A Further Note on the Confinement Problem

William E. Boebert
Sandia National Laboratories
Albuquerque, New Mexico

Richard Y. Kain
University of Minnesota
Minneapolis, Minnesota

ABSTRACT

"Confinement," in computer systems, is the ability to limit the amount of damage that can be done by malicious or malfunctioning software. Confinement is a requirement when either security or safety is a concern.

In this paper we demonstrate why the access control mechanisms of common operating systems do not constitute a confinement mechanism. We describe the early confinement mechanism mandated by the Trusted Computing System Evaluation Criteria and note its shortcomings. We summarize prior attempts to overcome those shortcomings. We describe an alternative confinement mechanism called "type enforcement" that was invented by the authors in 1984 and subsequently implemented in several secure computers. We show how type enforcement overcomes the limitations of the early mechanisms and outline its uses, with special emphasis on the way in which the type enforcement mechanism supports assurance and safety. We conclude by describing the application of type enforcement to the problem of confining the actions of "mobile agents," which are active agents downloaded to client machines from servers.

INTRODUCTION

Increases in the networking and sharing among computer systems increases one's exposure to people desiring unauthorized access to data stored within the systems. Even before computers were in widespread use, many organizations developed rules and regulations that served to limit document access and thus to limit the risk of unauthorized modification or revelation of the contents of the documents. A careful analysis reveals, however, that several of these schemes that have intuitive appeal are, in fact, inherently flawed.

As they specify systems to be interconnected into the 21st century, contemporary users and designers must become aware of these flaws and of modern approaches to guaranteeing that file access is indeed limited according to the proscribed policies of the organization. In this paper we show how type enforcement mechanisms can satisfy many confinement requirements; in particular, they can be used to build firewalls and to confine the access from applets arriving across the network.

The paper starts with a review of the confinement problem and the military classification and access limitation rules, illustrating their positive and negative aspects. After showing the difficulties and limitations of minimal access control schemes, we survey implementations of limited access control schemes. Type enforcement approaches to system specification, design and implementation greatly increase the assurance provided by the system; we show how this important approach has been used to build operational systems and how it can increase one's level of assurance that the system does indeed support the organization's access limitation policies. We also show how type enforcement can be used to confine, in accordance with local access policies, the activities of mobile agents, such as applets arriving across the network.

CONFINEMENT

Background

"Confinement," as used in this paper, is the property of a computer system that permits it to limit the actions of programs in execution. It was first discussed in the early 1970's [26], and consideration of confinement mechanisms was a central concern of research into secure computer systems. Few, if any, such mechanisms made their way into "mainstream" operating systems.

The importance of confinement has been heightened recently with the advent of "mobile agents" or "applets." [19] These are objects containing executable or interpretable commands and state which are transferred from a server to a client and executed locally at the client. The potential for abuse or damage as a consequence of malfunction is significant.

The Discretionary Trojan Horse Attack

The need for a confinement mechanism first became apparent when researchers noted an important inherent limitation of discretionary (or identity-based) access control mechanisms [31]. A simple example of such a mechanism has the following elements:

1. Each process executing in a system has associated with it the name of the user on whose behalf it is executing. As we will see later, the "on behalf of" relationship can become quite complex; for purposes of this example, assume a simple multiuser system such as an early UNIX[1] installation. Users log in and spawn processes as a consequence of invoking commands from a shell. The system maintains the association between the user name and these processes.
2. Each file on the system has associated with it an access control list. The entries in the list consist of pairs of the form (name, access right) where an access right is one or more of a fixed set such as [read, write, execute].
3. When a process seeks a particular mode of access to a file (e.g., "open for read") the system searches the access control list for an entry corresponding to the user on whose behalf the process is executing[2]. It then compares the allowed access modes from the access control list entry with the requested access mode, and proceeds or aborts as appropriate. Thus if a process executing on behalf of user "Smith" asked to "open for read" file "foo," the access control list for "foo" would be searched for an entry that read at least (Smith, read). If such an entry was found, the file would be opened; if no such entry was found, a well-defined error response would result.

The above mechanism appears relatively strong, and in particular appeared to its early implementors as being sufficient to protect data in a multiuser environment such as VMS, Multics or UNIX. The strength of the access control mechanism proved illusory, however, as demonstrated by the following attack:

1. The target of the attack, whom we will call "Smith," has a file containing valuable data, which we will call "hotstuff." Smith sets the access control list for "hotstuff" to contain a single entry, reading (Smith, read/write). This gives the illusion that no other user on the system can gain access to "hotstuff."
2. The attacker, who we will call "Drake," first makes a temporary file we will call "backpocket." Drake places two entries in the access control list for "backpocket:" (Smith, write) and (Drake, read). Drake does not inform Smith of the existence of this file.

1. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd.
2. So-called "wildcard" characters can be used to eliminate the need to denote every allowed user explicitly.

0-7803-3537-6-9/96/$4.00 ©1996 IEEE

3. Drake then constructs an attractive program, which we will call "Lure." Besides its normal function (such as a game or other desirable utility) the program contains a malicious subset called a "Trojan Horse.[3]" The Trojan Horse has the effect of extracting data from "hotstuff" and copying it to "backpocket."
4. Drake then contrives to get Smith to invoke Lure, containing the Trojan Horse. At this instant, the Trojan Horse is executing on Smith's behalf, and the access control mechanism will therefore permit the leak to "backpocket" after going through the following actions and checks:

- Trojan Horse: Open "hotstuff" for read. Checker: The process executes on behalf of Smith, and access control list entry is found (Smith, read/write), so permission is granted.
- Trojan Horse: Open "backpocket" for write. Checker: The process executes on behalf of Smith, and an access control list entry is found (Smith, write), so permission is granted. "Hotstuff" may now be copied into "backpocket."
- Drake: Open "backpocket" for read. Checker: The process executes on behalf of Drake, an access control list entry is found (Drake, read), so permission is granted.

The net effect is that there has then been a transfer of data from Smith to Drake in violation of the intent (but not the effect) of Smith's access control list configuration. The transfer occurred for two reasons: the existence of a mechanism for sharing program text, and the characteristic of the discretionary access control mechanism that a program in execution on behalf of an individual inherits the privileges of that individual without limitation.

The reader is encouraged to study this example until it is fully understood, because the operation of the discretionary Trojan Horse attack is as basic to understanding the confinement problem as the use of frequency counts is to the understanding of cryptanalysis.

Introducing mobile agents simply facilitates the invocation of programs containing the Trojan Horse. For example, Trojan Horses can be embedded in "applets" whose invocation is an invisible side effect of browsing. One hopes that browsing appears to be passive and benign with respect to data files. However, if, as will usually be the case, the browser is operating on behalf of a user who thinks s/he is protected by a discretionary access control mechanism, the "applet" will inherit all the privileges of that user and be able to examine, modify, or destroy data at will, contrary to the perceived protection.

Minimal MLS Confinement Mechanisms

Background

When the limitations of the pure discretionary model became apparent[4], researchers sought other mechanisms to achieve confinement [41]. Since the bulk of the work was supported by the Department of Defense, the next set of mechanisms centered on the prevalent concepts of data classification and personnel clearance.

To support these mechanisms, additional attributes are required. In addition to the access control list described above, data files, or "objects," in a system are assigned a label consisting of two parts: a hierarchical level and a set of non-hierarchical categories[5]. These labels were intended to reflect the classification of the data. The hierarchical level was intended to reflect the well-known levels of UNCLASSIFIED, CONFIDENTIAL, SECRET, and TOP SECRET; the non-hierarchical categories were intended to reflect special classifications for highly segregated data, where clearance to one such category did not imply clearance to any others. (This can be contrasted with hierarchical levels, where a clearance at the SECRET level implies the right to read CONFIDENTIAL data).

Processes, or "subjects," were assigned a similarly structured attribute, which was intended to reflect the clearance of the person on whose behalf the process was executing.

3. This term was coined by Dan Edwards of NSA.
4. A discussion of formal limitations of discretionary access control appears in [23]
5. Confusingly, the entire field is often referred to as the "sensitivity level"

An algorithm was developed that used the classification and clearance attributes to determine the allowed modes of access of a process to an object [1] [4]. This algorithm was presumed to be implemented in the "Trusted Computing Base," or "TCB" [2]. The algorithm was named "Mandatory Access Control" because it overrode any permissions that may have been given in the access control list. The most prevalent statement of this algorithm is in the Trusted Computing System Evaluation Criteria (TCSEC) which is used as a basis for U.S. Government ratings of system security. The wording below describes the mandatory access control requirement for class B1 systems; higher classes differ only in detail.

The TCB shall enforce a mandatory access control policy over all subjects and storage objects under its control (e.g., processes, files, segments, devices). These subjects and objects shall be assigned sensitivity labels that are a combination of hierarchical classification levels and non-hierarchical categories, and the labels shall be used as the basis for mandatory access control decisions. The TCB shall be able to support two or more such security levels. (See the mandatory access control guidelines.) The following requirements shall hold for all accesses between subjects and objects controlled by the TCB: a subject can read an object only if the hierarchical classification in the subject's security level is greater than or equal to the hierarchical classification in the object's security level and the non- hierarchical categories in the subject's security level include all the non-hierarchical categories in the object's security level. A subject can write an object only if the hierarchical classification in the subject's security level is less than or equal to the hierarchical classification in the object's security level and all the non-hierarchical categories in the subject's security level are included in the non- hierarchical categories in the object's security level.

The algorithm is also often referred to as the "MLS model" or "MLS policy," for "Multi-Level Security." We will call this bare-bones algorithm "Minimal MLS", or MMLS.

We encourage readers to observe how the MMLS algorithm stops the discretionary Trojan Horse attack described above. Assume that Smith has assigned a level of UNCLASSIFIED|SMITHS to "hotstuff," where "UNCLASSIFIED" is the hierarchical level and "SMITHS" is the single non-hierarchical category used to segregate the data. Assume further that Drake cannot initiate a process at a level other than UNCLASSIFIED, not being cleared into the category SMITHS. Note how the Minimal MLS algorithm prevents a the Trojan Horse from simultaneously being able to read "hotstuff" and write into "backpocket" that Drake could read, no matter whether Smith logs in at UNCLASSIFIED or UNCLASSIFIED|SMITHS. A complete explanation appears in [43].

Limits of the MMLS Algorithm

Pipelines: A useful structure that is encountered repeatedly in practical secure systems design is a "pipeline." In its simplest form, a pipeline consists of a producer process and a consumer process separated by an intermediate filter or other process whose operation is essential for security. The processes are connected by files that act as buffers, and some suitable mechanism for synchronization of reads and writes is imposed. An example would be a producer process which is a data preparation subsystem and a consumer process which is a network manager. The intermediate process would be a cryptographic transform. The pipeline is structured to insure that all data is encrypted before it goes out on the network. However, if this structure could be bypassed, the data could be sent out on the network as cleartext.

The MMLS algorithm cannot, in and of itself, enforce the pipeline structure. Any data which is readable by the cryptographic intermediary is also readable by the network manager. Thus a malfunctioning or malicious program executing with the privileges of the network manager could directly access unencrypted data and place it out on the network.

Downgrading: The MMLS Algorithm constrains information flow to the direction of ever-increasing restrictiveness. Thus data can flow from UNCLASSIFIED to UNCLASSIFIED|SMITHS but not the other way. Acknowledgments and other controlled bidirectional traffic is thereby restricted, often with severe practical effect. Also, the algorithm does not permit downgrading of data even after it has been encrypted, thus forbidding one of the most basic uses of cryptography.

Data Integrity: The MMLS algorithm permits processes to write into files they cannot read. As a consequence, the algorithm in and of itself

cannot prevent a Trojan Horse from altering or destroying data[6].

Limitations on Execute: The MMLS algorithm does not, in and of itself, impose any control on "execute" access. Thus, in the absence of other controls, it permits any data file to be executed. It further permits programs to access data unrelated to the program's stated purpose, thus facilitating Trojan Horses.

"On Behalf Of": The MMLS algorithm requires that the label of each process be derived from the privileges (clearance) of the person on whose behalf the process is executing. This requirement is difficult if not impossible to satisfy in a network environment, as opposed to the single, multiuser processing environment which existed at the time of the algorithm's development. In networks it is often quite difficult to determine on whose behalf a particular action is being taken. Consider a mail transfer agent, which examines email "envelopes" and makes routing and forwarding decisions. Is that agent acting on behalf of the local system administrator, the mail administrator, the sender of the email, the recipient of the email, or some other person? Useful answers are not easily forthcoming.

The "on behalf of" requirement, combined with the lack of control of execute access, does not support forensic studies. Flaws or vulnerabilities are almost always associated with bodies of program text.[7] A fundamental forensic question then is: "What data could this program have touched?" This question cannot be easily answered when the Minimal MLS algorithm is in effect, because the access is a function of on whose behalf the program is being run, not the identity of the program itself.

Augmented MLS

In the past, many attempts were made to overcome the above limitations in MMLS. The downgrade problem was recognized in [4], which introduced the concept of "trusted processes." A trusted process has the special privilege of downgrading data; since this privilege is very dangerous, trusted processes have to be carefully controlled. No control mechanism was specified in [4], and a variety of ad hoc approaches were developed as early systems were implemented.

The next limitation that was addressed was the lack of mechanisms to maintain data integrity. A widely-adopted mechanism is described in [7]. This mechanism involves the assignment of a second attribute, called "integrity level," to objects. These levels have a hierarchical level plus non-hierarchical category structure similar to the sensitivity levels described above. A second set of rules for access is imposed using the principles expounded in [4]. A complete description of system architecture using sensitivity levels, integrity levels, and associated rules is given in [18], which is strongly recommended to anyone wishing to understand the rationale behind this class of systems.

The other limitations of MMLS have also been addressed. A mechanism for controlling execute access is proposed in [35], and a method for modifying the mechanism of [7] to permit pipelines is proposed in [27].

Implementation of Confinement Mechanisms

There are two ways of implementing of static storage objects that have arisen in operating system design. The first way, implemented most notably in the Multics system [5], treats storage as a single homogeneous name space. Objects are accessed implicitly by referencing the object name in the body of program text. Another way of describing this is that all storage in the system appears to programs as local memory.

The second way, implemented most notably in the UNIX system [30], treats storage as named files. Objects are accessed explicitly by operating system calls (e.g., "open", "read"). Another way of describing this is that all storage on the system appears to programs as external files.

The "objects as memory" approach yields higher-assurance implementations of confinement mechanisms because the hardware used to map global names to process-local addresses (the "memory management unit" or MMU) can be modified to provide continuous enforcement of the rules [5]; since the mechanism resides in a single module, it is easier to convince people of its functionality and conformance to the system's policy regarding access limitations. The "objects as files" approach requires that the rules be enforced by the file system in a more decentralized manner; the strength of the decentralized implementation is harder to assess.

Confinement mechanisms have been proposed, and in some cases implemented, using both approaches [36] [42]. The "objects as files" approach has prevailed in the marketplace, largely because it requires less specialized hardware support and because it is more amenable to networking.

Type Enforcement

Background

The origins of the type enforcement mechanism can be traced back to a design study for a high-assurance operating system called PSOS [14], which was based on the use of access tokens called "capabilities." This project was continued in a proof-of principle implementation called the Secure Ada Target [12] [13], which evolved in turn into a system called LOCK [9]. During this evolution the use of capabilities was dropped because it required an impractical degree of special hardware support, and because capability-based designs present difficult implementation problems since the access limitations in a capability must be revised before it can be used in a new execution context [24].

In the course of these projects the authors considered many of the then-prevalent confinement mechanisms, particularly the "integrity" mechanism described in [7]. (The project specifications required conformance to the TCSEC, so the MMLS mechanism was required.) A shortcoming in [7] noted by the authors was the inability to enforce pipelines, which the authors had observed arising in a large number of contexts [10]. The specific problem which triggered the invention of type enforcement was that of verifying that a system meet the requirement that sensitivity levels be accurately included in printed output.

Description

For type enforcement we assign to each object an attribute called its type, and to each process an attribute called its domain. When a process seeks access to an object (e.g., "open for read"), a centralized table called the Domain Definition Table (DDT) is consulted. The DDT includes, conceptually, a row for each type and a column for each domain. The entry at the intersection of a row and column specifies the maximum access permissions that processes in that domain are allowed to have to objects of that type.

The DDT entries are set prior to system operation. They may be set according to any criteria the system designers choose to use; unlike MMLS, there is no need to correspond to any pre-existing structure such as clearances and classifications. The most commonly used criteria are the establishment of pipelines and the isolation of highly assured subsystems. A fuller treatment of the mechanism appears in [25].

Uses and Implications

As type enforcement made the transition from principle to practice, it became apparent that the mechanism could be applied to other confinement problems [11][17][22][39][44], and thus could support a range of applications [29][34]. Early investigations showed that the mechanism overcame the limitations of MMLS [28] and subsequent work reinforced that conclusion [3].

Prior to the development of type enforcement, the prevailing view held that all the security-relevant functionality could be concentrated in a small "security kernel." The need for trusted processes was the first indication that the establishment of the TCB perimeter was not as simple as first thought. LOCK introduced "kernel extensions," which are security-relevant modules of limited and specialized privilege, whose operations are constrained by type enforcement [8][9][32][33].

Type enforcing systems have been deployed in two forms. The Secure Network Server provides network security in a military environment, and is built on the "objects as memory" model. The Sidewinder[8] provides net-

6. The reader is invited to verify this claim by adding a destructive Trojan Horse to the previous example.

7. e.g., "Bug in sendmail allows unauthorized root access."

8. Sidewinder is a registered trademark of Secure Computing Corporation.

work security in a commercial environment, and is built on the "objects as files" model [40]. A UNIX-based prototype system, which implements the basic principles in a different way, is also under development [3].

The principal practical problem with type enforcement systems centers on filling in the DDT. The Secure Network Server project implemented a specialized language for the construction and analysis of DDTs. A second approach was taken in [3], in which types and domains are derived from the position of the file in the hierarchy. It is argued that this approach requires fewer modifications to the base system and is more consistent with a networked file system.

Assurance

The ability to reason *a priori* about system behavior is a major concern for achieving high levels of certification, and therefore was a major goal of PSOS and of the projects in which type enforcement was developed [16][20][21][37]. The assurance activities were integrated with the development process, with emphasis on practical assurance. This was defined by the technical leadership of the project in the following way: The assurance team and the development team should use the same mental model of the system. In other words, the modular decomposition used by the developers should be directly reflected in the structure of the argument seeking to convince people that the system is correctly specified and implemented [6].

The LOCK assurance effort focused on what it called "journal level proofs." These are intended to be at the level of published mathematical proofs, where the argument is presented at a higher level of abstraction than that required by the typical mechanical proof checker. The objective is to produce an assurance argument which can be subjected to peer group review [15].

An essential part of practical assurance is an approach called "factored assurance." Factored assurance involves constructing an assurance argument in the same form as a mathematical proof, with lemmas that taken together support the final conclusion. The truth of individual lemmas are demonstrated in a variety of ways and to different degrees of rigor. In general, there are two classes of lemmas. Lemmas of the first class demonstrate that a module takes positive steps to achieve security ("do the right thing"). Lemmas in the other class demonstrate that a module is benign, that is, refrains from performing an actively malicious act ("don't do the wrong thing").

The following example contains both classes of lemmas. Consider the problem of designing and verifying a subsystem whose duty it is to place sensitivity labels on printed output. This subsystem is organized as pipeline consisting of three elements: a data preparation module, a labelling module, and a print module, communicating through intermediate files as shown in Figure 1. Each of these modules would be executed within a separate process to enable type enforcement to work and placed in a separate domain. The relevant DDT subset is shown in Table 1. The assurance argument is then structured as follows:

Theorem: *Correct labels appear on all printed output.*

Lemma 1: *Type enforcement works as specified.* This lemma appears in assurance arguments for all subsystems, and therefore is one deserving of the most stringent assurance steps.

Lemma 2: *All data flows through the labelling module prior to being printed.* This lemma is demonstrated by examining the DDT subset given in the table; note that it is the pattern of blanks (no access) which is the most important; particularly the fact that the Print Module cannot access Unlabelled Data.

Lemma 3: *The labelling module inserts a correct label at all correct points in the printed output.* This "do the right thing" property can be demonstrated by formal or informal techniques that show correspondence between specifications and implementations.

Lemma 4: *The print module does not modify the labels.* This is a "don't do the wrong thing" property. Demonstration of this lemma, and others like it, is complicated by the fact that detailed documentation or source code for the print module may not be available to the organization doing the assurance. The assurance argument to this property may therefore involve such steps as "black box" testing and reverse engineering. Since these methods are not foolproof, there is a certain degree of risk that a sophisticated flaw or Trojan Horse may remain.

Comparison with Other Mechanisms

Two UNIX commands have effects that might be intimately related to type enforcement designs. Both setuid and chroot modify the execution context and can change the process' view of the file system. The relationship to the setuid mechanism is complex, and is treated thoroughly in [38]. The chroot mechanism limits a process to a set of files subordinate to a designated directory. While the chroot mechanism provides a crude way of associating a set of files with a process, it does not permit the enforcement of pipelines or support the kind of assurance steps described above.

Application of Type Enforcement to the Mobile Agent Confinement Problem

From the point of view of a designer of a confinement mechanism, the principal problem posed by mobile agents is the inability to predict in advance what agents will be loaded and what data it is appropriate for them to access. This uncertainty precludes static table configurations such as described in [10] and [40]. The problem is different than that addressed in [3], which describes a way in which a type enforcing client or server can access remote types and domains over a network. In that approach, software in remote domains remains on the remote machine. In the case of mobile agents, the software and its associated state moves to the client, executes, and possibly exports data to the server or another client.

The uncertainty can be resolved by the use of a public registry of domains in the client machine, and logic that is very similar to that used by a dynamic program linker to resolve module names at run time.

Domains are, in general, associated with major subsystems (e.g., mail, database, workgroup support, World Wide Web interface, etc.). For each "public" domain that is permitted to interface with mobile agents the registry could contain the following information:

- Types of objects that the domain is willing to have a mobile agent read.
- Types of objects that the domain is willing to have a mobile agent write.
- Types of objects that the domain is willing to have a mobile agent execute (invoke).

This is very similar to the list of entry points in a dynamic program load module.

Prior to downloading of the mobile agent, the server will interrogate the registry to determine if there is a domain (subsystem) of interest and, if so, how the agent should interface with it. If there are more than one type with the same access (e.g., three types of objects the mobile agent can write to) then the server needs to recognize the names of these types. Since the server is dispatching a mobile agent tailored to a specific client subsystem, this is not an onerous requirement; applications such as Web browsers can update the registry with type names or aliases as part of the installation process. The server also needs to transfer to the client the configuration of private types and domains the mobile agent requires for its safe execution.

Once this information is at the client, the type enforcement mechanism can configure its internal tables [10] or control language [3] to confine the mobile agent.

Two observations need to be made about this proposed scheme. The first is, of course, that the client system needs to implement type enforcement. The second is that the server must trust the client but the client does not need to trust the server. That is, a malicious or malfunctioning client could interfere with the operation of a mobile agent (or run something else entirely) and return misleading results.

Conclusion

The confinement problem has become more important since the advent of mobile agents roaming the Internet. The confinement problem has been extensively studied and solutions have been proposed, prototyped and deployed during the past 20 years. At least one solution, type enforcement, contains an avenue for accommodating and confining mobile agents by simple extensions of the table within its basic confinement mechanism - the domain definition table.

Despite its importance, the confinement problem has been ignored by many major operating system vendors. Now the increasing awareness of the potential for abuse and damage by mobile agents should suggest that more attention be directed toward confining processes within networked systems. Before designers and implementors attempt new, unseasoned approaches to this problem, they should become familiar with existing approaches and solutions, such as type enforcement.

REFERENCES

[1] Ames, Stanley R., Jr. *File Attributes and Their Relationship to Computer Security.* Case Western Reserve Univ. Dept. Computing and Information Sciences. NTIS AD-A002 159, 1974.

[2] Anderson, James P. *Computer Security Technology Planning Study, Vol.1.* USAF ESD-TR-73-51, 1972.

[3] Badger, Lee; Sterne, D.F.; Sherman, D.L.; Walker, K.M.; and Haghighat, S.A. "Practical Domain and Type Enforcement for UNIX." *Proc. IEEE Symp. on Security and Privacy,* 1995.

[4] Bell, David E., and La Padula, L. J. *Secure Computer Systems: Unified Exposition and Multics Interpretation,* Mitre Technical Report MTR-2997, rev 2, March 1976.

[5] Bensoussan, Andre; Clingen, C. T.; and Daley, R. C. "The Multics Virtual Memory: Concepts and Design". *Communications of the ACM* 15(5), May 1972.

[6] Berg, Helmut K.; Boebert, W. Earl; Franta, W. R.; and Moher, T. G. *Formal Methods of Program Verification and Specification.* Prentice-Hall Inc., 1982.

[7] Biba, Kenneth J. *Integrity Considerations for Secure Computer Systems.* Mitre Technical Report MTR-3153 Rev 1, USAF ESD-TR-76-372, NTIS AD-A039324, 1977.

[8] Boebert, W. Earl "Constructing an Infosec System Using LOCK Technology." *Proc. National Computer Security Conf.,* 1988.

[9] Boebert, W. Earl "The LOCK Demonstration." *Proc. National Computer Security Conf.,* 1988.

[10] Boebert, W. Earl and Kain, R. Y. "A Practical Alternative to Hierarchical Integrity Policies." *Proc. National Computer Security Conf.,*1985.

[11] Boebert, W. Earl and Ferguson, C. "A Partial Solution to the Discretionary Trojan Horse Problem." *Proc. National Computer Security Conf.,* 1985.

[12] Boebert, W. Earl; Kain, R. Y.; and Young, W. D. "Secure Computing: The Secure Ada Target Approach". *Scientific Honeyweller* 6(2), Jul. 1985.

[13] Boebert, W. Earl; Young, W. D.; Kain, R. Y.; and Hansohn, S. A. "Secure Ada Target: Issues, System Design, and Verification." *Proc. IEEE Symp. on Security and Privacy,* 1985.

[14] Delashmutt, L. F. , Jr. "Steps Toward a Provably Secure Operating System." *Proc. IEEE COMPCON,* 1979.

[15] DeMillo, Richard A.; Lipton, R. J.; and Perlis, A. J. "Social Processes and the Proofs of Theorems and Programs". *Communications of the ACM* 22(5), May 1979.

[16] Fine, Todd; Haigh, J. T.; O'Brien, R. C.; and Toups, D. L. "Noninterference and Unwinding for LOCK." *Proc. IEEE Computer Security Foundations Workshop,* 1989.

[17] Fine, Todd and Minear, S. E. "Assuring Distributed Trusted Mach." *Proc. IEEE Symp. on Security and Privacy,* 1993.

[18] Gasser, Morrie. *Building a Secure Computer System.* Van Nostrand/ Reinhold, 1988.

[19] Gosling, James and McGilton, H. *The Java Language Overview.* Sun Microsystems Technical Report, May 1995.

[20] Haigh, J. Thomas; Kemmerer, Richard A.; McHugh, J.; Young, W. D. "An Experience using Two Covert Channel Analysis Techniques on a Real System Design". *Proc. IEEE Symp. on Security and Privacy,* 1986.

[21] Haigh, J. Thomas and Young, W. D. "Extending the Non-Interference Version of MLS for SAT". *Proc. IEEE Symp. on Security and Privacy,* 1986.

[22] Haigh, J. Thomas; O'Brien, R.C.; and Thomsen, D.J. "The LDV Secure Relational Database Model", In Database Security IV., North-Holland, 1991.

[23] Harrison, Michael A.; Ruzzo, W. L.; and Ullman, J. D. "Protection in operating systems". *Communications of the ACM* 19(8), Aug 1976.

[24] Kain, Richard Y. and Landwehr, C.L. "On Access Checking in Capability-Based Systems". *Proc IEEE Symp. on Security and Privacy,* 1986.

[25] Kain, Richard Y. *Advanced Computer Architecture: A Systems Design Approach,* Prentice-Hall, 1996.

[26] Lampson, Butler W. "Protection". Proc. 5th Princeton Symp. on Information Sciences and Systems, 1971. Reprinted in *ACM SIGOPS Operating Systems Review* 8(1), Jan. 1974.

[27] Lee, Theodore M. P. "Using Mandatory Integrity to Enforce 'Commercial' Security". *Proc. IEEE Symp. on Security and Privacy,* 1988.

[28] McHugh, John. "An EMACS-Based Downgrader for the SAT". *Proc. National Computer Security Conf.,* 1985.

[29] O'Brien, Richard and Rogers, C. "Developing Applications on LOCK". *Proc. National Computer Security Conf.,* 1991.

[30] Ritchie, Dennis M. and Thompson, K. "The UNIX time-sharing system." *Bell System Technical Journal.* 57(6), Jul. 1978.

[31] Saltzer, Jerome H. and Schroeder, M. D. "The protection of information in computer systems". *Proceedings of the IEEE* 63(9), Sep. 1975.

[32] Saydjari, O. Sami; Beckman, J. M.; and Leaman, J. R. "LOCK Trek: Navigating Uncharted Space". *Proc. IEEE Symp. on Security and Privacy,* 1989.

[33] Saydjari, O. Sami.; Beckman, J. M.; and Leaman, J. R. (NCSC). "Locking Computers Securely." *Proc. National Computer Seucrity Conf.,* 1987.

[34] Schaffer, Mark and Walsh, G. "LOCK/ix: On Implementing UNIX on the LOCK TCB". *Proc. National Computer Security Conf.,* 1988.

[35] Shirley, L. J.; Schell, R. R. "Mechanism Sufficiency Validation by Assignment." *Proc. IEEE Symp. on Security and Privacy,* 1981.

[36] Schroeder, Michael D.; Clark, David D.; Saltzer, Jerome H. "The Multics Kernel Design Project". *ACM SIGOPS Operating Systems Review* 11(5), Nov. 1977.

[37] Taylor, Tad. "FTLS-Based Security Testing for LOCK." *Proc. National Computer Security Conf.,* 1989.

[38] Thomsen, Daniel J. and Haigh, D.T. "A Comparison of Type Enforcement and UNIX Setuid Implementation of Well Formed Transactions." *Proc. IEEE Computer Security Applications Conf.,* 1990.

[39] Thomsen, Daniel J., "Role-Based Application Design and Enforcement." *Proc. IFIP Working Group 11.3 in Database Security,* 1990.

[40] Thomsen, Daniel J. "Sidewinder: Combining Type Enforcement and UNIX". *Proc. IEEE Computer Security Applications Conf.,* 1995.

[41] Weissman, C. "Security Controls in the ADEPT-50 Time-Sharing System." *Proc. AFIPS Fall Joint Computer Conf.,* 1969.

[42] Wong, Raymond M. "A Comparison of Secure UNIX Operating Systems". *Proc. IEEE Computer Security Applications Conf.,* 1990.

[43] Young, Wiliam. D.; Boebert, W. E.; and Kain, R. Y. "Proving a Computer System Secure". *Scientific Honeyweller* 6(2), Jul. 1985.

[44] Young, William. D.; Telega, P. A.; and Boebert, W. E. "A Verified Labeller for the Secure ADA Target". *Proc. National Computer Security Conf.,* 1986.

FIGURE 1

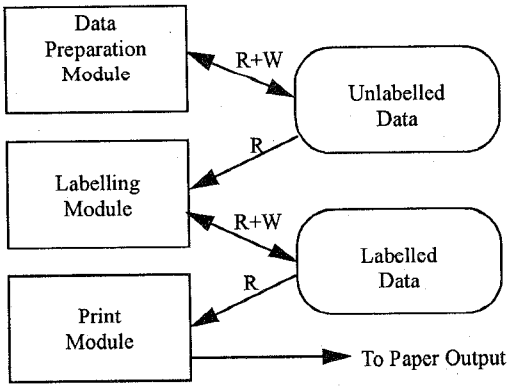

TABLE 1

Domain →

Type ↓

	Data Prep	Labelling	Print
Unlabelled	R+W	R	
Labelled		R+W	R