state u can be simulated by a corresponding history applied to the fully unfolded state v derived from u. The maximal state corresponding to v is $\#u$; the history deriving this state has no creates. From Theorem 3–16, for every history that derives h from the initial state,

$$(\forall\, \mathbf{X}, \mathbf{Y} \in SUB^h)[flow^h(\mathbf{X}, \mathbf{Y}) \subseteq flow^{\#u}(\sigma(\mathbf{X}), \sigma(\mathbf{Y}))]$$

For $\mathbf{X} \in SUB^0$, $\sigma(\mathbf{X}) = \mathbf{X}$; therefore, $(\forall\, \mathbf{X}, \mathbf{Y} \in SUB^0)[flow^h(\mathbf{X}, \mathbf{Y}) \subseteq flow^{\#u}(\mathbf{X}, \mathbf{Y})]$. This demonstrates the following corollary.

Corollary 3–4. The state $\#u$ is a maximal state for a system with an acyclic attenuating scheme.

Not only is $\#u$ derivable from u, it is derivable in time polynomial with respect to $|SUB^u|$ (and therefore to $|SUB^0|$). Moreover, the straightforward algorithm for computing $flow^{\#u}$ will be exponential in $|TS|$ in the worst case. This means that for acyclic attenuating schemes, the safety question is decidable.

The child also has a rule of the form

$$cr_C(\tau(\mathbf{X}_1), \ldots, \tau(\mathbf{X}_n), \tau(\mathbf{Y})) = \mathbf{Y}/R_3 \cup \mathbf{X}_1/R_{4,1} \cup \ldots \cup \mathbf{X}_n/R_{4,n}$$

These rules are analogous to the single-parent creation rules, but with one for each parent.

EXAMPLE: To expand on this concept, let's revisit Anna and Bill's situation. Anna and Bill are equals, so for modeling purposes they have the same type a. The proxy is of type p; because the proxy has delegated authority, a and p may be different. We model the rights that Anna and Bill have over the proxy by the right x in R. Thus:

$$cc(a, a) = p$$
$$cr_{\text{Anna}}(a, a, p) = cr_{\text{Bill}}(a, a, p) = \varnothing$$
$$cr_{\text{proxy}}(a, a, p) = \text{Anna}/x \cup \text{Bill}/x$$

Then the proxy can use the right x to transfer whatever set of privileges the proxy requires.

Considering two-parent joint creation operations is sufficient for modeling purposes. To demonstrate this, we show how the two-parent joint creation operation can implement a three-parent joint creation operation.

Let $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$ be three subjects; they will create a (child) entity $\mathbf{C}$. With a three-parent joint creation operation, *can-create* will be

$$cc(\tau(\mathbf{P}_1), \tau(\mathbf{P}_2), \tau(\mathbf{P}_3)) = Z \subseteq T$$

and the type of the child is $\tau(\mathbf{C}) \in T$. The creation rules are

$$cr_{\mathbf{P}1}(\tau(\mathbf{P}_1), \tau(\mathbf{P}_2), \tau(\mathbf{P}_3)) = \mathbf{C}/R_{1,1} \cup \mathbf{P}_1/R_{2,1}$$
$$cr_{\mathbf{P}2}(\tau(\mathbf{P}_1), \tau(\mathbf{P}_2), \tau(\mathbf{P}_3)) = \mathbf{C}/R_{1,2} \cup \mathbf{P}_2/R_{2,2}$$
$$cr_{\mathbf{P}3}(\tau(\mathbf{P}_1), \tau(\mathbf{P}_2), \tau(\mathbf{P}_3)) = \mathbf{C}/R_{1,3} \cup \mathbf{P}_3/R_{2,3}$$
$$cr_{\mathbf{C}}(\tau(\mathbf{P}_1), \tau(\mathbf{P}_2), \tau(\mathbf{P}_3)) = \mathbf{C}/R_3 \cup \mathbf{P}_1/R_{4,1} \cup \mathbf{P}_2/R_{4,2} \cup \mathbf{P}_3/R_{4,3}$$

Our demonstration requires that we use the two-parent joint creation rule, not the three-parent rule. At the end of the demonstration, the parents and the child should have exactly the same tickets for one another. We will create additional entities and types, but they cannot interact with any other entities (in effect, they do not exist for the rest of the entities). Finally, if the creation fails, the parents get no new tickets.

For convenience, and to simplify the notation, we assume that the parents and child are all of different types.

Define four new entities $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, and $\mathbf{S}$; each $\mathbf{A}_i$, of type $a_i = \tau(\mathbf{A}_i)$, will act as an agent for the corresponding parent $\mathbf{P}_i$, and $\mathbf{S}$, of type $s = \tau(\mathbf{S})$, will act as an agent for the child. Let the type t represent parentage—that is, an entity with the ticket **X**/*t* has **X** as a parent. Again, without loss of generality, we assume that a_1, a_2, a_3, *s*, and *t* are all new types.

During the construction, each agent will act as a surrogate for its parent; this agent obtains tickets on behalf of the parent, and only after the child is created does the agent give the parent the ticket. That way, if the construction fails, the parent has no new tickets.

Augment the *can-create* rules as follows:

$$cc(p_1) = a_1$$
$$cc(p_2, a_1) = a_2$$
$$cc(p_3, a_2) = a_3$$
$$cc(a_3) = s$$
$$cc(s) = c$$

These rules enable the parents to create the agents. The final agent can create the agent for the child, which subsequently creates the child. Note that the second agent has two parents ($\mathbf{P}_2$ and $\mathbf{A}_1$), as does the third agent ($\mathbf{P}_3$ and $\mathbf{A}_2$); these rules are the two-parent joint creation operation.

On creation, the *creation* rules dictate the new tickets given to the parent and the child. The following rules augment the existing rules.

$$\begin{array}{ll}
cr_P(p_1, a_1) = \varnothing & cr_C(p_1, a_1) = p_1/Rtc \\
cr_{Pfirst}(p_2, a_1, a_2) = \varnothing & \\
cr_{Psecond}(p_2, a_1, a_2) = \varnothing & cr_C(p_2, a_1, a_2) = p_2/Rtc \cup a_1/tc \\
cr_{Pfirst}(p_3, a_2, a_3) = \varnothing & \\
cr_{Psecond}(p_3, a_2, a_3) = \varnothing & cr_C(p_3, a_2, a_3) = p_3/Rtc \cup a_2/tc \\
cr_P(a_3, s) = \varnothing & cr_C(a_3, s) = a_3/tc \\
cr_P(s, c) = \mathbf{C}/Rtc & cr_C(s, c) = c/R_3t
\end{array}$$

Here, cr_{Pfirst} and $cr_{Psecond}$ indicate the tickets given to the first and second parents, respectively.

The link predicates indicate over which links rights can flow; essentially, no tickets can flow to the parents until the child is created. The following links restrain flow to the parents by requiring each agent to have its own "parent" right.

$$link_1(\mathbf{A}_1, \mathbf{A}_2) = \mathbf{A}_1/t \in dom(\mathbf{A}_2) \wedge \mathbf{A}_2/t \in dom(\mathbf{A}_2)$$
$$link_1(\mathbf{A}_2, \mathbf{A}_3) = \mathbf{A}_2/t \in dom(\mathbf{A}_3) \wedge \mathbf{A}_3/t \in dom(\mathbf{A}_3)$$
$$link_2(\mathbf{S}, \mathbf{A}_3) = \mathbf{A}_3/t \in dom(\mathbf{S}) \wedge \mathbf{C}/t \in dom(\mathbf{C})$$
$$link_3(\mathbf{A}_1, \mathbf{C}) = \mathbf{C}/t \in dom(\mathbf{A}_1)$$
$$link_3(\mathbf{A}_2, \mathbf{C}) = \mathbf{C}/t \in dom(\mathbf{A}_2)$$
$$link_3(\mathbf{A}_3, \mathbf{C}) = \mathbf{C}/t \in dom(\mathbf{A}_3)$$
$$link_4(\mathbf{A}_1, \mathbf{P}_1) = \mathbf{P}_1/t \in dom(\mathbf{A}_1) \wedge \mathbf{A}_1/t \in dom(\mathbf{A}_1)$$
$$link_4(\mathbf{A}_2, \mathbf{P}_2) = \mathbf{P}_2/t \in dom(\mathbf{A}_2) \wedge \mathbf{A}_2/t \in dom(\mathbf{A}_2)$$
$$link_4(\mathbf{A}_3, \mathbf{P}_3) = \mathbf{P}_3/t \in dom(\mathbf{A}_3) \wedge \mathbf{A}_3/t \in dom(\mathbf{A}_3)$$

The filter functions dictate which tickets are copied from one entity to another:

$f_1(a_2, a_1) = a_1/t \cup c/Rtc$
$f_1(a_3, a_2) = a_2/t \cup c/Rtc$
$f_2(s, a_3) = a_3/t \cup c/Rtc$
$f_3(a_1, c) = p_1/R_{4,1}$
$f_3(a_2, c) = p_2/R_{4,2}$
$f_3(a_3, c) = p_3/R_{4,3}$
$f_4(a_1, p_1) = c/R_{1,1} \cup p_1/R_{2,1}$
$f_4(a_2, p_2) = c/R_{1,2} \cup p_2/R_{2,2}$
$f_4(a_3, p_3) = c/R_{1,3} \cup p_3/R_{2,3}$

Now we begin the construction. The creations proceed in the obvious order; after all are completed, we have

- $\mathbf{P}_1$ has no relevant tickets.
- $\mathbf{P}_2$ has no relevant tickets.
- $\mathbf{P}_3$ has no relevant tickets.
- $\mathbf{A}_1$ has $\mathbf{P}_1/Rtc$.
- $\mathbf{A}_2$ has $\mathbf{P}_2/Rtc \cup \mathbf{A}_1/tc$.
- $\mathbf{A}_3$ has $\mathbf{P}_3/Rtc \cup \mathbf{A}_2/tc$.
- $\mathbf{S}$ has $\mathbf{A}_3/tc \cup \mathbf{C}/Rtc$.
- $\mathbf{C}$ has $\mathbf{C}/R_3$.

We now apply the links and filter functions to copy rights. The only link predicate that is true is $link_2(\mathbf{S}, \mathbf{A}_3)$, so we apply f_2; then $\mathbf{A}_3$'s set of tickets changes, as follows:

- $\mathbf{A}_3$ has $\mathbf{P}_3/Rtc \cup \mathbf{A}_2/tc \cup \mathbf{A}_3/t \cup \mathbf{C}/Rtc$.

Now $link_1(\mathbf{A}_3, \mathbf{A}_2)$ is true, so applying f_1 yields

- $\mathbf{A}_2$ has $\mathbf{P}_2/Rtc \cup \mathbf{A}_1/tc \cup \mathbf{A}_2/t \cup \mathbf{C}/Rtc$.

Now $link_1(\mathbf{A}_2, \mathbf{A}_1)$ is true, so applying f_1 again yields

- $\mathbf{A}_1$ has $\mathbf{P}_1/Rtc \cup \mathbf{A}_1/t \cup \mathbf{C}/Rtc$.

At this point, all $link_3$s in this construction hold, so

- $\mathbf{C}$ has $\mathbf{C}/R_3 \cup \mathbf{P}_1/R_{4,1} \cup \mathbf{P}_2/R_{4,2} \cup \mathbf{P}_3/R_{4,3}$.

Then the filter functions associated with $link_4$, all of which are also true, finish the construction:

- $\mathbf{P}_1$ has $\mathbf{C}/R_{1,1} \cup \mathbf{P}_1/R_{2,1}$.
- $\mathbf{P}_2$ has $\mathbf{C}/R_{1,2} \cup \mathbf{P}_2/R_{2,2}$.
- $\mathbf{P}_3$ has $\mathbf{C}/R_{1,3} \cup \mathbf{P}_3/R_{2,3}$.

This completes the construction. As required, it adds no tickets to $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$, and $\mathbf{C}$ except those that would be added by the three-parent joint creation operation. The intermediate entities, being of unique types, can have no effect on other entities. Finally, if the creation of $\mathbf{C}$ fails, no tickets can be added to $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$ because none of the link predicates in this construction is true; hence, no filter functions apply.

Generalizing this construction to n parents leads to the following theorem.

Theorem 3–17. [19] The two-parent joint creation operation can implement an n-parent joint creation operation with a fixed number of additional types and rights, and augmentations to the link predicates and filter functions.

A logical question is the relationship between ESPM and HRU; Ammann and Sandhu show that the following theorem holds.

Theorem 3–18. [19] Monotonic ESPM and the monotonic HRU model are equivalent.

Furthermore, the safety analysis is similar to that of SPM; the only difference is in the definition of the state function σ. The corresponding function σ' takes the joint creation operation into account; given this, the nature of the unfolding algorithm is roughly analogous to that of SPM. This leads to the equivalent of Theorem 3–16:

Theorem 3–19. [19] For an ESPM system with an acyclic attenuating scheme, for every history H that derives h from the initial state there exists a history G without create operations that derives g from the fully unfolded state u such that

$$(\forall\, \mathbf{X}, \mathbf{Y} \in SUB^h)[flow^h(\mathbf{X}, \mathbf{Y}) \subseteq flow^g(\sigma'(\mathbf{X}), \sigma'(\mathbf{Y}))]$$

Because the proof is analogous to that of Theorem 3–16, we omit it.

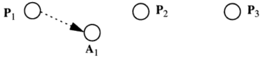
What is the benefit of this alternative representation? If SPM and ESPM model the same systems, the addition of n-parent joint creation operations is not at all interesting. But if ESPM can represent systems that SPM cannot, the addition is very interesting. More generally, how can we compare different models?

3.5.3 Simulation and Expressiveness

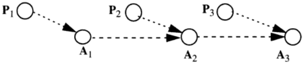
Ammann, Sandhu, and Lipton [21] use a graph-based representation to compare different models. An abstract machine represents an access control model; as usual, that machine has a set of states and a set of transformations for moving from one state to another. A directed graph represents a state of this machine. A vertex is an entity; it has an associated type that is static. Each edge corresponds to a right and, like a vertex, has a static type determined on creation. The source of the edge has some right(s) over the target. The allowed operations are as follows.

1. *Initial state operations*, which simply create the graph in a particular state
2. *Node creation operations*, which add new vertices and edges with those vertices as targets
3. *Edge adding operations*, which add new edges between existing vertices

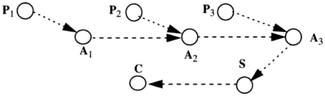
As an example, we simulate the three-parent joint creation operation with two-parent joint creation operations. As before, nodes $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$ are the parents; they create a new node **C** of type c with edges of type e. First, $\mathbf{P}_1$ creates $\mathbf{A}_1$, which is of type a, and an edge from $\mathbf{P}_1$ to $\mathbf{A}_1$ of type e'. Both a and e' are used only in this construction.



Then $\mathbf{A}_1$ and $\mathbf{P}_2$ create a new node $\mathbf{A}_2$, which is of type a, and $\mathbf{A}_2$ and $\mathbf{P}_3$ create a new node $\mathbf{A}_3$, with type a, and edges of type e' as indicated:



Next, $\mathbf{A}_3$ creates a new node **S**, which is of type a, which in turn creates a new node **C**, of type c:



Because edges can be added only by using the two-parent joint creation operation in scheme *A*, all nodes in scheme *A* have even numbers of incoming edges. But given the edge adding rule in scheme *B*, because we can add an edge from $\mathbf{X}_2$ to $\mathbf{Y}$, we can also add an edge from $\mathbf{X}_3$ to $\mathbf{Y}$. Thus, there is a state in scheme *B* containing a node with three incoming edges. Scheme *A* cannot enter this state. Furthermore, because there is no remove rule and only one edge type, scheme *B* cannot transition from this state to a state in which $\mathbf{Y}$ has an even number of incoming edges. Hence, scheme *B* has reached a state not corresponding to any state in scheme *A*, and from which no state corresponding to a state in scheme *A* can be reached. Thus, scheme *B* cannot simulate scheme *A*, and so model *N* is less expressive than model *M*.

Given these definitions, Ammann, Lipton, and Sandhu prove the following theorem.

Theorem 3–20. [21] Monotonic single-parent models are less expressive than monotonic multiparent models.

Proof Begin with scheme *A* in the preceding example. We show by contradiction that this scheme cannot be simulated by any monotonic scheme *B* with only a single-parent creation operation. (The example does not show this because we are removing the requirement that scheme *B* begin in the same initial state as scheme *A*.)

Consider a scheme *B* that simulates scheme *A*. Let nodes $\mathbf{X}_1$ and $\mathbf{X}_2$ in *A* create node $\mathbf{Y}_1$ with edges from $\mathbf{X}_1$ and $\mathbf{X}_2$ to $\mathbf{Y}_1$. Then in scheme *B* there is a node $\mathbf{W}$ that creates $\mathbf{Y}_1$ with a single incoming edge from $\mathbf{W}$. The simulation must also use edge adding operations to add edges from $\mathbf{X}_1$ to $\mathbf{Y}_1$ and from $\mathbf{X}_2$ to $\mathbf{Y}_1$ (assuming that $\mathbf{W} \neq \mathbf{X}_1$ and $\mathbf{W} \neq \mathbf{X}_2$).

Let $\mathbf{W}$ invoke the single-parent creation operation twice more to create nodes $\mathbf{Y}_2$ and $\mathbf{Y}_3$ and use the edge adding rules to add edges from $\mathbf{X}_1$ to $\mathbf{Y}_1$, $\mathbf{Y}_2$, and $\mathbf{Y}_3$ and from $\mathbf{X}_2$ to $\mathbf{Y}_1$, $\mathbf{Y}_2$, and $\mathbf{Y}_3$. The resulting state clearly corresponds to a state in scheme *A*.

Because scheme *A* has exactly one node type, $\mathbf{Y}_1$, $\mathbf{Y}_2$, and $\mathbf{Y}_3$ are indistinguishable as far as the application of the node creation and edge adding rules is concerned. So proceed as in the example above: in scheme *A*, let $\mathbf{Y}_1$ and $\mathbf{Y}_2$ create $\mathbf{Z}$. In the simulation, without loss of generality, let $\mathbf{Y}_1$ create $\mathbf{Z}$ using a single-parent creation operation. Then scheme *B* uses an edge adding operation to add an edge from $\mathbf{Y}_2$ to $\mathbf{Z}$—but that same edge adding rule can be used to add one more edge into $\mathbf{Z}$, from $\mathbf{Y}_3$. Thus, there are three edges coming into $\mathbf{Z}$, which (as we saw earlier) is a state that scheme *A* cannot reach, and from which no future state in scheme *B* that corresponds to a state in scheme *A* can be reached. Hence, scheme *B* does not simulate scheme *A*, which contradicts the hypothesis.

Thus, no such scheme *B* can exist.

This theorem answers the question posed earlier: because ESPM has a multi-parent joint creation operation and SPM has a single-parent creation operation, ESPM is indeed more expressive than SPM.

3.5.4 Typed Access Matrix Model

The strengths of SPM and ESPM appear to derive from the notion of "types." In particular, ESPM and HRU are equivalent, but the safety properties of ESPM are considerably stronger than those of HRU. Sandhu expanded the access control matrix model by adding a notion of "type" and revisiting the HRU results. This model, called the *Typed Access Matrix (TAM) Model* [875], has safety properties similar to those of ESPM and supports the notion that types are critical to the safety problem's analysis.

TAM augments the definitions used in the access control matrix model by adding types.

Definition 3–26. There is a finite set of types T, containing a subset of types TS for subjects.

The type of an entity is fixed when the entity is created (or in the initial state) and remains fixed throughout the lifetime of the model. The notion of *protection state* is similarly augmented.

Definition 3–27. The *protection state* of a system is (S, O, τ, A), where S is the set of subjects, O is the set of objects, A is the access control matrix, and $\tau{:}O \rightarrow T$ is a *type function* that specifies the type of each object. If $X \in S$, then $\tau(X) \in TS$, and if $X \in O$, then $\tau(X) \in T - TS$.

The TAM primitive operations are the same as for the access control matrix model, except that the *create* operations are augmented with types.

1. Precondition: $s \notin S$
   Primitive command: **create subject** s **of type** ts
   Postconditions: $S' = S \cup \{ s \}$, $O' = O \cup \{ s \}$,
   $(\forall y \in O)[\tau'(y) = \tau(y)]$, $\tau'(s) = ts$,
   $(\forall y \in O')[a'[s, y] = \varnothing]$, $(\forall x \in S')[a'[x, s] = \varnothing]$,
   $(\forall x \in S)(\forall y \in O)[a'[x, y] = a[x, y]]$

   In other words, this primitive command creates a new subject s. Note that s must not exist as a subject *or object* before this command is executed.
2. Precondition: $o \notin O$
   Primitive command: **create object** o **of type** to
   Postconditions: $S' = S$, $O' = O \cup \{ o \}$,
   $(\forall y \in O)[\tau'(y) = \tau(y)]$, $\tau'(o') = to$,
   $(\forall x \in S')[a'[x, o] = ,\varnothing]$, $(\forall x \in S')(\forall y \in O)[a'[x, y] = a[x, y]]$

In other words, this primitive command creates a new object o. Note that o must not exist before this command is executed.

These primitive operations are combined into commands defined as in the access control matrix model. Commands with conditions are called *conditional commands*; commands without conditions are called *unconditional commands*.

Finally, we define the models explicitly.

Definition 3–28. A *TAM authorization scheme* consists of a finite set of rights R, a finite set of types T, and a finite collection of commands. A *TAM system* is specified by a TAM scheme and an initial state.

Definition 3–29. The *MTAM (Monotonic Typed Access Matrix) Model* is the TAM Model without the **delete**, **destroy subject**, and **destroy object** primitive operations.

Definition 3–30. Let $\alpha(x_1 : t_1, \ldots, x_k : t_k)$ be a creating command, where $x_1, \ldots, x_k \in$ O and $\tau(x_1) = t_1, \ldots, \tau(x_k) = t_k$. Then t_i is a *child type* in $\alpha(x_1 : t_1, \ldots, x_k : t_k)$ if any of **create subject** x_i **of type** t_i or **create object** x_i **of type** t_i occurs in the body of $\alpha(x_1 : t_1, \ldots, x_k : t_k)$. Otherwise, t_i is a *parent type* in $\alpha(x_1 : t_1, \ldots, x_k : t_k)$.

From this, we can define the notion of acyclic creations.

Definition 3–31. The *creation graph* of an MTAM scheme is a directed graph with vertex set V and an edge from $u \in V$ to $v \in V$ if and only if there is a creating command in which u is a parent type and v is a child type. If the creation graph is acyclic, the MTAM system is said to be acyclic; otherwise, the MTAM system is said to be cyclic.

As an example, consider the following command, where s and p are subjects and f is an object.

```
command create•havoc(s : u, p : u, f : v, q : w)
    create subject p of type u;
    create object f of type v;
    enter own into a[s, p];
    enter r into a[q, p];
    enter own into a[p, f];
    enter r into a[p, f];
    enter w into a[p, f];
end
```

Here, u and v are child types and u and w are parent types. Note that u is both a parent type and a child type. The creation graph corresponding to the MTAM scheme with the single command *create•havoc* has the edges (u, u), (u, w), (v, u), and (v, w). Thus, this MTAM scheme is cyclic. Were the **create subject** p **of type** u deleted from the command, however, u would no longer be a child type, and the resulting MTAM scheme would be acyclic.

Sandhu has shown that the following theorem is true.

Theorem 3–21. [875] Safety is decidable for systems with acyclic MTAM schemes.

The proof is similar in spirit to the proof of Theorem 3–16.

Furthermore, because MTAM subsumes monotonic mono-operational HRU systems, a complexity result follows automatically:

Theorem 3–22. [875] Safety is *NP*-hard for systems with acyclic MTAM schemes.

However, Sandhu [875] has also developed a surprising result. If all MTAM commands are limited to three parameters, the resulting model (called "ternary MTAM") is equivalent in expressive power to MTAM. However:

Theorem 3–23. [875] Safety for the acyclic ternary MTAM model is decidable in time polynomial in the size of the initial access control matrix.

3.6 Summary

The safety problem is a rich problem that has led to the development of several models and analysis techniques. Some of these models are useful in other contexts. We will return, for example, to both the Take-Grant Protection Model and ESPM later. These models provide insights into the boundary line between decidability and undecidability, which speaks to the degree of generality of analysis. Ultimately, however, security (the analogue of safety) is analyzed for a system or for a class of systems, and the models help us understand when such analysis is tractable and when it is not.

3.7 Research Issues

The critical research issue is the characterization of the class of models for which the safety question is decidable. The SRM results state sufficiency but not necessity. A set of characteristics that are both necessary and sufficient would show exactly what causes the safety problem to become undecidable, which is an open issue.

Related questions involve the expressive power of the various models. The models allow policies to be expressed more succinctly than in the access control matrix model. Can these more sophisticated models express the same set of policies that the access control matrix model can express? Are there other models that are easy to work with yet allow all protection states of interest to be expressed?

3.8 Further Reading

Sandhu and Ganta [877] have explored the effects of allowing testing for the *absence* of rights in an access control matrix (as opposed to testing for the *presence* of rights, which all the models described in this chapter do). Biskup [119] presents some variants on the Take-Grant Protection Model, and Budd [154] analyzes safety properties of *grammatical protection schemes*, which he and Lipton defined earlier [640].

Sandhu has also presented interesting work on the representation of models, and has unified many of them with his *transform* model [873, 874, 878].

3.9 Exercises

1. The proof of Theorem 3–1 states the following: Suppose two subjects s_1 and s_2 are created and the rights in $A[s_1, o_1]$ and $A[s_2, o_2]$ are tested. The same test for $A[s_1, o_1]$ and $A[s_1, o_2] = A[s_1, o_2] \cup A[s_2, o_2]$ will produce the same result. Justify this statement. Would it be true if one could test for the absence of rights as well as for the presence of rights?
2. Devise an algorithm that determines whether or not a system is safe by enumerating all possible states. Is this problem *NP*-complete? Justify your answer.
3. Prove Theorem 3–3. (*Hint*: Use a diagonalization argument to test each system as the set of protection systems is enumerated. Whenever a protection system leaks a right, add it to the list of unsafe protection systems.)
4. Prove or disprove: The claim of Lemma 3–1 holds when **x** is an object.
5. Prove or give a counterexample:
   The predicate *can•share*(α, **x**, **y**, G_0) is true if and only if there is an edge from **x** to **y** in G_0 labeled α, or if the following hold simultaneously.
   a. There is a vertex $s \in G_0$ with an **s**-to-**y** edge labeled α.
   b. There is a subject vertex **x**′ such that **x**′ = **x** or **x**′ initially spans to **x**.
   c. There is a subject vertex **s**′ such that **s**′ = **s** or **s**′ terminally spans to **s**.

image not available

Chapter 4
Security Policies

PORTIA: Of a strange nature is the suit you follow;
Yet in such rule that the Venetian law
Cannot impugn you as you do proceed.
[*To Antonio*.] You stand within his danger, do you not?
—*The Merchant of Venice*, IV, i, 177–180.

A security policy defines "secure" for a system or a set of systems. Security policies can be informal or highly mathematical in nature. After defining a security policy precisely, we expand on the nature of "trust" and its relationship to security policies. We also discuss different types of policy models.

4.1 Security Policies

Consider a computer system to be a finite-state automaton with a set of transition functions that change state. Then:

Definition 4–1. A *security policy* is a statement that partitions the states of the system into a set of *authorized*, or *secure*, states and a set of *unauthorized*, or *nonsecure*, states.

A security policy sets the context in which we can define a secure system. What is secure under one policy may not be secure under a different policy. More precisely:

Definition 4–2. A *secure system* is a system that starts in an authorized state and cannot enter an unauthorized state.

Consider the finite-state machine in Figure 4–1. It consists of four states and five transitions. The security policy partitions the states into a set of authorized states $A = \{ s_1, s_2 \}$ and a set of unauthorized states $UA = \{ s_3, s_4 \}$. This system is not

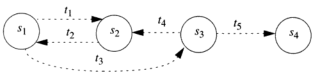


Figure 4–1 A simple finite-state machine. In this example, the authorized states are s_1 and s_2.

secure, because regardless of which authorized state it starts in, it can enter an unauthorized state. However, if the edge from s_1 to s_3 were not present, the system would be secure, because it could not enter an unauthorized state from an authorized state.

Definition 4–3. A *breach of security* occurs when a system enters an unauthorized state.

We informally discussed the three basic properties relevant to security in Section 1.1. We now define them precisely.

Definition 4–4. Let X be a set of entities and let I be some information. Then I has the property of *confidentiality* with respect to X if no member of X can obtain information about I.

Confidentiality implies that information must not be disclosed to some set of entities. It may be disclosed to others. The membership of set X is often implicit—for example, when we speak of a document that is confidential. Some entity has access to the document. All entities not authorized to have such access make up the set X.

Definition 4–5. Let X be a set of entities and let I be some information or a resource. Then I has the property of *integrity* with respect to X if all members of X trust I.

This definition is deceptively simple. In addition to trusting the information itself, the members of X also trust that the conveyance and storage of I do not change the information or its trustworthiness (this aspect is sometimes called *data integrity*). If I is information about the origin of something, or about an identity, the members of X trust that the information is correct and unchanged (this aspect is sometimes called *origin integrity* or, more commonly, *authentication*). Also, I may be a resource rather than information. In that case, integrity means that the resource functions correctly (meeting its specifications). This aspect is called *assurance* and will be discussed in Part 6, "Assurance." As with confidentiality, the membership of X is often implicit.

Definition 4–6. Let X be a set of entities and let I be a resource. Then I has the property of *availability* with respect to X if all members of X can access I.

The exact definition of "access" in Definition 4–6 varies depending on the needs of the members of X, the nature of the resource, and the use to which the resource is put. If a book-selling server takes up to 1 hour to service a request to purchase a book, that may meet the client's requirements for "availability." If a server of medical information takes up to 1 hour to service a request for information regarding an allergy to an anesthetic, that will not meet an emergency room's requirements for "availability."

A security policy considers all relevant aspects of confidentiality, integrity, and availability. With respect to confidentiality, it identifies those states in which information leaks to those not authorized to receive it. This includes not only the leakage of rights but also the illicit transmission of information without leakage of rights, called *information flow*. Also, the policy must handle dynamic changes of authorization, so it includes a temporal element. For example, a contractor working for a company may be authorized to access proprietary information during the lifetime of a nondisclosure agreement, but when that nondisclosure agreement expires, the contractor can no longer access that information. This aspect of the security policy is often called a *confidentiality policy*.

With respect to integrity, a security policy identifies authorized ways in which information may be altered and entities authorized to alter it. Authorization may derive from a variety of relationships, and external influences may constrain it; for example, in many transactions, a principle called *separation of duties* forbids an entity from completing the transaction on its own. Those parts of the security policy that describe the conditions and manner in which data can be altered are called the *integrity policy*.

With respect to availability, a security policy describes what services must be provided. It may present parameters within which the services will be accessible—for example, that a browser may download Web pages but not Java applets. It may require a level of service—for example, that a server will provide authentication data within 1 minute of the request being made. This relates directly to issues of quality of service.

The statement of a security policy may formally state the desired properties of the system. If the system is to be provably secure, the formal statement will allow the designers and implementers to prove that those desired properties hold. If a formal proof is unnecessary or infeasible, analysts can test that the desired properties hold for some set of inputs. Later chapters will discuss both these topics in detail.

In practice, a less formal type of security policy defines the set of authorized states. Typically, the security policy assumes that the reader understands the context in which the policy is issued—in particular, the laws, organizational policies, and other environmental factors. The security policy then describes conduct, actions, and authorizations defining "authorized users" and "authorized use."

EXAMPLE: A university disallows cheating, which is defined to include copying another student's homework assignment (with or without permission). A computer science class requires the students to do their homework on the department's computer. One student notices that a second student has not read protected the file containing her homework and copies it. Has either student (or have both students) breached security?

The second student has not, despite her failure to protect her homework. The security policy requires no action to prevent files from being read. Although she may have been too trusting, the policy does not ban this; hence, the second student has not breached security.

The first student has breached security. The security policy disallows the copying of homework, and the student has done exactly that. Whether the security policy specifically states that "files containing homework shall not be copied" or simply says that "users are bound by the rules of the university" is irrelevant; in the latter case, one of those rules bans cheating. If the security policy is silent on such matters, the most reasonable interpretation is that the policy disallows actions that the university disallows, because the computer science department is part of the university.

The retort that the first user could copy the files, and therefore the action is allowed, confuses *mechanism* with *policy*. The distinction is sharp:

Definition 4–7. A *security mechanism* is an entity or procedure that enforces some part of the security policy.

EXAMPLE: In the preceding example, the policy is the statement that no student may copy another student's homework. One mechanism is the file access controls; if the second student had set permissions to prevent the first student from reading the file containing her homework, the first student could not have copied that file.

EXAMPLE: Another site's security policy states that information relating to a particular product is proprietary and is not to leave the control of the company. The company stores its backup tapes in a vault in the town's bank (this is common practice in case the computer installation is completely destroyed). The company must ensure that only authorized employees have access to the backup tapes even when the tapes are stored off-site; hence, the bank's controls on access to the vault, and the procedures used to transport the tapes to and from the bank, are considered security mechanisms. Note that these mechanisms are not technical controls built into the computer. Procedural, or operational, controls also can be security mechanisms.

Security policies are often implicit rather than explicit. This causes confusion, especially when the policy is defined in terms of the mechanisms. This definition may be ambiguous—for example, if some mechanisms prevent a specific action and others allow it. Such policies lead to confusion, and sites should avoid them.

EXAMPLE: The UNIX operating system, initially developed for a small research group, had mechanisms sufficient to prevent users from accidentally damaging one another's files; for example, the user *ken* could not delete the user *dmr*'s files (unless *dmr* had set the files and the containing directories appropriately). The implied security policy for this friendly environment was "do not delete or corrupt another's files, and any file not protected may be read."

When the UNIX operating system moved into academic institutions and commercial and government environments, the previous security policy became inadequate; for example, some files had to be protected from individual users (rather than from groups of users). Not surprisingly, the security mechanisms were inadequate for those environments.

The difference between a policy and an abstract description of that policy is crucial to the analysis that follows.

Definition 4–8. A *security model* is a model that represents a particular policy or set of policies.

A model abstracts details relevant for analysis. Analyses rarely discuss particular policies; they usually focus on *specific characteristics* of policies, because many policies exhibit these characteristics; and the more policies with those characteristics, the more useful the analysis. By the HRU result (see Theorem 3–2), no single nontrivial analysis can cover all policies, but restricting the class of security policies sufficiently allows meaningful analysis of that class of policies.

4.2 Types of Security Policies

Each site has its own requirements for the levels of confidentiality, integrity, and availability, and the site policy states these needs for that particular site.

Definition 4–9. A *military security policy* (also called a *governmental security policy*) is a security policy developed primarily to provide confidentiality.

The name comes from the military's need to keep information, such as the date that a troop ship will sail, secret. Although integrity and availability are important, organizations using this class of policies can overcome the loss of either—for example, by using orders not sent through a computer network. But the compromise of confidentiality would be catastrophic, because an opponent would be able to plan countermeasures (and the organization may not know of the compromise).

Confidentiality is one of the factors of privacy, an issue recognized in the laws of many government entities (such as the Privacy Act of the United States and similar legislation in Sweden). Aside from constraining what information a government entity can legally obtain from individuals, such acts place constraints on the disclosure and use of that information. Unauthorized disclosure can result in penalties that include jail or fines; also, such disclosure undermines the authority and respect that individuals have for the government and inhibits them from disclosing that type of information to the agencies so compromised.

executables be owned by the user *bin*. The vendor's patch had to be undone and fixed for the local configuration. This assumption also covers possible conflicts between different patches, as well as patches that conflict with one another (such as patches from different vendors of software that the system is using).

4. She is assuming that the patch is installed correctly. Some patches are simple to install, because they are simply executable files. Others are complex, requiring the system administrator to reconfigure network-oriented properties, add a user, modify the contents of a registry, give rights to some set of users, and then reboot the system. An error in any of these steps could prevent the patch from correcting the problems, as could an inconsistency between the environments in which the patch was developed and in which the patch is applied. Furthermore, the patch may claim to require specific privileges, when in reality the privileges are unnecessary and in fact dangerous.

These assumptions are fairly high-level, but invalidating any of them makes the patch a potential security problem.

Assumptions arise also at a much lower level. Consider formal verification (see Chapter 20), an oft-touted panacea for security problems. The important aspect is that formal verification provides a formal mathematical proof that a given program P is correct—that is, given any set of inputs i, j, k, the program P will produce the output x that its specification requires. This level of assurance is greater than most existing programs provide, and hence makes P a desirable program. Suppose a security-related program S has been formally verified for the operating system O. What assumptions would be made when it was installed?

1. The formal verification of S is correct—that is, the proof has no errors. Because formal verification relies on automated theorem provers as well as human analysis, the theorem provers must be programmed correctly.
2. The assumptions made in the formal verification of S are correct; specifically, the preconditions hold in the environment in which the program is to be executed. These preconditions are typically fed to the theorem provers as well as the program S. An implicit aspect of this assumption is that the version of O in the environment in which the program is to be executed is the same as the version of O used to verify S.
3. The program will be transformed into an executable whose actions correspond to those indicated by the source code; in other words, the compiler, linker, loader, and any libraries are correct. An experiment with one version of the UNIX operating system demonstrated how devastating a rigged compiler could be, and attackers have replaced libraries with others that performed additional functions, thereby increasing security risks.

4. The hardware will execute the program as intended. A program that relies on floating point calculations would yield incorrect results on some computer CPU chips, regardless of any formal verification of the program, owing to a flaw in these chips [202]. Similarly, a program that relies on inputs from hardware assumes that specific conditions cause those inputs.

The point is that *any* security policy, mechanism, or procedure is based on assumptions that, if incorrect, destroy the superstructure on which it is built. Analysts and designers (and users) must bear this in mind, because unless they understand what the security policy, mechanism, or procedure is based on, they jump from an unwarranted assumption to an erroneous conclusion.

4.4 Types of Access Control

A security policy may use two types of access controls, alone or in combination. In one, access control is left to the discretion of the owner. In the other, the operating system controls access, and the owner cannot override the controls.

The first type is based on user identity and is the most widely known:

Definition 4–13. If an individual user can set an access control mechanism to allow or deny access to an object, that mechanism is a *discretionary access control* (DAC), also called an *identity-based access control* (IBAC).

Discretionary access controls base access rights on the identity of the subject and the identity of the object involved. Identity is the key; the owner of the object constrains who can access it by allowing only particular subjects to have access. The owner states the constraint in terms of the identity of the subject, or the owner of the subject.

EXAMPLE: Suppose a child keeps a diary. The child controls access to the diary, because she can allow someone to read it (grant read access) or not allow someone to read it (deny read access). The child allows her mother to read it, but no one else. This is a discretionary access control because access to the diary is based on the identity of the subject (mom) requesting read access to the object (the diary).

The second type of access control is based on fiat, and identity is irrelevant:

Definition 4–14. When a system mechanism controls access to an object and an individual user cannot alter that access, the control is a *mandatory access control* (MAC), occasionally called a *rule-based access control*.

The operating system enforces mandatory access controls. Neither the subject nor the owner of the object can determine whether access is granted. Typically, the system mechanism will check information associated with both the subject and the object to determine whether the subject should access the object. Rules describe the conditions under which access is allowed.

EXAMPLE: The law allows a court to access driving records without the owners' permission. This is a mandatory control, because the owner of the record has no control over the court's accessing the information.

Definition 4–15. An *originator controlled access control* (ORCON or ORGCON) bases access on the creator of an object (or the information it contains).

The goal of this control is to allow the originator of the file (or of the information it contains) to control the dissemination of the information. The owner of the file has no control over who may access the file. Section 7.3 discusses this type of control in detail.

EXAMPLE: Bit Twiddlers, Inc., a company famous for its embedded systems, contracts with Microhackers Ltd., a company equally famous for its microcoding abilities. The contract requires Microhackers to develop a new microcode language for a particular processor designed to be used in high-performance embedded systems. Bit Twiddlers gives Microhackers a copy of its specifications for the processor. The terms of the contract require Microhackers to obtain permission before it gives any information about the processor to its subcontractors. This is an originator controlled access mechanism because, even though Microhackers owns the file containing the specifications, it may not allow anyone to access that information unless the creator, Bit Twiddlers, gives permission.

4.5 Policy Languages

A *policy language* is a language for representing a security policy. High-level policy languages express policy constraints on entities using abstractions. Low-level policy languages express constraints in terms of input or invocation options to programs existing on the systems.

4.5.1 High-Level Policy Languages

A policy is independent of the mechanisms. It describes constraints placed on entities and actions in a system. A high-level policy language is an unambiguous expression of policy. Such precision requires a mathematical or programmatic formulation of policy; common English is not precise enough.

Assume that a system is connected to the Internet. A user runs a World Wide Web browser. Web browsers download programs from remote sites and execute them locally. The local system's policy may constrain what these downloaded programs can do.

EXAMPLE: Java is a programming language designed for programs to be downloaded by Web browsers. Pandey and Hashii [794] developed a policy constraint language for Java programs. Their high-level policy language specifies access constraints on resources and on how those constraints are inherited.

Their language expresses entities as classes and methods. A *class* is a set of objects to which a particular access constraint may be applied; a *method* is the set of ways in which an operation can be invoked. *Instantiation* occurs when a subject s creates an instance of a class c, and is written $s \dashv c$. *Invocation* occurs when a subject s_1 executes an object s_2 (which becomes a subject, because it is active) and is written $s_1 \mapsto s_2$. A *condition* is a Boolean condition. Access constraints are of the form

$$\textbf{deny}(s\ op\ x)\ \text{when}\ b$$

where op is $\dashv$ or $\mapsto$, s is a subject, x is another subject or a class, and b is a Boolean expression. This constraint states that subject s cannot perform operation op on x when condition b is true. If s is omitted, the action is forbidden to all entities.

Inheritance causes access constraints to be conjoined. Specifically, let class c_1 define a method f, and have a subclass c_2. Then c_2 inherits f. Assume that the constraints are

$$\textbf{deny}(s \mapsto c_1.f)\ \textbf{when}\ b_1$$
$$\textbf{deny}(s \mapsto c_2.f)\ \textbf{when}\ b_2$$

A subclass inherits constraints on the parent class. Hence, *both* constraints b_1 and b_2 constrain c_2's invocation of f. The appropriate constraint is

$$\textbf{deny}(s \mapsto c_2.f)\ \textbf{when}\ b_1 \vee b_2$$

Suppose the policy states that the downloaded program is not allowed to access the password file on a UNIX system. The program accesses local files using the following class and methods.

```
class File {
public file(String name);
public String getfilename();
public char read();
...
```

Then the appropriate constraint would be

```
deny( |→ file.read) when (file.getfilename() ==
"/etc/passwd")
```

As another example, let the class *Socket* define the network interface, and let the method *Network.numconns* define the number of network connections currently active. The following access constraint bars any new connections when 100 connections are currently open.

```
deny( -| Socket) when (Network.numconns >= 100).
```

This language ignores implementation issues, and so is a high-level policy language. The *domain-type enforcement language* (DTEL) [54] grew from an observation of Boebert and Kain [126] that access could be based on types; they confine their work to the types "data" and "instructions." This observation served as the basis for a firewall [996] and for other secure system components. DTEL uses implementation-level constructs to express constraints in terms of language types, but not as arguments or input to specific system commands. Hence, it combines elements of low-level and high-level languages. Because it describes configurations in the abstract, it is a high-level policy language.

EXAMPLE: DTEL associates a type with each object and a domain with each subject. The constructs of the language constrain the actions that a member of a domain can perform on an object of a specific type. For example, a subject cannot execute a text file, but it can execute an object file.

Consider a policy that restricts all users from writing to system binaries. Only subjects in the administrative domain can alter system binaries. A user can enter this domain only after rigorous authentication checks. In the UNIX world, this suggests four distinct subject domains:

1. *d_user*, the domain for ordinary users
2. *d_admin*, the domain for administrative users (who can alter system binaries)
3. *d_login*, the domain for the authentication processes that comply with the domain-type enforcement
4. *d_daemon*, the domain for system daemons (including those that spawn login)

The *login* program (in the *d_login* domain) controls access between *d_user* and *d_admin*. The system begins in the *d_daemon* domain because the *init* process lies there (and *init* spawns the *login* process whenever anyone tries to log in).

The policy suggests five object types:

lines are processed in order, so everything on the system without a type assigned by the last four lines is of type *t_generic* (because of the first line).

If a user process tries to alter a system binary, the enforcement mechanisms will check to determine if something in the domain *d_user* is authorized to write to an object of type *t_sysbin*. Because the domain description does not allow this, the request is refused.

Now augment the policy above to prevent users from modifying system logs. Define a new type *t_log* for the log files. Only subjects in the *d_admin* domain, and in a new domain *d_log*, can alter the log files. The set of domains would be extended as follows.

```
type t_readable, t_writable, t_sysbin, t_dte, t_generic, t_log;
domain d_daemon = (/sbin/init),
                  (crwd->t_writable),
                  (rxd->t_readable),
                  (rd->t_generic, t_dte, t_sysbin),
                  (auto->d_login, d_log);
domain d_log =    (/usr/sbin/syslogd),
                  (crwd->t_log),
                  (rwd->t_writable),
                  (rd->t_generic, t_readable);
assign -r t_log /usr/var/log;
assign t_writable /usr/var/log/wtmp, /usr/var/log/utmp;
```

If a process in the domain *d_daemon* invokes the *syslogd* process, the *syslogd* process enters the *d_log* domain. It can now manipulate system logs and can read and write writable logs but cannot access system executables. If a user tries to manipulate a log object, the request is denied. The *d_user* domain gives its subjects no rights over *t_log* objects.

4.5.2 Low-Level Policy Languages

A low-level policy language is simply a set of inputs or arguments to commands that set, or check, constraints on a system.

EXAMPLE: The UNIX-based windowing system X11 provides a language for controlling access to the console (on which X11 displays its images). The language consists of a command, *xhost*, and a syntax for instructing the command to allow access based on host name (IP address). For example,

```
xhost +groucho -chico
```

sets the system so that connections from the host *groucho* are allowed but connections from *chico* are not.

EXAMPLE: File system scanning programs check conformance of a file system with a stated policy. The policy consists of a database with desired settings. Each scanning program uses its own little language to describe the settings desired.

One such program, *tripwire* [569], assumes a policy of constancy. It records an initial state (the state of the system when the program is first run). On subsequent runs, it reports files whose settings have changed.

The policy language consists of two files. The first, the *tw.config* file, contains a description of the attributes to be checked. The second, the database, contains the values of the attributes from a previous execution. The database is kept in a readable format but is very difficult to edit (for example, times of modification are kept using base 64 digits). Hence, to enforce conformance with a specific policy, an auditor must ensure that the system is in the desired state initially and set up the *tw.config* file to ignore the attributes not relevant to the policy.

The attributes that *tripwire* can check are protection, file type, number of links, file size, file owner, file group, and times of creation, last access, and last modification. *Tripwire* also allows the cryptographic checksumming of the contents of the file. An example *tripwire* configuration file looks like

```
/usr/mab/tripwire-1.1 +gimnpsu012345678-a
```

This line states that all attributes are to be recorded, including all nine cryptographic checksums, but that the time of last access (the "a") is to be ignored (the "-"). This applies to the directory and to all files and subdirectories contained in it. After *tripwire* is executed, the database entry for that README file might be

```
/usr/mab/tripwire-1.1/README 0 ..../. 100600 45763 1 917
10 33242 .gtPvf .gtPvY .gtPvY 0 .ZD4cc0Wr8i21ZKaI..LUOr3
.0fwo5:hf4e4.8TAqd0V4ubv ?...... ...9b3
1M4GX01xbGIX0oVuGo1h15z3 ?:Y9jfa04rdzM1q:eqt1APgHk
?.Eb9yo.2zkEh1XKovX1:d0wF0kfAvC
?1M4GX01xbGIX2947jdyrior38h15z3 0
```

Clearly, administrators are not expected to edit the database to set attributes properly. Hence, if the administrator wishes to check conformance with a particular *policy* (as opposed to looking for changes), the administrator must ensure that the system files conform to that policy and that the configuration file reflects the attributes relevant to the policy.

EXAMPLE: The RIACS file system checker [105] was designed with different goals. It emphasized the ability to set policy and then check for conformance. It uses a database file and records fixed attributes (with one exception—the cryptographic checksum). The property relevant to this discussion is that the database entries are easy to understand and edit:

```
/etc/pac 0755 1 root root 16384 12 22341 Jan 12, 1987 at 12:47:54
```

The attribute values follow the file name and are permissions, number of links, owner and group, size (in bytes), checksum, and date and time of last modification. After generating such a file, the analyst can change the values as appropriate (and replace those that are irrelevant with a wild card "*"). On the next run, the file system state is compared with these values.

4.6 Example: Academic Computer Security Policy

Security policies can have few details, or many. The explicitness of a security policy depends on the environment in which it exists. A research lab or office environment may have an unwritten policy. A bank needs a very explicit policy. In practice, policies begin as generic statements of constraints on the members of the organization. These statements are derived from an analysis of threats, as described in Chapter 1, "An Overview of Computer Security." As questions (or incidents) arise, the policy is refined to cover specifics. As an example, we present an academic security policy. The full policy is presented in Chapter 35, "Example Academic Security Policy."

4.6.1 General University Policy

This policy is an "Acceptable Use Policy" (AUP) for the Davis campus of the University of California. Because computing services vary from campus unit to campus unit, the policy does not dictate how the specific resources can be used. Instead, it presents generic constraints that the individual units can tighten.

The policy first presents the goals of campus computing: to provide access to resources and to allow the users to communicate with others throughout the world. It then states the responsibilities associated with the privilege of using campus computers. All users must "respect the rights of other users, respect the integrity of the systems and related physical resources, and observe all relevant laws, regulations, and contractual obligations."[1]

The policy states the intent underlying the rules, and notes that the system managers and users must abide by the law (for example, "Since electronic information is volatile and easily reproduced, users must exercise care in acknowledging and respecting the work of others through strict adherence to software licensing agreements and copyright laws").[2]

The enforcement mechanisms in this policy are procedural. For minor violations, either the unit itself resolves the problem (for example, by asking the offender not to do it again) or formal warnings are given. For more serious infractions, the administration may take stronger action such as denying access to campus computer

[1] See Section 35.2.1.2.
[2] See Section 35.2.1.2.

systems. In very serious cases, the university may invoke disciplinary action. The Office of Student Judicial Affairs hears such cases and determines appropriate consequences.

The policy then enumerates specific examples of actions that are considered to be irresponsible use. Among these are illicitly monitoring others, spamming, and locating and exploiting security vulnerabilities. These are examples; they are not exhaustive. The policy concludes with references to other documents of interest.

This is a typical AUP. It is written informally and is aimed at the user community that is to abide by it. The electronic mail policy presents an interesting contrast to the AUP, probably because the AUP is for UC Davis only, and the electronic mail policy applies to all nine University of California campuses.

4.6.2 Electronic Mail Policy

The university has several auxiliary policies, which are subordinate to the general university policy. The electronic mail policy describes the constraints imposed on access to, and use of, electronic mail. It conforms to the general university policy but details additional constraints on both users and system administrators.

The electronic mail policy consists of three parts. The first is a short summary intended for the general user community, much as the AUP for UC Davis is intended for the general user community. The second part is the full policy for all university campuses and is written as precisely as possible. The last document describes how the Davis campus implements the general university electronic mail policy.

4.6.2.1 The Electronic Mail Policy Summary

The summary first warns users that their electronic mail is not private. It may be read accidentally, in the course of normal system maintenance, or in other ways stated in the full policy. It also warns users that electronic mail can be forged or altered as well as forwarded (and that forwarded messages may be altered). This section is interesting because policies rarely alert users to the threats they face; policies usually focus on the remedial techniques.

The next two sections are lists of what users should, and should not, do. They may be summarized as "think before you send; be courteous and respectful of others; and don't interfere with others' use of electronic mail." They emphasize that supervisors have the right to examine employees' electronic mail that relates to the job. Surprisingly, the university does not ban personal use of electronic mail, probably in the recognition that enforcement would demoralize people and that the overhead of carrying personal mail is minimal in a university environment. The policy does require that users not use personal mail to such an extent that it interferes with their work or causes the university to incur extra expense.

Finally, the policy concludes with a statement about its application. In a private company, this would be unnecessary, but the University of California is a quasi-governmental institution and as such is bound to respect parts of the United States

Constitution and the California Constitution that private companies are not bound to respect. Also, as an educational institution, the university takes the issues surrounding freedom of expression and inquiry very seriously. Would a visitor to campus be bound by these policies? The final section says yes. Would an employee of Lawrence Livermore National Laboratories, run for the Department of Energy by the University of California, also be bound by these policies? Here, the summary suggests that they would be, but whether the employees of the lab are Department of Energy employees or University of California employees could affect this. So we turn to the full policy.

4.6.2.2 The Full Policy

The full policy also begins with a description of the context of the policy, as well as its purpose and scope. The scope here is far more explicit than that in the summary. For example, the full policy does not apply to e-mail services of the Department of Energy laboratories run by the university, such as Lawrence Livermore National Laboratories. Moreover, this policy does not apply to printed copies of e-mail, because other university policies apply to such copies.

The general provisions follow. They state that e-mail services and infrastructure are university property, and that all who use them are expected to abide by the law and by university policies. Failure to do so may result in access to e-mail being revoked. The policy reiterates that the university will apply principles of academic freedom and freedom of speech in its handling of e-mail, and so will seek access to e-mail without the holder's permission only under extreme circumstances, which are enumerated, and only with the approval of a campus vice chancellor or a university vice president (essentially, the second ranking officer of a campus or of the university system). If this is infeasible, the e-mail may be read only as is needed to resolve the emergency, and then authorization must be secured after the fact.

The next section discusses legitimate and illegitimate use of the university's e-mail. The policy allows anonymity to senders provided that it does not violate laws or other policies. It disallows using mail to interfere with others, such as by sending spam or letter bombs. It also expressly permits the use of university facilities for sending personal e-mail, provided that doing so does not interfere with university business; and it cautions that such personal e-mail may be treated as a "University record" subject to disclosure.

The discussion of security and confidentiality emphasizes that, although the university will not go out of its way to read e-mail, it can do so for legitimate business purposes and to keep e-mail service robust and reliable. The section on archiving and retention says that people may be able to recover e-mail from end systems where it may be archived as part of a regular backup.

The last three sections discuss the consequences of violations and direct the chancellor of each campus to develop procedures to implement the policy.

An interesting sidelight occurs in Appendix A, "Definitions." The definition of "E-mail" includes any computer records viewed with e-mail systems or services, and

Now we define a confidentiality policy.

Definition 4–18. A *confidentiality policy* for the program $p{:}I_1 \times ... \times I_n \rightarrow R$ is a function $c{:}I_1 \times ... \times I_n \rightarrow A$, where $A \subseteq I_1 \times ... \times I_n$.

In this definition, A corresponds to the set of inputs that may be revealed. The complement of A with respect to $I_1 \times ... \times I_n$ corresponds to the confidential inputs. In some sense, the function c filters out inputs that are to be kept confidential.

The next definition captures how well a security mechanism conforms to a stated confidentiality policy.

Definition 4–19. Let $c{:}I_1 \times ... \times I_n \rightarrow A$ be a confidentiality policy for a program p. Let $m{:}I_1 \times ... \times I_n \rightarrow R \cup E$ be a security mechanism for the same program p. Then the mechanism m is *secure* if and only if there is a function $m'{:}A \rightarrow R \cup E$ such that, for all $i_k \in I_k$, $1 \le k \le n$, $m(i_1, ..., i_n) = m'(c(i_1, ..., i_n))$.

In other words, given any set of inputs, the protection mechanism m returns values consistent with the stated confidentiality policy c. Here, the term "secure" is a synonym for "confidential." We can derive analogous results for integrity policies.

EXAMPLE: If $c(i_1, ..., i_n)$ is a constant, the policy's intent is to deny the observer any information, because the output does not vary with the inputs. But if $c(i_1, ..., i_n) = (i_1, ..., i_n)$, and $m' = m$, then the policy's intent is to allow the observer full access to the information. As an intermediate policy, if $c(i_1, ..., i_n) = i_1$, then the policy's intent is to allow the observer information about the first input but no information about other inputs.

The distinguished policy $allow{:}I_1 \times ... \times I_n \rightarrow A$ generates a selective permutation of its inputs. By "selective," we mean that it may omit inputs. Hence, the function $c(i_1, ..., i_n) = i_1$ is an example of *allow*, because its output is a permutation of some of its inputs. More generally, for $k \le n$,

$$allow(i_1, ..., i_n) = (i_1', ..., i_k')$$

where $i_1', ..., i_k'$ is a permutation of any k of $i_1, ..., i_n$.

EXAMPLE: Revisit the program that checks user name and password association. As a function, $auth$: $U \times P \times D \rightarrow \{ T, F \}$, where U is the set of potential user names, D is the databases, and P is the set of potential passwords. T and F represent true and false, respectively. Then for $u \in U, p \in P$, and $d \in D$, $auth(u, p, d) = T$ if and only if the pair $(u, p) \in d$. Under the policy $allow(i_1, i_2, i_3) = (i_1, i_2)$, there is no function $auth'$ such that

$$auth'(allow(u, p, d)) = auth'(u, p) = auth(u, p, d)$$

for all d. So *auth* is not secure as an enforcement mechanism.

EXAMPLE: Consider a program q with k non-negative integer inputs; it computes a single non-negative integer. A Minsky machine [717] can simulate this program by starting with the input $i_j \in I_j$ in register j (for $1 \leq j \leq k$). The output may disclose information about one or more inputs. For example, if the program is to return the third input as its output, it is disclosing information. Fenton [345] examines these functions to determine if the output contains confidential information.

The observability postulate does not hold for the program q above, because q ignores runtime. The computation may take more time for certain inputs, revealing information about them. This is an example of a *covert channel* (see Section 17.3). It also illustrates the need for precise modeling. The policy does not consider runtime as an output when, in reality, it is an output.

As an extreme example, consider the following program.

```
if x = null then halt;
```

Fenton does not define what happens if x is not **null**. If an error message is printed, the resulting mechanism may not be secure. To see this, consider the program

```
y := 0;
if x = 0 then begin
    y := 1;
    halt;
end;
halt;
```

Here, the value of y is the error message. It indicates whether or not the value of x is 0 when the program terminates. If the security policy says that information about x is not to be revealed, then this mechanism is not secure.

A secure mechanism ensures that the policy is obeyed. However, it may also disallow actions that do not violate the policy. In that sense, a secure mechanism may be overly restrictive. The notion of *precision* measures the degree of overrestrictiveness.

Definition 4–20. Let m_1 and m_2 be two distinct protection mechanisms for the program p under the policy c. Then m_1 is *as precise as* m_2 ($m_1 \approx m_2$) provided that, for all inputs $(i_1, \ldots, i_n)$, if $m_2(i_1, \ldots, i_n) = p(i_1, \ldots, i_n)$, then $m_1(i_1, \ldots, i_n) = p(i_1, \ldots, i_n)$. We say that m_1 is *more precise than* m_2 ($m_1 \sim m_2$) if there is an input $(i_1', \ldots, i_n')$ such that $m_1(i_1', \ldots, i_n') = p(i_1', \ldots, i_n')$ and $m_2(i_1', \ldots, i_n') \neq p(i_1', \ldots, i_n')$.

An obvious question is whether or not two protection mechanisms can be combined to form a new mechanism that is as precise as the two original ones. To answer this, we need to define "combines," which we formalize by the notion of "union."

Definition 4–21. Let m_1 and m_2 be protection mechanisms for the program p. Then their union $m_3 = m_1 \cup m_2$ is defined as

$$m_3(i_1, \ldots, i_n) = \begin{cases} = p(i_1, \ldots, i_n) \text{ when } m_1(i_1, \ldots, i_n) = p(i_1, \ldots, i_n) \text{ or} \\ \quad m_2(i_1, \ldots, i_n) = p(i_1, \ldots, i_n) \\ = m_1(i_1, \ldots, i_n) \text{ otherwise.} \end{cases}$$

This definition says that for inputs on which either m_1 and m_2 return the same value as p, their union does also. Otherwise, that mechanism returns the same value as m_1. From this definition and the definitions of *secure* and *precise*, we have:

Theorem 4–1. Let m_1 and m_2 be secure protection mechanisms for a program p and policy c. Then $m_1 \cup m_2$ is also a secure protection mechanism for p and c. Furthermore, $m_1 \cup m_2 \approx m_1$ and $m_1 \cup m_2 \approx m_2$.

Generalizing, we have:

Theorem 4–2. For any program p and security policy c, there exists a precise, secure mechanism m^* such that, for all secure mechanisms m associated with p and c, $m^* \approx m$.

Proof Immediate by induction on the number of secure mechanisms associated with p and c.

This "maximally precise" mechanism m^* is the mechanism that ensures security while minimizing the number of denials of legitimate actions. If there is an effective procedure for determining this mechanism, we can develop mechanisms that are both secure and precise. Unfortunately:

Theorem 4–3. There is no effective procedure that determines a maximally precise, secure mechanism for any policy and program.

Proof Let the policy c be the constant function—that is, no information about any of the inputs is allowed in the output. Let p be a program that computes the value of some total function $T(x)$ and assigns it to the variable z. We may without loss of generality take $T(0) = 0$.

Let q be a program of the following form:

```
p;
if z = 0 then y := 1 else y := 2;
halt;
```

Now consider the value of the protection mechanism m at 0. Because c is constant, m must also be constant. Using the program above, either $m(0) = 1$ (if p, and hence q, completes) or it is undefined (if p halts before the "if" statement).

If, for all inputs x, $T(x) = 0$, then $m(x) = 1$ (because m is secure). If there is an input x' for which $T(x') \neq 0$, then $m(x') = 2$ (again, because m is secure) or is undefined (if p halts before the assignment). In either case, m is not a constant; hence, no such p can exist. Thus, $m(0) = 1$ if and only if $T(x) = 0$ for all x.

If we can effectively determine m, we can effectively determine whether $T(x) = 0$ for all x. This contradicts the security policy c, so no such effective procedure can exist.

There is no general procedure for devising a mechanism that conforms exactly to a specific security policy and yet allows all actions that the policy allows. It may be possible to do so in specific cases, especially when a mechanism defines a policy, but there is no general way to devise a precise and secure mechanism.

4.8 Summary

Security policies define "security" for a system or site. They may be implied policies defined by the common consensus of the community, or they may be informal policies whose interpretations are defined by the community. Both of these types of policies are usually ambiguous and do not precisely define "security." A policy may be formal, in which case ambiguities arise either from the use of natural languages such as English or from the failure to cover specific areas.

Formal mathematical models of policies enable analysts to deduce a rigorous definition of "security" but do little to improve the average user's understanding of what "security" means for a site. The average user is not mathematically sophisticated enough to read and interpret the mathematics.

Trust underlies all policies and enforcement mechanisms. Policies themselves make assumptions about the way systems, software, hardware, and people behave. At a lower level, security mechanisms and procedures also make such assumptions. Even when rigorous methodologies (such as formal mathematical models or formal verification) are applied, the methodologies themselves simply push the assumptions, and therefore the trust, to a lower level. Understanding the assumptions and the trust involved in any policies and mechanisms deepens one's understanding of the security of a system.

This brief overview of policy, and of policy expression, lays the foundation for understanding the more detailed policy models used in practice.

4.9 Research Issues

The critical issue in security policy research is the expression of policy in an easily understood yet precise form. The development of policy languages focuses on supplying mathematical rigor that is intelligible to humans. A good policy language allows not only the expression of policy but also the analysis of a system to determine if it conforms to that policy. The latter may require that the policy language be compiled into an enforcement program (to enforce the stated policy, as DTEL does) or into a verification program (to verify that the stated policy is enforced, as *tripwire* does). Balancing enforcement with requirements is also an important area of research, particularly in real-time environments.

The underlying role of trust is another crucial issue in policy research. Development of methodologies for exposing underlying assumptions and for analyzing the effects of trust and the results of belief is an interesting area of formal mathematics as well as a guide to understanding the safety and security of systems. Design and implementation of tools to aid in this work are difficult problems on which research will continue for a long time to come.

4.10 Further Reading

Much of security analysis involves definition and refinement of security policies. Wood [1059] has published a book of templates for specific parts of policies. That book justifies each part and allows readers to develop policies by selecting the appropriate parts from a large set of possibilities. Essays by Bailey [55] and Abrams and Bailey [4] discuss management of security issues and explain why different members of an organization interpret the same policy differently. Sterne's wonderful paper [970] discusses the nature of policy in general.

Jajodia and his colleagues [520] present a "little language" for expressing authorization policies. They show that their language can express many aspects of existing policies and argue that it allows elements of these policies to be combined into authorization schemes.

Fraser and Badger [371] have used DTEL to enforce many policies. Cholvy and Cuppens [194] describe a method of checking policies for consistency and determining how they apply to given situations.

Son, Chaney, and Thomlinson [951] discuss enforcement of partial security policies in real-time databases to balance real-time requirements with security. Their idea of "partial security policies" has applications in other environments. Zurko and Simon [1074] present an alternative focus for policies.

4.11 Exercises

1. In Figure 4–1, suppose that edge t_3 went from s_1 to s_4. Would the resulting system be secure?

Chapter 5
Confidentiality Policies

SHEPHERD: Sir, there lies such secrets in this fardel
and box which none must know but the king;
and which he shall know within this hour, if I
may come to the speech of him.
—*The Winter's Tale*, IV, iv, 785–788.

Confidentiality policies emphasize the protection of confidentiality. The importance of these policies lies in part in what they provide, and in part in their role in the development of the concept of security. This chapter explores one such policy—the Bell-LaPadula Model—and the controversy it engendered.

5.1 Goals of Confidentiality Policies

A confidentiality policy, also called an *information flow policy*, prevents the unauthorized disclosure of information. Unauthorized alteration of information is secondary. For example, the navy must keep confidential the date on which a troop ship will sail. If the date is changed, the redundancy in the systems and paperwork should catch that change. But if the enemy knows the date of sailing, the ship could be sunk. Because of extensive redundancy in military communications channels, availability is also less of a problem.

The term "governmental" covers several requirements that protect citizens' privacy. In the United States, the Privacy Act requires that certain personal data be kept confidential. Income tax returns are legally confidential and are available only to the Internal Revenue Service or to legal authorities with a court order. The principle of "executive privilege" and the system of nonmilitary classifications suggest that the people working in the government need to limit the distribution of certain documents and information. Governmental models represent the policies that satisfy these requirements.

5.2 The Bell-LaPadula Model

The Bell-LaPadula Model [67, 68] corresponds to military-style classifications. It has influenced the development of many other models and indeed much of the development of computer security technologies.[1]

5.2.1 Informal Description

The simplest type of confidentiality classification is a set of *security clearances* arranged in a linear (total) ordering (see Figure 5–1). These clearances represent sensitivity levels. The higher the security clearance, the more sensitive the information (and the greater the need to keep it confidential). A subject has a *security clearance*. In the figure, Claire's security clearance is C (for CONFIDENTIAL), and Thomas' is TS (for TOP SECRET). An object has a *security classification*; the security classification of the electronic mail files is S (for SECRET), and that of the telephone list files is UC (for UNCLASSIFIED). (When we refer to both subject clearances and object classifications, we use the term "classification.") The goal of the Bell-LaPadula security model is to prevent read access to objects at a security classification higher than the subject's clearance.

The Bell-LaPadula security model combines mandatory and discretionary access controls. In what follows, "*S* has discretionary read (write) access to *O*" means that the access control matrix entry for *S* and *O* corresponding to the discretionary access control component contains a read (write) right. In other words, were the mandatory controls not present, *S* would be able to read (write) *O*.

TOP SECRET (TS)	Tamara, Thomas	Personnel Files
|	|	|
SECRET (S)	Sally, Samuel	Electronic Mail Files
|	|	|
CONFIDENTIAL (C)	Claire, Clarence	Activity Log Files
|	|	|
UNCLASSIFIED (UC)	Ulaley, Ursula	Telephone List Files

Figure 5–1 At the left is the basic confidentiality classification system. The four security levels are arranged with the most sensitive at the top and the least sensitive at the bottom. In the middle are individuals grouped by their security clearances, and at the right is a set of documents grouped by their security levels.

[1] The terminology in this section follows that of the unified exposition of the Bell-LaPadula Model [68].

Let $L(S) = l_s$ be the security clearance of subject S, and let $L(O) = l_o$ be the security classification of object O. For all security classifications l_i, $i = 0, ..., k - 1$, $l_i < l_{i+1}$.

- ***Simple Security Condition*, *Preliminary Version:*** S can read O if and only if $l_o \le l_s$ and S has discretionary read access to O.

In Figure 5–1, for example, Claire and Clarence cannot read personnel files, but Tamara and Sally can read the activity log files (and, in fact, Tamara can read any of the files, given her clearance), assuming that the discretionary access controls allow it.

Should Tamara decide to copy the contents of the personnel files into the activity log files and set the discretionary access permissions appropriately, Claire could then read the personnel files. Thus, for all practical purposes, Claire could read the files at a higher level of security. A second property prevents this:

- ****-Property (Star Property)*, *Preliminary Version:*** S can write O if and only if $l_s \le l_o$ and S has discretionary write access to O.

Because the activity log files are classified C and Tamara has a clearance of TS, she cannot write to the activity log files.

Define a *secure system* as one in which both the simple security condition, preliminary version, and the *-property, preliminary version, hold. A straightforward induction establishes the following theorem.

> **Theorem 5–1.** ***Basic Security Theorem*, *Preliminary Version:*** Let Σ be a system with a secure initial state σ_0, and let T be a set of state transformations. If every element of T preserves the simple security condition, preliminary version, and the *-property, preliminary version, then every state σ_i, $i \ge 0$, is secure.

Expand the model by adding a set of *categories* to each security classification. Each category describes a kind of information. Objects placed in multiple categories have the kinds of information in all of those categories. These categories arise from the "need to know" principle, which states that no subject should be able to read objects unless reading them is necessary for that subject to perform its functions. The sets of categories to which a person may have access is simply the power set of the set of categories. For example, if the categories are NUC, EUR, and US, someone can have access to any of the following sets of categories: $\varnothing$ (none), { NUC }, { EUR }, { US }, { NUC, EUR }, {NUC, US }, { EUR, US }, and { NUC, EUR, US }. These sets of categories form a lattice under the operation $\subseteq$ (subset of); see Figure 5–2. (Chapter 30, "Lattices," discusses the mathematical nature of lattices.)

Each security level and category form a *security level*.[2] As before, we say that subjects *have clearance at* (or *are cleared into*, or *are in*) a security level and that

[2] There is less than full agreement on this terminology. Some call security levels "compartments." However, others use this term as a synonym for "categories." We follow the terminology of the unified exposition [68].

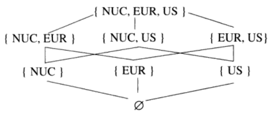


Figure 5–2 Lattice generated by the categories NUC, EUR, and US. The lines represent the ordering relation induced by $\subseteq$.

objects *are at the level of* (or *are in*) a security level. For example, William may be cleared into the level (SECRET, { EUR }) and George into the level (TOP SECRET, { NUC, US }). A document may be classified as (CONFIDENTIAL, {EUR }).

Security levels change access. Because categories are based on a "need to know," someone with access to the category set { NUC, US } presumably has no need to access items in the category EUR. Hence, read access should be denied, even if the security clearance of the subject is higher than the security classification of the object. But if the desired object is in any of the security levels ∅, { NUC }, { US }, or { NUC, US } and the subject's security clearance is no less than the document's security classification, access should be granted because the subject is cleared into the same category set as the object.

This suggests a new relation for capturing the combination of security classification and category set. Define the relation *dom* (dominates) as follows.

> **Definition 5–1.** The security level (L, C) *dominates* the security level (L', C') if and only if $L' \le L$ and $C' \subseteq C$.

We write (L, C) $\neg dom$ (L', C') when (L, C) *dom* (L', C') is false. This relation also induces a lattice on the set of security levels [267].

EXAMPLE: George is cleared into security level (SECRET, { NUC, EUR}), DocA is classified as (CONFIDENTIAL, { NUC }), DocB is classified as (SECRET, { EUR, US}), and DocC is classified as (SECRET, { EUR }). Then:

George *dom* DocA as CONFIDENTIAL ≤ SECRET and { NUC } $\subseteq$ { NUC, EUR }
George $\neg dom$ DocB as { EUR, US } $\not\subseteq$ { NUC, EUR }
George *dom* DocC as SECRET ≤ SECRET and { EUR } $\subseteq$ { NUC, EUR }

Let $C(S)$ be the category set of subject S, and let $C(O)$ be the category set of object O. The simple security condition, preliminary version, is modified in the obvious way:

- ***Simple Security Condition:*** *S* can read *O* if and only if *S dom O* and *S* has discretionary read access to *O*.

In the example above, George can read DocA and DocC but not DocB (again, assuming that the discretionary access controls allow such access).

Suppose Paul is cleared into security level (SECRET, { EUR, US, NUC }) and has discretionary read access to DocB. Paul can read DocB; were he to copy its contents to DocA and set its access permissions accordingly, George could then read DocB. The modified *-property prevents this:

- ***-Property:*** *S* can write to *O* if and only if *O dom S* and *S* has discretionary write access to *O*.

Because DocA *dom* Paul is false (because $C(\text{Paul}) \not\subseteq C(\text{DocA})$), Paul cannot write to DocA.

The simple security condition is often described as "no reads up" and the *-property as "no writes down."

Redefine a *secure system* as one in which both the simple security property and the *-property hold. The analogue to the Basic Security Theorem, preliminary version, can also be established by induction.

Theorem 5–2. ***Basic Security Theorem:*** Let Σ be a system with a secure initial state σ_0, and let T be a set of state transformations. If every element of T preserves the simple security condition and the *-property, then every σ_i, $i \geq 0$, is secure.

At times, a subject must communicate with another subject at a lower level. This requires the higher-level subject to write into a lower-level object that the lower-level subject can read.

EXAMPLE: A colonel with (SECRET, { NUC, EUR }) clearance needs to send a message to a major with (SECRET, { EUR }) clearance. The colonel must write a document that has at most the (SECRET, { EUR }) classification. But this violates the *-property, because (SECRET, { NUC, EUR }) *dom* (SECRET, { EUR }).

The model provides a mechanism for allowing this type of communication. A subject has a *maximum security level* and a *current security level*. The maximum security level must dominate the current security level. A subject may (effectively) decrease its security level from the maximum in order to communicate with entities at lower security levels.

EXAMPLE: The colonel's maximum security level is (SECRET, { NUC, EUR }). She changes her current security level to (SECRET, { EUR }). This is valid, because the maximum security level dominates the current security level. She can then create the document at the major's clearance level and send it to him.

```
stat(".", &stat_buffer)
```

returns a different inode number for each process, because it returns the inode number of the current working directory—the hidden directory. The system call

```
dg_mstat(".", &stat_buffer)
```

translates the notion of "current working directory" to the multilevel directory when the current working directory is a hidden directory.

Mounting unlabeled file systems requires the files to be labeled. Symbolic links aggravate this problem. Does the MAC label the target of the link control, or does the MAC label the link itself? DG/UX uses a notion of inherited labels (called *implicit labels*) to solve this problem. The following rules control the way objects are labeled.

1. Roots of file systems have explicit MAC labels. If a file system without labels is mounted on a labeled file system, the root directory of the mounted file system receives an explicit label equal to that of the mount point. However, the label of the mount point, and of the underlying tree, is no longer visible, and so its label is unchanged (and will become visible again when the file system is unmounted).
2. An object with an implicit MAC label inherits the label of its parent.
3. When a hard link to an object is created, that object must have an explicit label; if it does not, the object's implicit label is converted to an explicit label. A corollary is that moving a file to a different directory makes its label explicit.
4. If the label of a directory changes, any immediate children with implicit labels have those labels converted to explicit labels before the parent directory's label is changed.
5. When the system resolves a symbolic link, the label of the object is the label of the target of the symbolic link. However, to resolve the link, the process needs access to the symbolic link itself.

Rules 1 and 2 ensure that every file system object has a MAC label, either implicit or explicit. But when a file object has an implicit label, and two hard links from different directories, it may have two labels. Let */x/y/z* and */x/a/b* be hard links to the same object. Suppose *y* has an explicit label IMPL_HI and *a* an explicit label *IMPL_B*. Then the file object can be accessed by a process at IMPL_HI as */x/y/z* and by a process at *IMPL_B* as */x/a/b*. Which label is correct? Two cases arise.

Suppose the hard link is created while the file system is on a DG/UX B2 system. Then the DG/UX system converts the target's implicit label to an explicit one (rule 3). Thus, regardless of the path used to refer to the object, the label of the object will be the same.

Suppose the hard link exists when the file system is mounted on the DG/UX B2 system. In this case, the target had no file label when it was created, and one must be added. If no objects on the paths to the target have explicit labels, the target will have the same (implicit) label regardless of the path being used. But if any object on any path to the target of the link acquires an explicit label, the target's label may depend on which path is taken. To avoid this, the implicit labels of a directory's children must be preserved when the directory's label is made explicit. Rule 4 does this.

Because symbolic links interpolate path names of files, rather than store inode numbers, computing the label of symbolic links is straightforward. If */x/y/z* is a symbolic link to */a/b/c*, then the MAC label of *c* is computed in the usual way. However, the symbolic link itself is a file, and so the process must also have access to the link file *z*.

5.2.2.2 Using MAC Labels

The DG/UX B2 system uses the Bell-LaPadula notion of dominance, with one change. The system obeys the simple security condition (reading down is permitted), but the implementation of the *-property requires that the process MAC label and the object MAC label be equal, so writing up is not permitted, but writing is permitted in the same compartment.

Because of this restriction on writing, the DG/UX system provides processes and objects with a range of labels called a *MAC tuple*. A *range* is a set of labels expressed by a *lower bound* and an *upper bound*. A MAC tuple consists of up to three ranges (one for each of the regions in Figure 5–3).

EXAMPLE: A system has two security levels, TS and S, the former dominating the latter. The categories are COMP, NUC, and ASIA. Examples of ranges are

[(S, { COMP }), (TS, { COMP })]
[(S, ∅), (TS, { COMP, NUC, ASIA })]
[(S, { ASIA }), (TS, { ASIA, NUC })]

The label (TS, {COMP}) is in the first two ranges. The label (S, {NUC, ASIA}) is in the last two ranges. However,

[(S, {ASIA}), (TS, { COMP, NUC})]

is not a valid range because *not* (TS, { COMP, NUC }) *dom* (S, { ASIA }).

An object can have a MAC tuple as well as the required MAC label. If both are present, the tuple overrides the label. A process has read access when its MAC label grants read access to the upper bound of the range. A process has write access when its MAC label grants write access to any label in the MAC tuple range.

EXAMPLE: Suppose an object's MAC tuple is the single range

[(S, { ASIA }), (TS, { ASIA, COMP})]

A subject with MAC label (S, { ASIA }) cannot read the object, because

(TS, { ASIA, COMP}) *dom* (S, { ASIA })

It can write to the object, because (S, { ASIA }) dominates the lower bound and is dominated by the upper bound. A subject with MAC label (TS, { ASIA, COMP, NUC }) can read the object but cannot write the object. A subject with MAC label (TS, { ASIA, COMP }) can both read and write the object. A subject with MAC label (TS, {EUR}) can neither read nor write the object, because its label is incomparable to that of the object, and the *dom* relation does not hold.

A process has both a MAC label and a MAC tuple. The label always lies within the range for the region in which the process is executing. Initially, the subject's accesses are restricted by its MAC label. However, the process may extend its read and write capabilities to within the bounds of the MAC tuple.

5.2.3 Formal Model

Let S be the set of subjects of a system and let O be the set of objects. Let P be the set of rights $\underline{r}$ for read, $\underline{a}$ for write, $\underline{w}$ for read/write, and $\underline{e}$ for empty.[5] Let M be a set of possible access control matrices for the system. Let C be the set of classifications (or clearances), let K be the set of categories, and let $L = C \times K$ be the set of security levels. Finally, let F be the set of 3-tuples (f_s, f_o, f_c), where f_s and f_c associate with each subject maximum and current security levels, respectively, and f_o associates with each object a security level. The relation *dom* from Definition 5–1 is defined here in the obvious way.

The system objects may be organized as a set of hierarchies (trees and single nodes). Let H represent the set of hierarchy functions $h: O \rightarrow P(O)$.[6] These functions have two properties. Let $o_i, o_j, o_k \in O$. Then:

1. If $o_i \neq o_j$, then $h(o_i) \cap h(o_j) = \varnothing$.
2. There is no set $\{ o_1, o_2, \ldots, o_k \} \subseteq O$ such that for each $i = 1, \ldots, k$, $o_{i+1} \in h(o_i)$, and $o_{k+1} = o_1$.

(See Exercise 6.)

[5] The right called "empty" here is called "execute" in Bell and LaPadula [68]. However, they define "execute" as "neither observation nor alteration" (and note that it differs from the notion of "execute" that most systems implement). For clarity, we changed the $\underline{e}$ right's name to the more descriptive "empty."

[6] $P(O)$ is the power set of O—that is, the set of all possible subsets of O.

A state $v \in V$ of a system is a 4-tuple (b, m, f, h), where $b \in P(S \times O \times P)$ indicates which subjects have access to which objects, and what those access rights are; $m \in M$ is the access control matrix for the current state; $f \in F$ is the 3-tuple indicating the current subject and object clearances and categories; and $h \in H$ is the hierarchy of objects for the current state. The difference between b and m is that the rights in m may be unusable because of differences in security levels; b contains the set of rights that may be exercised, and m contains the set of discretionary rights.

R denotes the set of requests for access. The form of the requests affects the instantiation, not the formal model, and is not discussed further here. Four outcomes of each request are possible: $\underline{y}$ for yes (allowed), $\underline{n}$ for no (not allowed), $\underline{i}$ for illegal request, and $\underline{o}$ for error (multiple outcomes are possible). D denotes the set of outcomes. The set $W \subseteq R \times D \times V \times V$ is the set of actions of the system. This notation means that an entity issues a request in R, and a decision in D occurs, moving the system from one state in V to another (possibly different) state in V. Given these definitions, we can now define the history of a system as it executes.

Let N be the set of positive integers. These integers represent times. Let $X = R^N$ be a set whose elements x are sequences of requests, let $Y = D^N$ be a set whose elements y are sequences of decisions, and let $Z = V^N$ be a set whose elements z are sequences of states. The ith components of x, y, and z are represented as x_i, y_i, and z_i, respectively. The interpretation is that for some $t \in N$, the system is in state $z_{t-1} \in V$; a subject makes request $x_t \in R$, the system makes a decision $y_t \in D$, and as a result the system transitions into a (possibly new) state $z_t \in V$.

A system is represented as an initial state and a sequence of requests, decisions, and states. In formal terms, $\Sigma(R, D, W, z_0) \subseteq X \times Y \times Z$ represents the system, and z_0 is the initial state of the system. $(x, y, z) \in \Sigma(R, D, W, z_0)$ if and only if $(x_t, y_t, z_t, z_{t-1}) \in W$ for all $t \in N$. (x, y, z) is an *appearance* of $\Sigma(R, D, W, z_0)$.

EXAMPLE: Consider a system with two levels (HIGH and LOW), one category (ALL), one subject s, one object o, and two rights, read ($\underline{r}$) and write ($\underline{w}$). Then:

$$S = \{ s \}, O = \{ o \}, P = \{ \underline{r}, \underline{w} \}, C = \{ \text{HIGH}, \text{LOW} \}, K = \{ \text{ALL} \}$$

For every function $f \in F$, $f_c(s)$ is either (LOW, { ALL}) or (HIGH, { ALL}), and $f_o(o)$ is either (LOW, { ALL}) or (HIGH, { ALL}). Now, suppose $b_1 = \{ (s, o, \underline{r}) \}$, $m_1 \in M$ gives s read access over o, and for $f_1 \in F$, $f_{c,1}(s)$ = (HIGH, { ALL}) and $f_{o,1}(o)$ = (LOW, { ALL}). This describes a state of the system in which s has read rights to o, so $v_0 = (b_1, m_1, f_1) \in V$.

Now suppose $S = \{ s, s' \}$, $f_{s,1}(s')$ = (LOW, { ALL}), and m_1 gives s' write access over o as well as giving s read access over o. Because s' has not yet written o, b_1 is unchanged. Take $z_0 = (b_1, m_1, f_1)$ and consider the system $\Sigma(R, D, W, z_0)$. If s' makes the request r_1 to write to o, the system will decide $d_1 = \underline{y}$ (yes), and will transition to the state $v_1 = (b_2, m_1, f_1) \in V$, where $b_2 = \{ (s, o, r), (s', o, w) \}$. In this case, $x = (r_1)$, $y = (yes)$, and $z = (v_0, v_1)$.

The next request r_2 is for s to write to o; however, this is disallowed ($d_2 = \underline{n}$, or no). The resulting state is the same as the preceding one. Now $x = (r_1, r_2)$, $y = (\underline{y}, \underline{n})$, and $z = (v_0, v_1, v_2)$, where $v_2 = v_1$.

5.2.3.1 Basic Security Theorem

The Basic Security Theorem combines the simple security condition, the *-property, and a discretionary security property. We now formalize these three properties.

Formally, the simple security condition is:

Definition 5–2. $(s, o, p) \in S \times O \times P$ satisfies the *simple security condition* relative to f (written *ssc rel f*) if and only if one of the following holds:

a. $p = \underline{e}$ or $p = \underline{a}$
b. $p = \underline{r}$ or $p = \underline{w}$ and $f_s(s)$ *dom* $f_o(o)$

In other words, if s can read o (or read and write to it), s must dominate o. A state (b, m, f, h) satisfies the simple security condition if all elements of b satisfy *ssc rel f*. A system satisfies the simple security condition if all its states satisfy the simple security condition.

Define $b(s{:}\ p_1, ..., p_n)$ to be the set of all objects that s has $p_1, ..., p_n$ access to:

$$b(s{:}\ p_1, ..., p_n) = \{\ o \mid o \in O \wedge [\ (s, o, p_1) \in b \vee ... \vee (s, o, p_n) \in b\ ]\ \}$$

Definition 5–3. A state (b, m, f, h) satisfies the **-property* if and only if, for each $s \in S$, the following hold:

a. $b(s{:}\ \underline{a}) \neq \varnothing \Rightarrow [\forall\ o \in b(s{:}\ \underline{a})\ [\ f_o(o)\ dom\ f_c(s)\ ]\ ]$
b. $b(s{:}\ \underline{w}) \neq \varnothing \Rightarrow [\forall\ o \in b(s{:}\ \underline{w})\ [\ f_o(o) = f_c(s)\ ]\ ]$
c. $b(s{:}\ \underline{r}) \neq \varnothing \Rightarrow [\forall\ o \in b(s{:}\ \underline{r})\ [\ f_c(s)\ dom\ f_o(o)\ ]\ ]$

This definition says that if a subject can write to an object, the object's classification must dominate the subject's clearance ("write up"); if the subject can also read the object, the subject's clearance must be the same as the object's classification ("equality for read"). A system satisfies the *-property if all its states satisfy the *-property. In many systems, only a subset S' of subjects satisfy the *-property; in this case, we say that the *-property is satisfied relative to $S' \subseteq S$.

Definition 5–4. A state (b, m, f, h) satisfies the *discretionary security property* (*ds-property*) if and only if, for each triple $(s, o, p) \in b$, $p \in m[s, o]$.

The access control matrix allows the controller of an object to condition access based on identity. The model therefore supports both mandatory and discretionary controls, and defines "secure" in terms of both. A system satisfies the discretionary security property if all its states satisfy the discretionary security property.

Definition 5–5. A system is *secure* if it satisfies the simple security condition, the *-property, and the discretionary security property.

The notion of an *action*, or a request and decision that moves the system from one state to another, must also be formalized, as follows.

Definition 5–8. Let $\omega = \{ \rho_1, ..., \rho_m \}$ be a set of rules. For request $r \in R$, decision $d \in D$, and states $v, v' \in V$, $(r, d, v, v') \in W(\omega)$ if and only if $d \neq \underline{i}$ and there is a unique integer i, $1 \leq i \leq m$, such that $\rho_i(r, v') = (d, v)$.

This definition says that if the request is legal and there is only one rule that will change the state of the system from v to v', the corresponding action is in $W(\omega)$.

The next theorem presents conditions under which a set of rules preserves the simple security condition.

Theorem 5–7. Let ω be a set of ssc-preserving rules, and let z_0 be a state satisfying the simple security condition. Then $\Sigma(R, D, W_{(\omega)}, z_0)$ satisfies the simple security condition.

Proof By contradiction. Let $(x, y, z) \in \Sigma(R, D, W(\omega), z_0)$ be a state that does not satisfy the simple security property. Without loss of generality, choose $t \in N$ such that (x_t, y_t, z_t) is the first appearance of $\Sigma(R, D, W(\omega), z_0)$ that does not satisfy the simple security property. Because $(x_t, y_t, z_t, z_{t-1}) \in W(\omega)$, there is a unique rule $\rho \in \omega$ such that $\rho(x_t, z_{t-1}) = (y_t, z_t)$, and $y_t \neq \underline{i}$. Because ρ is ssc-preserving, and z_{t-1} satisfies the simple security condition, by Definition 5–7, z_t must meet the simple security condition. This contradicts our choice of t, and the assumption that (x, y, z) does not meet the simple security property. Hence, the theorem is proved.

When does adding a state preserve the simple security property?

Theorem 5–8. Let $v = (b, m, f, h)$ satisfy the simple security condition. Let $(s, o, p) \notin b$, $b' = b \cup \{ (s, o, p) \}$, and $v' = (b', m, f, h)$. Then v' satisfies the simple security condition if and only if either of the following conditions is true.

a. Either $p = \underline{e}$ or $p = \underline{a}$.
b. Either $p = \underline{r}$ or $p = \underline{w}$, and $f_s(s)$ *dom* $f_o(o)$.

Proof For (a), the theorem follows from Definition 5–2 and v' satisfying *ssc rel f*. For (b), if v' satisfies the simple security condition, then, by definition, $f_s(s)$ *dom* $f_o(o)$. Moreover, if $f_s(s)$ *dom* $f_o(o)$, then $(s, o, p) \in b'$ satisfies *ssc rel f*; hence, v' is secure.

Similar theorems hold for the *-property:

Theorem 5–9. Let ω be a set of *-property-preserving rules, and let z_0 be a state satisfying the *-property. Then $\Sigma(R, D, W, z_0)$ satisfies the *-property.

Proof See Exercise 11.

Theorem 5–10. Let $v = (b, m, f, h)$ satisfy the *-property. Let $(s, o, p) \notin b$, $b' = b \cup \{ (s, o, p) \}$, and $v' = (b', m, f, h)$. Then v' satisfies the *-property if and only if one of the following conditions holds.

a. $p = \underline{a}$ and $f_o(o)$ *dom* $f_c(s)$
b. $p = \underline{w}$ and $f_o(o) = f_c(s)$
c. $p = \underline{r}$ and $f_c(s)$ *dom* $f_o(o)$
d. $p = \underline{e}$

Proof If v' satisfies the *-property, then the claim follows from Definition 5–3 and the definition of $\underline{e}$. Conversely, assume that conditions (a) holds. Let $(s', o', p') \notin b'$. If $(s', o', p') \in b$, the assumption that v satisfies the *-property means that v' also satisfies the *-property. Otherwise, $(s', o', p') = (s, o, p)$ and, by condition (a) and Definition 5–3, the *-property holds. The proof for each of the other conditions is similar. Thus, v' satisfies the *-property.

Theorem 5–11. Let ω be a set of ds-property-preserving rules, and let z_0 be a state satisfying the ds-property. Then $\Sigma(R, D, W(\omega), z_0)$ satisfies the ds-property.

Proof See Exercise 11.

Theorem 5–12. Let $v = (b, m, f, h)$ satisfy the ds-property. Let $(s, o, p) \notin b$, $b' = b \cup \{ (s, o, p) \}$, and $v' = (b', m, f, h)$. Then v' satisfies the ds-property if and only if $p \in m[s, o]$.

Proof If v' satisfies the ds-property, then the claim follows immediately from Definition 5–4. Conversely, assume that $p \in m[s, o]$. Because $(s', o', p') \in b'$, the ds-property holds for v'. Thus, v' satisfies the ds-property.

Finally, we present the following theorem.

Theorem 5–13. Let ρ be a rule and $\rho(r, v) = (d, v')$, where $v = (b, m, f, h)$ and $v' = (b', m', f', h')$. Then:

a. If $b' \subseteq b, f' = f$, and v satisfies the simple security condition, then v' satisfies the simple security condition.

b. If $b' \subseteq b, f' = f$, and v satisfies the *-property, then v' satisfies the *-property.

c. If $b' \subseteq b$, $m[s, o] \subseteq m'[s, o]$ for all $s \in S$ and $o \in O$, and v satisfies the ds-property, then v' satisfies the ds-property.

Proof Suppose that v satisfies the simple security property. Because $b' \subseteq b$, $(s, o, \underline{r}) \in b'$ implies $(s, o, \underline{r}) \in b$, and $(s, o, \underline{w}) \in b'$ implies $(s, o, \underline{w}) \in b$. So $f_s(s)$ *dom* $f_o(o)$. But $f' = f$. Thus, $f_s'(s)$ *dom* $f_o'(o)$. So v' satisfies the simple security condition.

The proofs of the other two parts are analogous.

5.2.4 Example Model Instantiation: Multics

We now examine the modeling of specific actions. The Multics system [68, 788] has 11 rules affecting the rights on the system. These rules are divided into five groups. Let the set Q contain the set of request operations (such as *get*, *give*, and so forth). Then:

1. $R^{(1)} = Q \times S \times O \times M$. This is the set of requests to request and release access. The rules are *get-read*, *get-append*, *get-execute*, *get-write*, and *release-read/execute/write/append*. These rules differ in the conditions necessary for the subject to be able to request the desired right. The rule *get-read* is discussed in more detail in Section 5.2.4.1.
2. $R^{(2)} = S \times Q \times S \times O \times M$. This is the set of requests to give access to and remove access from a different subject. The rules are *give-read/execute/write/append* and *rescind-read/execute/write/append*. Again, the rules differ in the conditions needed to acquire and delete the rights, but within each rule, the right being added or removed does not affect the conditions. Whether the right is being added or deleted does affect them. The rule *give-read/execute/write/append* is discussed in more detail in Section 5.2.4.2.
3. $R^{(3)} = Q \times S \times O \times L$. This is the set of requests to create and reclassify objects. It contains the *create-object* and *change-object-security-level* rules. The object's security level is either assigned (*create-object*) or changed (*change-object-security-level*).
4. $R^{(4)} = S \times O$. This is the set of requests to remove objects. It contains only the rule *delete-object-group*, which deletes an object and all objects beneath it in the hierarchy.
5. $R^{(5)} = S \times L$. This is the set of requests to change a subject's security level. It contains only the rule *change-subject-current-security-level*, which changes a subject's current security level (not the maximum security level).

Then, the set of requests $R = R^{(1)} \cup R^{(2)} \cup R^{(3)} \cup R^{(4)} \cup R^{(5)}$.

The Multics system includes the notion of trusted users. The system does not enforce the *-property for this set of subjects $S_T \subseteq S$; however, members of S_T are trusted not to violate that property.

For each rule ρ, define $\Delta(\rho)$ as the domain of the request (that is, whether or not the components of the request form a valid operand for the rule).

We next consider two rules in order to demonstrate how to prove that the rules preserve the simple security property, the *-property, and the discretionary security property.

5.2.4.1 The *get-read* Rule

The *get-read* rule enables a subject s to request the right to read an object o. Represent this request as $r = (get, s, o, \underline{r}) \in R^{(1)}$, and let the current state of the system be $v = (b, m, f, h)$. Then *get-read* is the rule $\rho_1(r, v)$:

```
**if** $(r \notin \Delta(\rho_1))$ **then** $\rho_1(r, v) = (\underline{i}, v)$;
**else if** $(f_s(s)\ dom\ f_o(o)$ **and** $[s \in S_T$ **or** $f_c(s)\ dom\ f_o(o)]$ **and** $\underline{r} \in m[s, o])$
        **then** $\rho_1(r, v) = (\underline{y}, (b \cup \{ (s, o, \underline{r}) \}, m, f, h))$;
**else** $\rho_1(r, v) = (\underline{n}, v)$;
```

The first *if* tests the parameters of the request; if any of them are incorrect, the decision is "illegal" and the system state remains unchanged. The second *if* checks three conditions. The simple security property for the maximum security level of the subject and the classification of the object must hold. Either the subject making the request must be trusted, or the *-property must hold for the *current* security level of the subject (this allows trusted subjects to read information from objects above their current security levels but at or below their maximum security levels; they are trusted not to reveal the information inappropriately). Finally, the discretionary security property must hold. If these three conditions hold, so does the Basic Security Theorem. The decision is "yes" and the system state is updated to reflect the new access. Otherwise, the decision is "no" and the system state remains unchanged.

We now show that if the current state of the system satisfies the simple security condition, the *-property, and the ds-property, then after the *get-read* rule is applied, the state of the system also satisfies those three conditions.

Theorem 5–14. The *get-read* rule ρ_1 preserves the simple security condition, the *-property, and the ds-property.

Proof Let v satisfy the simple security condition, the *-property, and the ds-property. Let $\rho_1(r, v) = (d, v')$. Either $v' = v$ or $v' = (b \cup \{ (s_2, o, \underline{r}) \}, m, f, h)$, by the *get-read* rule. In the former case, because v satisfies the simple security condition, the *-property, and the ds-property, so does v'. So let $v' = (b \cup \{ (s_2, o, \underline{r}) \}, m, f, h)$.

Consider the simple security condition. From the choice of v', either $b' - b = \varnothing$ or $b' - b = \{ (s_2, o, \underline{r}) \}$. If $b' - b = \varnothing$, then $\{ (s_2, o, \underline{r}) \} \in b$, so $v = v'$, proving that v' satisfies the simple security condition. Otherwise, because the *get-read* rule requires that $f_c(s)\ dom\ f_o(o)$, Theorem 5–8 says that v' satisfies the simple security condition.

Consider the *-property. From the definition of the *get-read* rule, either $s \in S_T$ or $f_s(s)\ dom\ f_o(o)$. If $s \in S_T$, then s is trusted and the *-property holds by the definition of S_T. Otherwise, by Theorem 5–10, because $f_c(s)\ dom\ f_o(o)$, v' satisfies the *-property.

Finally, consider the ds-property. The condition in the *get-read* rule requires that $\underline{r} \in m[s, o]$ and $b' - b = \varnothing$ or $b' - b = \{ (s_2, o, \underline{r}) \}$. If $b' - b = \varnothing$, then $\{ (s_2, o, \underline{r}) \} \in b$, so $v = v'$, proving that v' satisfies the ds-property. Otherwise, $\{ (s_2, o, \underline{r}) \} \notin b$, which meets the conditions of Theorem 5–12. From that theorem, v' satisfies the ds-property.

Hence, the *get-read* rule preserves the security of the system.

5.2.4.2 The *give-read* Rule

The *give-read* rule[7] enables a subject s to give subject s_2 the (discretionary) right to read an object o. Conceptually, a subject can give another subject read access to an object if the giver can alter (write to) the parent of the object. If the parent is the root of the hierarchy containing the object, or if the object itself is the root of the hierarchy, the subject must be specially authorized to grant access.

Some terms simplify the definitions and proofs. Define *root*(o) as the root object of the hierarchy h containing o, and define *parent*(o) as the parent of o in h. If the subject is specially authorized to grant access to the object in the situation just mentioned, the predicate *canallow*(s, o, v) is true. Finally, define $m \wedge m[s, o] \leftarrow \underline{r}$ as the access control matrix m with the right $\underline{r}$ added to entry $m[s, o]$.

Represent the *give-read* request as $r = (s_1, give, s_2, o, \underline{r}) \in R^{(2)}$, and let the current state of the system be $v = (b, m, f, h)$. Then, *give-read* is the rule $\rho_6(r, v)$:

```
**if** $(r \notin \Delta(\rho_6))$ **then** $\rho_6(r, v) = (\underline{i}, v)$;
**else if** ( [ $o \neq root(o)$ **and** $parent(o) \neq root(o)$ **and** $parent(o) \in b(s_1\!: \underline{w})$] **or**
        [ $parent(o) = root(o)$ **and** $canallow(s_1, o, v)$ ] **or**
        [ $o = root(o)$ and $canallow(s_1, root(o), v)$ ])
    **then** $\rho_6(r, v) = (\underline{y}, (b, m \wedge m[s_2, o] \leftarrow \underline{r}, f, h))$;
**else** $\rho_6(r, v) = (\underline{n}, v)$;
```

The first *if* tests the parameters of the request; if any of them are incorrect, the decision is "illegal" and the system state remains unchanged. The second *if* checks several conditions. If neither the object nor its parent is the root of the hierarchy containing the object, then s_1 must have write rights to the parent. If the object or its parent is the root of the hierarchy, then s_1 must have special permission to give s_2 the read right to o. The decision is "yes" and the access control matrix is updated to reflect the new access. Otherwise, the decision is "no" and the system state remains unchanged.

We now show that if the current state of the system satisfies the simple security condition, the *-property, and the ds-property, then after the *give-read* rule is applied, the state of the system also satisfies those three conditions.

[7] Actually, the rule is *give-read/execute/write/append*. The generalization is left as an exercise for the reader.

Further, what is captured by the [Basic Security Theorem] is so trivial that it is hard to imagine a realistic security model for which it does not hold" ([682], p. 47). The basis for McLean's argument was that given assumptions known to be nonsecure, the Basic Security Theorem could prove a nonsecure system to be secure. He defined a complement to the *-property:

Definition 5–11. A state (b, m, f, h) satisfies the *†-property* if and only if, for each subject $s \in S$, the following conditions hold:

a. $b(s\text{: }\underline{a}) \neq \varnothing \Rightarrow [\forall\, o \in b(s\text{: }\underline{a})\ [\, f_c(s)\ dom\ f_o(o)\ ]\ ]$
b. $b(s\text{: }\underline{w}) \neq \varnothing \Rightarrow [\forall\, o \in b(s\text{: }\underline{w})\ [\, f_c(s) = f_o(o)\ ]\ ]$
c. $b(s\text{: }\underline{r}) \neq \varnothing \Rightarrow [\forall\, o \in b(s\text{: }\underline{r})\ [\, f_c(s)\ dom\ f_o(o)\ ]\ ]$

In other words, the †-property holds for a subject s and an object o if and only if the clearance of s dominates the classification of o. This is exactly the reverse of the *-property, which holds if and only if the classification of o dominates the clearance of s. A state satisfies the †-property if and only if, for every triplet (s, o, p), where the right p involves writing (that is, $p = \underline{a}$ or $p = \underline{w}$), the †-property holds for s and o.

McLean then proved the analogue to Theorem 5–4:

Theorem 5–16. $\Sigma(R, D, W, z_0)$ satisfies the †-property relative to $S' \subseteq S$ for any secure state z_0 if and only if, for every action $(r, d, (b, m, f, h), (b', m', f', h'))$, W satisfies the following conditions for every $s \in S'$:

a. Every $(s, o, p) \in b - b'$ satisfies the †-property with respect to S'.
b. Every $(s, o, p) \in b'$ that does not satisfy the †-property with respect to S' is not in b.

Proof See Exercise 8, with "*-property" replaced by "†-property."

From this theorem, and from Theorems 5–3 and 5–5, the analogue to the Basic Security Theorem follows.

Theorem 5–17. ***Basic Security Theorem:*** $\Sigma(R, D, W, z_0)$ is a secure system if and only if z_0 is a secure state and W satisfies the conditions of Theorems 5–3, 5–16, and 5–5.

However, the system $\Sigma(R, D, W, z_0)$ is clearly nonsecure, because a subject with HIGH clearance can write information to an object with LOW classification. Information can flow down, from HIGH to LOW. This violates the basic notion of security in the confidentiality policy.

Consider the role of the Basic Security Theorem in the Bell-LaPadula Model. The goal of the model is to demonstrate that specific rules, such as the *get-read* rule, preserve security. But what is security? The model defines that term using the Basic Security Theorem: an instantiation of the model is secure if and only if the initial state satisfies the simple security condition, the *-property, and the ds-property, and the transition rules preserve those properties. In essence, the theorems are assertions about the three properties.

The rules describe the changes in a *particular* system instantiating the model. Showing that the system is secure, as defined by the analogue of Definition 5–5, requires proving that the rules preserve the three properties. Given that they do, the Basic Security Theorem asserts that reachable states of the system will also satisfy the three properties. The system will remain secure, given that it starts in a secure state.

LaPadula pointed out that McLean's statement does not reflect the assumptions of the Basic Security Theorem [617]. Specifically, the Bell-LaPadula Model assumes that a transition rule introduces no changes that violate security, but does *not* assume that any *existing* accesses that violate security are eliminated. The rules instantiating the model do no elimination (see the *get-read* rule, Section 5.2.4.1, as an example).

Furthermore, the *nature* of the rules is irrelevant to the model. The model accepts a definition of "secure" as axiomatic. The specific *policy* defines "security" and is an instantiation of the model. The Bell-LaPadula Model uses a military definition of security: information may not flow from a dominating entity to a dominated entity. The *-property captures this requirement. But McLean's variant uses a different definition: rather than meet the *-property, his policy requires that information not flow from a dominated entity to a dominating entity. This is not a confidentiality policy. Hence, a system satisfying McLean's policy will not satisfy a confidentiality policy. McLean's argument eloquently makes this point.

However, the sets of properties in both policies (the confidentiality policy and McLean's variant) are inductive, and the Basic Security Theorem holds. The properties may not make sense in a real system, but this is irrelevant to the model. It is *very* relevant to the *interpretation* of the model, however. The confidentiality policy requires that information not flow from a dominating subject to a dominated object. McLean substitutes a policy that allows this. These are alternative instantiations of the model.

McLean makes these points by stating problems that are central to the use of any security model. The model must abstract the notion of security that the system is to support. For example, McLean's variant of the confidentiality policy does not provide a correct definition of security for military purposes. An analyst examining a system could not use this variant to show that the system implemented a confidentiality classification scheme. The Basic Security Theorem, and indeed *all* theorems, fail to capture this, because the definition of "security" is axiomatic. The analyst must establish an appropriate definition. All the Basic Security Theorem requires is that the definition of security be inductive.

McLean's second observation asks whether an analyst can prove that the system being modeled meets the definition of "security." Again, this is beyond the province of the model. The model makes claims based on hypotheses. The issue is whether the hypotheses hold for a real system.

5.4.2 McLean's System Z and More Questions

In a second paper [683], McLean sharpened his critique. System transitions can alter any system component, including b, f, m, and h, as long as the new state does not violate security. McLean used this property to demonstrate a system, called System Z, that satisfies the model but is not a confidentiality security policy. From this, he concluded that the Bell-LaPadula Model is inadequate for modeling systems with confidentiality security policies.

System Z has the weak tranquility property and supports exactly one action. When a subject requests any type of access to any object, the system downgrades *all* subjects and objects to the lowest security level, adds access permission to the access control matrix, and allows the access.

Let System Z's initial state satisfy the simple security condition, the *-property, and the ds-property. It can be shown that successive states of System Z also satisfy those properties and hence System Z meets the requirements of the Basic Security Theorem. However, with respect to the confidentiality security policy requirements, the system clearly is not secure, because all entities are downgraded.

McLean reformulated the notion of a secure action. He defined an alternative version of the simple security condition, the *-property, and the discretionary security property. Intuitively, an action satisfies these properties if, given a state that satisfies the properties, the action transforms the system into a (possibly different) state that satisfies these properties, *and eliminates any accesses present in the transformed state that would violate the property in the initial state*. From this, he shows:

Theorem 5–18. $\Sigma(R, D, W, z_0)$ is a secure system if z_0 is a secure state and each action in W satisfies the alternative versions of the simple security condition, the *-property, and the discretionary security property.

Proof See [683].

Under this reformulation, System Z is not secure because this rule is not secure. Specifically, consider an instantiation of System Z with two security clearances, (HIGH, { ALL }) and (LOW, { ALL }) (HIGH > LOW). The initial state has a subject s and an object o. Take $f_c(s)$ = (LOW, { ALL }), $f_o(o)$ = (HIGH, { ALL }), $m[s, o] = \{ \underline{w} \}$, and $b = \{ (s, o, \underline{w}) \}$. When s requests read access to o, the rule transforms the system into a state wherein $f_o'(o)$ = (LOW, { ALL }), $(s, o, \underline{r}) \in b'$, and $m'[s, o] = \{ \underline{r}, \underline{w} \}$. However, because $(s, o, \underline{r}) \in b' - b$ and $f_o(o)$ *dom* $f_s(s)$, an illegal access has been added. Yet, under the traditional Bell-LaPadula formulation, in the final state $f_c'(s) = f_o'(o)$, so the read access is legal and the state is secure, hence the system is secure.

McLean's conclusion is that proving that states are secure is insufficient to prove the security of a system. One must consider both states and transitions.

Bell [64] responded by exploring the fundamental nature of modeling. Modeling in the physical sciences abstracts a physical phenomenon to its fundamental properties. For example, Newtonian mathematics coupled with Kepler's laws of planetary motion provide an abstract description of how planets move. When observers noted that Uranus did not follow those laws, they calculated the existence of another, trans-Uranean planet. Adams and Lavoisier, observing independently, confirmed its existence. Refinements arise when the theories cannot adequately account for observed phenomena. For example, the precession of Mercury's orbit suggested another planet between Mercury and the sun. But none was found.[8] Einstein's theory of general relativity, which modified the theory of how planets move, explained the precession, and observations confirmed his theory.

Modeling in the foundations of mathematics begins with a set of axioms. The model demonstrates the consistency of the axioms. A model consisting of points, lines, planes, and the axioms of Euclidean geometry can demonstrate the consistency of those axioms. Attempts to prove the inconsistency of a geometry created without the Fifth Postulate[9] failed; eventually, Riemann replaced the plane with a sphere, replaced lines with great circles, and using that model demonstrated the consistency of the axioms (which became known as "Riemannian geometry"). Gödel demonstrated that consistency cannot be proved using only axioms within a system (hence Riemannian geometry assumes the consistency of Euclidean geometry, which in turn assumes the consistency of another axiomatizable system, and so forth). So this type of modeling has natural limits.

The Bell-LaPadula Model was developed as a model in the first sense. Bell pointed out that McLean's work presumed the second sense.

In the first sense of modeling, the Bell-LaPadula Model is a tool for demonstrating certain properties of rules. Whether the properties of System Z are desirable is an issue the model cannot answer. If no rules should change security compartments of entities, the system should enforce the principle of strong tranquility. System Z clearly violates this principle, and hence would be considered not secure. (The principle of tranquility adds requirements to state transitions, so given that principle, the Bell-LaPadula Model actually constrains both states and state transitions.)

In the second sense, Bell pointed out that the two models (the original Bell-LaPadula Model, and McLean's variant) define security differently. Hence, that System Z is not secure under one model, but secure under the other, is not surprising. As an example, consider the following definitions of *prime number.*

Definition 5–12. A *prime number* is an integer $n > 1$ that has only 1 and itself as divisors.

Definition 5–13. A *prime number* is an integer $n > 0$ that has only 1 and itself as divisors.

[8] Observers reported seeing this planet, called Vulcan, in the mid-1800s. The sighting was never officially confirmed, and the refinements discussed above explained the precession adequately. Willy Ley's book [625] relates the charming history of this episode.

[9] The Fifth Postulate of Euclid states that given a line and a point, there is exactly one line that can be drawn through that point parallel to the existing line. Attempts to prove this postulate failed; in the 1800s, Riemann and Lobachevsky demonstrated the axiomatic nature of the postulate by developing geometries in which the postulate does not hold [774].

Both definitions, from a mathematical point of view, are acceptable and consistent with the laws of mathematics. So, is the integer 1 prime? By Definition 5–12, no; by Definition 5–13, yes. Neither answer is "right" or "wrong" in an absolute sense.[10]

5.4.3 Summary

McLean's questions and observations about the Bell-LaPadula Model raised issues about the foundations of computer security, and Bell and LaPadula's responses fueled interest in those issues. The annual Foundations of Computer Security Workshop began shortly after to examine foundational questions.

5.5 Summary

The influence of the Bell-LaPadula Model permeates all policy modeling in computer security. It was the first mathematical model to capture attributes of a real system in its rules. It formed the basis for several standards, including the Department of Defense's Trusted Computer System Evaluation Criteria (the TCSEC or the "Orange Book" discussed in Chaper 21) [285]. Even in controversy, the model spurred further studies in the foundations of computer security.

Other models of confidentiality arise in practical contexts. They may not form lattices. In this case, they can be embedded into a lattice model. Still other confidentiality models are not multilevel in the sense of Bell-LaPadula. These models include integrity issues, and Chapter 7, "Hybrid Policies," discusses several.

Confidentiality models may be viewed as models constraining the way information moves about a system. The notions of noninterference and nondeducibility provide an alternative view that in some ways matches reality better than the Bell-LaPadula Model; Chapter 8, "Noninterference and Policy Composition," discusses these models.

5.6 Research Issues

Research issues in confidentiality arise in the application of multilevel security models. One critical issue is the inclusion of declassification within the model (as opposed to being an exception, allowed by a trusted user such as the system security

[10] By convention, mathematicians use Definition 5–12. The integer 1 is neither prime nor composite.

Chapter 6
Integrity Policies

ISABELLA: Some one with child by him? My cousin Juliet?
LUCIO: Is she your cousin?
ISABELLA: Adoptedly; as school-maids change their names
By vain, though apt affection.
—*Measure for Measure*, I, iv, 45–48.

An inventory control system may function correctly if the data it manages is released; but it cannot function correctly if the data can be randomly changed. So integrity, rather than confidentiality, is key. Integrity policies focus on integrity rather than confidentiality, because most commercial and industrial firms are more concerned with accuracy than disclosure. In this chapter we discuss the major integrity security policies and explore their design.

6.1 Goals

Commercial requirements differ from military requirements in their emphasis on preserving data integrity. Lipner [636] identifies five requirements:

1. Users will not write their own programs, but will use existing production programs and databases.
2. Programmers will develop and test programs on a nonproduction system; if they need access to actual data, they will be given production data via a special process, but will use it on their development system.
3. A special process must be followed to install a program from the development system onto the production system.
4. The special process in requirement 3 must be controlled and audited.
5. The managers and auditors must have access to both the system state and the system logs that are generated.

These requirements suggest several principles of operation.

First comes *separation of duty*. The principle of separation of duty states that if two or more steps are required to perform a critical function, at least two different people should perform the steps. Moving a program from the development system to the production system is an example of a critical function. Suppose one of the application programmers made an invalid assumption while developing the program. Part of the installation procedure is for the installer to certify that the program works "correctly," that is, as required. The error is more likely to be caught if the installer is a different person (or set of people) than the developer. Similarly, if the developer wishes to subvert the production data with a corrupt program, the certifier either must not detect the code to do the corruption, or must be in league with the developer.

Next comes *separation of function*. Developers do not develop new programs on production systems because of the potential threat to production data. Similarly, the developers do not process production data on the development systems. Depending on the sensitivity of the data, the developers and testers may receive sanitized production data. Further, the development environment must be as similar as possible to the actual production environment.

Last comes *auditing*. Commercial systems emphasize recovery and accountability. Auditing is the process of analyzing systems to determine what actions took place and who performed them. Hence, commercial systems must allow extensive auditing and thus have extensive logging (the basis for most auditing). Logging and auditing are especially important when programs move from the development system to the production system, since the integrity mechanisms typically do not constrain the certifier. Auditing is, in many senses, external to the model.

Even when disclosure is at issue, the needs of a commercial environment differ from those of a military environment. In a military environment, clearance to access specific categories and security levels brings the ability to access information in those compartments. Commercial firms rarely grant access on the basis of "clearance"; if a particular individual needs to know specific information, he or she will be given it. While this can be modeled using the Bell-LaPadula Model, it requires a large number of categories and security levels, increasing the complexity of the modeling. More difficult is the issue of controlling this proliferation of categories and security levels. In a military environment, creation of security levels and categories is centralized. In commercial firms, this creation would usually be decentralized. The former allows tight control on the number of compartments, whereas the latter allows no such control.

More insidious is the problem of information aggregation. Commercial firms usually allow a limited amount of (innocuous) information to become public, but keep a large amount of (sensitive) information confidential. By aggregating the innocuous information, one can often deduce much sensitive information. Preventing this requires the model to track what questions have been asked, and this complicates the model enormously. Certainly the Bell-LaPadula Model lacks this ability.

Security

COMPUTER SECURITY

"This is an excellent text that should be read by every computer security professional and student."

—Dick Kemmerer, University of California, Santa Barbara

"This is the most complete book on information security theory, technology, and practice that I have encountered anywhere!"

—Marvin Schaefer, Former Chief Scientist, National Computer Security Center, NSA

This highly anticipated book fully introduces the theory and practice of computer security. It is both a comprehensive text, explaining the most fundamental and pervasive aspects of the field, and a detailed reference filled with valuable information for even the most seasoned practitioner. In this one extraordinary volume the author incorporates concepts from computer systems, networks, human factors, and cryptography. In doing so, he effectively demonstrates that computer security is an art as well as a science.

Computer Security: Art and Science includes detailed discussions on:

- The nature and challenges of computer security
- The relationship between policy and security
- The role and application of cryptography
- The mechanisms used to implement policies
- Methodologies and technologies for assurance
- Vulnerability analysis and intrusion detection

Computer Security discusses different policy models, and presents mechanisms that can be used to enforce these policies. It concludes with examples that show how to apply the principles discussed in earlier sections, beginning with networks and moving on to systems, users, and programs.

This important work is essential for anyone who needs to understand, implement, or maintain a secure network or computer system.

Matt Bishop is Associate Professor in the Department of Computer Science at the University of California at Davis. He is widely recognized as an expert in vulnerability analysis, the design of secure systems and software, network security, formal models of access control, user authentication, and UNIX security. A well-known educator, Matt is focused on improving the teaching of computer security. He earned his Ph.D. from Purdue University in 1984.

http://www.awprofessional.com

Cover design by Karin Hansen
Cover photograph by G.I. Bernard, Oxford Scientific Films

Text printed on recycled paper

Addison-Wesley
Pearson Education

57999
9 780201 440997
ISBN 0-201-44099-7

$79.99 US
$120.99 CANADA