LOCK : An Historical Perspective

O. Sami Saydjari {ssaydjari@CyberDefenseAgency.com}
Cyber Defense Agency, LLC

Introduction Prescript

It has been nearly three decades since the National Computer Security Center's Logical Coprocessing Kernel (LOCK) program originated in a program called the Provably Secure Operating Systems. LOCK was the last of the highly trustworthy operating system research projects. The original paper, on which this is based [22], captures the essence of the goals and accomplishments of the program. I have not updated the verb tense in the paper to keep its original form when published in 1987. At the end of the paper, I added a commentary on the outcomes and lessons learned.

The design principles described remain highly relevant today and serve as one of the most valuable contributions from the LOCK program for successor projects. Important concepts from the LOCK program which have stood the test of time are: the notion of separating policy decision making from policy enforcement (which later became known as policy-neutral design), and the general power of type enforcement mechanism for implementing a large variety of practical security policies.

Abstract

LOCK is an advanced development of hardware-based computer security and crypto-graphic service modules. Much of the design and some of the implementation specifications are complete. The Formal Top Level Specification (FTLS) also is complete and the advanced noninterference proofs are beginning. This hardware-based approach has brought the LOCK project into many uncharted areas in the design, verification, and evaluation of an integrated information security system. System integration promises to be the single largest programmatic problem. Our verification tools seem able to verify design only and not implementation.

Introduction

The purpose of this paper is to capture the essence of the design principles guiding the LOCK project. This paper provides an historical perspective with some projections to the future. If the reader is unfamiliar with LOCK, we suggest reading our previous overview [1].

LOCK is an advanced development of two reusable hardware security modules for general purpose computers. The System-Independent Domain-Enforcing Assured Reference Monitor (SIDEARM) separates different levels of classified information.[1] The Bulk Encryption Device (BED) separates unencrypted data processed by the computer from encrypted data stored on disk and tape.[2] The integration of these modules into a computer provides high-grade INFOrmation SECurity (INFOSEC). We call such a system a "LOCKed" computer.

The NCSC sponsors LOCK to subsidize industry risk in the development of standardized reusable INFOSEC modules. We encourage vendor production of high-grade secure systems by encapsulating much of it in the LOCK standard modules (SIDEARM and BED). We expect these modules to reduce system development time and cost. LOCK provides the open architecture platform on which many secure systems may be developed [2].

The LOCK program has a 17-year history. The project grew out of the Provably Secure Operating System (PSOS) study [3] begun in 1973 at the Stanford Research Institute (SRI), An implementation study by Ford Aerospace [4] began in 1980. The study concluded that implementation was not feasible with available resources. The NCSC continued the development of a limited-functionality hardware-based Trusted Computing Base (TCB) with Honeywell. This project, called the Secure Ada Target (SAT), marked the beginning of the current three-phase development described below.

The SAT project, begun in 1982, was a research effort to design secure computers. Phase one yielded the high-level requirements specification in 1983. Phase two yielded the functional design specification in 1986. Phase three, now called LOCK, will yield a detailed design specification and a secure microcomputer prototype by 1990.

The theme of this paper is INFOSEC design principles and their application to LOCK. Section 2 briefly describes each of the major principles. Section 3 describes the application of these principles in major design decisions. Sections 4-5 focus on two important design areas: design verification and porting Unix[3] System V to a LOCK host. Section 6 projects future LOCK directions.

[1] This feature is also called red-red Separation> multi-level security, and is part of COMPUter SECurity (COMPUSEC).

[2] This feature is also called red-black separation and is part of COMmunications SECurity (COMSEC),

[3] Unix is a registered trademark of AT&T.

LOCK design principles

This section lists the design principles on which LOCK is based. All other sections focus on how we applied these principles. The principles are:

(1) Small-and-Real Beats Big-and-Theoretical,
(2) Incremental Development,
(3) Focus on the Unknown,
(4) Minimize Security Impact on Users
(5) Maintain Performance with Hardware,
(6) Theories of Security Come from Theories of Insecurity,
(7) Enemy Inside the System,
(8) Enhance Security Features, and
(9) Least Privilege.

Small-and-real beats big-and-theoretical

The PSOS design was large and complex. To build a real system, one had to focus on a subset of the problems. Further, design discussions had proceeded sufficiently that developing a prototype secure system was the next logical step.

SAT began as a development of a small secure platform on which further work could be built. It was to be a target machine (execution environment only) with only a minimal executive for dedicated Ada applications (hence the name Secure Ada Target). We call this design a TCB builder's tool because it provides a secure base and set of primitive security operations, but it is not functionally complete.

Incremental development

An important lesson learned from the PSOS implementation study was the importance of incremental development. The idea is to develop the system base and then slowly add well-defined modules, In verification, this principle translates to "build-a-little, prove-a-little" (see Verification Approach) and in design, to "integrate-a-little, test-a-little."

Focus on the unknown: prototype versus product

LOCK is one or two stages away from final product development. Because of limited resources, we focus on aspects of the design we least understand. For example, we defer full functional testing until the final product. Instead, we focus on new problems, such as the requirements of software drivers for the cryptographic subsystem. One corollary of this principle is to use commodity (off-the-shelf) rather than customized parts. This frees us from doing more design work than necessary.

Minimize security impact on users

Users want to use computers to meet their operational goal—whether it be processing bank transactions or managing battles. Users want security, but not at the expense of the primary mission. Security features should be as transparent as possible to the user. This means secure systems must minimize (1) change to the user interface and (2) system performance degradation.

LOCK's portability provides a common solution for many different architectures. The LOCK modules will support different operating systems with minimal change to the application and user interfaces.

Maintain performance with hardware

Security and performance goals cannot be optimized simultaneously. Security enforcement requires System resources. LOCK's dedicated security-enforcing hardware reduces the performance impact on the host computer. The speedup over a software security kernel is analogous to using a math coprocessor compared to software floating point emulation.

Processors and memory are relatively inexpensive and the cost trend is down. Both the SIDEARM and the BED contain at least two microprocessors. Whenever a security feature may degrade performance, we first try to support it with hardware. If that is not possible, we must trade performance for our primary goal of security. Despite this, we believe that a LOCKed computer will lose less than 10% performance compared to an unLOCKed version.

Security has top priority in the prototype. To cut cost and increase performance, we can always back off and create less secure (e.g. B1) versions. We could move SIDEARM functions into host software, or delete or combine module functions.

Theories of security come from theories of insecurity

LOCK focuses on security that counters specific threats. We believe that theories of security come from theories of insecurity. For example, we must do extensive formal analysis from at least two different perspectives to find covert channels [5].

Enemy inside the system

A designer of a secure system must assume that his enemy knows the details of his system. The designer must further assume that the enemy is inside the system - that all untrusted software is subverted. The TCB must ensure that untrusted software cannot (1) violate security policy and (2) spoof users performing security-relevant operations [6].

Enhance security features

In computer security, users need to control access to their information on a per-user basis in addition to per-level control [7]. There are many ways to provide this control. One way is to leave per-user control entirely to the discretion of the owner (usually the creator) of an object. This is called Discretionary Access Control (DAC). Once a user can read an object he can copy the object's contents and give access to his COPY. This weakness of DAC allows Discretionary Trojan Horses [8] to operate within a given Mandatory Access Control (MAC) level.
We sought a stronger mechanism that would give the same per-user control.

In addition to traditional access control lists, LOCK deals with per-user access control by allowing extensions to the security lattice. These extensions allow for increased granularity control within the stronger MAC scheme. LOCK's dynamic lattice allows special users to add, delete, and reconfigure parts of the dominance lattice with a trusted program. This user-administration of lattice "branches" models the paper world in which control of compartments often is decentralized. The flexible lattice approach also allows minimal representation, a Partially Ordered SET (POSET), that excludes disallowed combinations of security levels [9].

Least privilege

The principle of least privilege means restricting system privileges to the absolute minimum required by a program to do its job. In addition to MAC, LOCK refines least privilege with Type Enforcement. This mechanism sets up tight domains of privilege in which specific programs must run. Each object has a type in addition to a security level. Access to specified types are restricted to specified domains [10, 1].

The partitioning of host-resident TCB software into many domains may degrade performance. If the time needed to switch security contexts in going from one domain to another is excessive, domains can be combined.

Major desire trade-off decisions

After an introduction to LOCK architecture, major project design decisions are discussed. Chronologically ordered to show the design progress, we show how we have applied the design principles discussed given our prioritized goals of (1) security, (2) performance, and (3) cost. The decisions are listed below and detailed in subsections:

(1) Secure Existing Computers,
(2) Capability-Based Access Control,
(3) Security Policy Extendibility,
(4) Loose versus Tight Coupling,
(5) Commodity versus Customized Components.

LOCK architecture requirements

To use the LOCK standard modules, computers must have (1) a two-state processor (e.g. user and master mode), and (2) one level of indirection in addressing - virtual memory. One allows the separa-tion of trusted from untrusted code. Two ensures all memory addressing will be mediated—usually using a Memory Management Unit (MMU).

The reference monitor is conceptually split into an access decider and an access enforcer (figure 1). In the LOCK implementation, the SIDEARM is the decider and the MMU is the enforcer.

We chose the architecture shown in figure 2 because it is simplest from an engineering standpoint. The MMU need not be drastically modified since the CPU is acting as the MMU table initialization mechanism. This is important in architectures where the MMU is tightly integrated into the CPU. On the other hand, this requires trust in the master-mode host CPU software to translate the reference monitor's access decision to a MMU table entry.

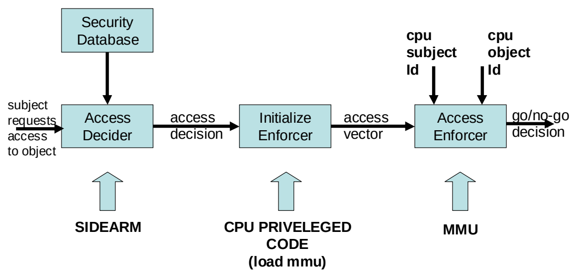

Figure 1 Reference Monitor

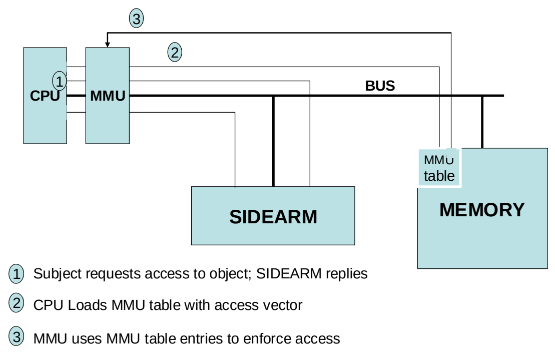
3
2
1
CPU
MMU
BUS
MMU table
SIDEARM
MEMORY
1 Subject requests access to object; SIDEARM replies
2 CPU Loads MMU table with access vector
3 MMU uses MMU table entries to enforce access

Figure 2 LOCK ACCESS CONTROL FLOW

More tightly integrated options are shown in figure 3. Sequence <1, 2b> shows the more tightly coupled option where the SIDEARM loads its decision directly into MMU-internal tables. This may require a large MMU-local memory depending on the number of active users and virtual memory scheme. The MMU also must be separate so that it can be customized. Vendors would have to change newer CPU-coupled MMU designs to accommodate this approach. The other sequence, <1, 2a, 3a>, requires less change since it loads MMU tables over the global bus into primary memory.

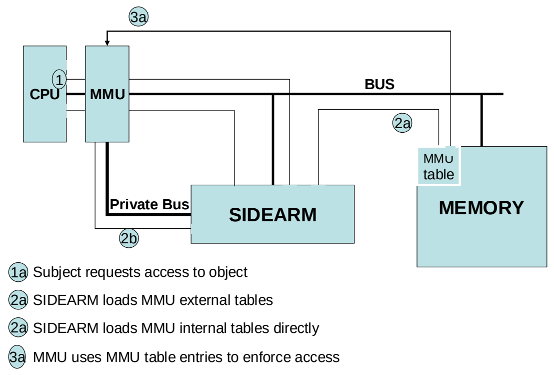
3a
1
CPU
MMU
BUS
2a
MMU table
Private Bus
SIDEARM
MEMORY
2b
1a Subject requests access to object
2a SIDEARM loads MMU external tables
2a SIDEARM loads MMU internal tables directly
3a MMU uses MMU table entries to enforce access

Figure 3 : ALTERNATE ACCESS CONTROL FLOW

Secure existing computers

A primary driver in many design decisions is the goal to retrofit the security into existing architectures. This led us to reusable hardware modules to extend the computers' effective Instruction Set Architecture to include security-enforcing operations.

There are two main reasons to retrofit: (1) reduce development cost and (2) extend applicability to fielded computers. We reduce development cost by avoiding the large investment required for a completely new computer design. We extend applicability with reusable modules. These modules allow us to introduce high-grade information security into systems without requiring major changes in procurement directions. We minimize impact on users' substantial investment in both hardware and software systems.

Capability-based access control

This section traces the historical progress of access control capabilities in the LOCK design. LOCK began as a pure capability machine (PSOS). A conceptual flaw with this approach [11] led to touch-limited capabilities. That approach then gave way to name capabilities (LOCK) in which access rights are computed each time the subject loads the capability. We recommend reviewing Kain's capability machine taxonomy [12] to fully appreciate this section.

Definition - A capability is essentially a ticket by which a subject gains access to an object; it usually has at least two fields: name and access rights. The name is a Unique IDentifer (UID) of the object. The access rights field contains the privileges the possessing subject has to that object.

Capabilities must have integrity; the UID field must be opaque to the user; and its propagation must be controlled. Integrity prevents a user from making his own capabilities thereby giving himself access to any system object [13]. Opaqueness prevents a user from seeing a predictable sequence of UIDS which he can then use to drive a covert channel, Controlled propagation prevents security policy violations [12].

Pure Capabilities - LOCK began as an implementation of a large and complex design of a distributed reference monitor resulting from SRI's PSOS Study. A reference monitor is distributed when the access decider is remotely separate from the access enforcer and the cost of communicating with the access decider is high. The access decision is stored in the capability to avoid asking the access decider each time a subject uses it,

In a pure capability approach, a subject with read access to an object in which a capability is stored may freely copy it. Unfortunately, this freedom leads to the violation of the *-property [11]. Capability propagation had to be controlled to solve this problem. Prohibiting capability propagation was not acceptable because it prevents information sharing entirely. Touch-limits were developed to control capability propagation.

Touch-Limited Capabilities - Touch-limiting adds security attribute information to a pure capability describing an object [14]. For example, an object security-level and type field may be added to the UID and rights field. The reference monitor compares the attributes of the object to those of the copying subject to restrict the access rights field when the capability is copied. For example, when a high-level subject copies a read-write capability referring to a low-level object, the reference monitor deletes write access in the copied capability to preserve the *-property.

This approach requires special atomic operations to create and manipulate capabilities. These operations increase the size and complexity of the reference monitor. This increased complexity is necessary to use capabilities with stored access rights.

Name Capabilities - The decision to continue storing access decisions in capabilities was based on our desire to keep the design general. Stored access decisions permit the possibility of a remote access decision maker. This goal justified bearing the complexity of capability operations.

During SAT-I, the evaluators required discretionary access revocation to be effective by the next Iogin. Because the discretionary access is part of the access decision stored in the capability, all effected capabilities had to be tracked down and modified. The performance overhead was too high. This was the final blow to the use of stored decisions.

We deleted the capability's access rights field leaving only the UID field. The capability is then simply a name. Access rights are computed when objects are first referenced in a session by a subject. The computation is based on the current security attributes of the subject and object and the system security policy.

The decision to defer access rights computation restricted the class of architectures to which the current LOCK design could be applied. On the other hand, touch-limiting is still a viable option.

Security policy extendibility

Supervisor code on the host is limited to that which must provide low-level security-related functionality. This consists primarily of physical resource management functions such as process multiplexing. The host-resident TCB code may be thought of as the driver software for the SIDEARM.

The operating system integration will consist mainly of moving security-related functionality from the operating system to the TCB. This will be discussed in section 5.

Kernel Extensions (KEs) extend the security policy implemented by the SIDEARM. This extension is necessary for two reasons. Anticipating security requirements for future applications is impossible; the requirements themselves are potentially contradictory. Additionally, including all known, non-contradictory security requirements would create a cost and performance burden. KEs are verified to have some property that refines the basic security policy. They execute "on top of" the supervisor code on the host under the control of SIDEARM [15,16] (figure 4). This method of extending TCB operations allows incremental advances to LOCK functionality from different software vendors [2].

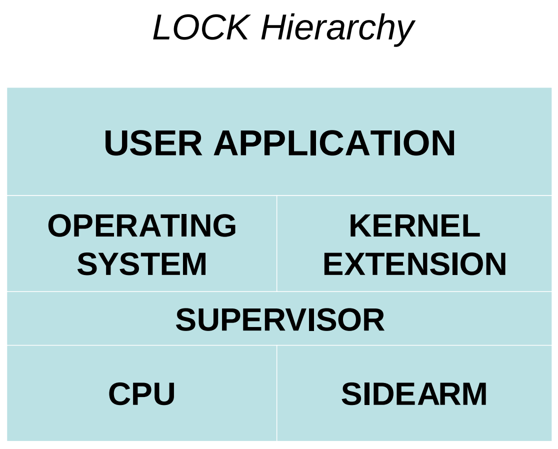

Figure 4 Secure Architectures

Loose versus tight coupling

The integration of LOCK modules can range from their use as peripherals to their placement on the CPU board. For the prototype we favored loose coupling. Both the SIDEARM and BED are separate devices. They are integrated into the system bus like disk controllers. The system integrator writes a SIDEARM and BED "device driver" to allow the CPU to talk to them like it talks to device controllers (disks, tapes, etc.).

Tight coupling can be done by incorporating the module device drivers directly into the host processor's microcode. One could even create a customized CPU board with both host processor and SIDEARM. Tight coupling follows the trend toward packing increasing functionality into processors.

The loose coupling approach allows prototyping flexibility. It is also easier to move from the loosely coupled to the tightly coupled approach.

Commodity versus customized components

Because of our principle to focus on the unknown, the LOCK prototype uses many commodity components. The SIDEARM module is made entirely of commodity components including: microprocessors and memory. Smaller and higher performance customized moduIes may be developed during the manufacturing stage.

Custom development is localized to three moduIes: the Host Interface Controller (HIC), the SIDEARM Interface Controller (SIC), and the MMU. The LOCK MMU is a customization of the host computer's native MMU.

Verification approach

LOCK is a significant example of a secure system development using verification technology. This section gives an overview of our goals and a flavor of some of the problems in achieving them.

Proof philosophy

A security proof of a specification is a convincing argument to increase one's confidence that a system design does not violate a security policy. Proof of security is not running a program specification through an automatic verifier. A real proof is a human-understandable argument that a design specification adheres to our intuitions of security. It must withstand scrutiny by human reviewers [17].

Humans must formalize the intuition of a security policy in statements that must be proved about specifications. Humans must judge whether the specification completely characterizes the system design. Humans must determine whether the proof argument is sound.

Security proofs, like mathematical theorem proofs, are not incontrovertible. Proofs increase confidence in correctness; they do not provide absolute certainty. Security proofs are inherently negative statements about access rules violations; they do not determine whether the design is free from unwanted functionality (e.g. trojan horses and viruses) that violates our intuitive sense of security.

Social review

Because we believe that proof arguments must be understandable, LOCK starts with "journal-level proofs." These are proof arguments that sketch the proof out in much the same way mathematicians sketch their proofs. The proofs can be refereed by mathematicians in the same way that proofs in mathematics journals are refereed. We have established a group of volunteer referees who will scrutinize our proofs: the LOCK Proof Referees.

Gypsy: A proof checker

Verification tools are in their infancy. None are production quality. Brute force application of the tools to specifications can be hazardous to the security health of the system designed. Questions remain on fundamental issues such as the soundness of the tools. Incorrect proofs of security show up because of subtle errors in assumptions on the part of the person using the tool. Tools like the Gypsy Verification Environment (GVE) are evolving to address these concerns. Unfortunately, this evolution makes operational use even more difficult.

We chose the GVE as the most mature and flexible among NCSC-endorsed tools. The GVE proves specifications semi-automatically with human guidance. The journal-level proofs establish the human proof strategy; GVE provides a machine check.

The GVE tool has caught several non-obvious design errors. For example, valid-state conditions were listed as unproven hypotheses while proving the specification. These conditions are mainly consistency requirements on the security attributes data base. The tool also caught a covert channel in a SIDEARM operation while trying to prove noninterference conditions (see Conditional Noninterference).

About $1.7 million is being spent on LOCK program verification -- about 9% of the total project cost. Much of the money is spent in the Gypsy proof checking process. Whether it is worth checking the journal-level proofs with the time-intensive tools is still an open question.

Pushing toward verified implementations

A verified design is only as good as the informal arguments mapping the design to the implementation. High-grade secure systems must push assurance closer to verified functionally correct implementations. Our current FTLS already are very detailed and our Kernel Extension (KE) proofs will be even more so.

Unfortunately, the increased size and detail of the formal specification is pushing the GVE past its capabilities. Limits on the specification language's expressiveness, the efficiency of the proof generation process, unsoundness, and a lack of power in the underlying theorem p-rover all characterize this difficulty. Despite this, we still believe that the GVE is the best choice for LOCK. These problems show the need for advances in verification tools and are not meant to indict any one tool.

We are providing feedback for Gypsy impr
ovements to the Gypsy developers based on our experience. Resource limitations caused us to scale back planned Formal Implementation Level Proofs (FILS). An FILS would be almost 1-1 mappable to source code. We plan to experiment in proving some parts of the FILS. We then will assess what properties tools must have to prove an FILS.

Proof optimization: layering and factoring

We have applied two techniques for increasing the efficiency of proof checking with the GVE: proof layering and proof factoring.

Proof layering structures the proof of one part of the specification to make use of the proof of another. For example, the proof of Type Enforcement is at the base layer; KE proofs are conditioned on this base proof. Type Enforcement guarantees that only certain programs will write objects of a given file type. Programs reading this file type guarantee that properties of the written files are maintained. KE correctness is simpler to prove when the input is provably constrained by Type Enforcement.

Proof factoring involves carefully choosing the parts of a proof to separate out as lemmas and generalizing them for subsequent re-use. Proof factoring is analogous to the concept of reusable software modules. It requires insight and experience which can be gained only by doing some proofs without factoring. One then sees some commonality, factors the proof, and generalizes.

Conditional noninterference

LOCK advanced system verification by integrating covert channel analysis into the proof analysis. We have done this with a newly applied technique called conditional noninterference [18]. Conditional nonin-terference is represented by several new theorems that must be proved about the specification. When the proof fails, you have identified a covert channel. Since most practical systems cannot function without some covert channels [6], it is not always possible to correct the design so that the proof succeeds. Sometimes one can only reduce the channel bandwidth and then audit it.

LOCK/ix

Although LOCK started as a target machine, we soon progressed to a self-hosting requirement. This decision brought with it many other considerations, primarily the choice of operating system.

We considered two options: porting an existing operating system on top of the LOCK TCB or developing a new operating system. We chose the port to demonstrate the generic nature of LOCK, i.e. it can work with any commercial operating system. This decision also saves us the time and expense of developing a new operating system and applications. We chose Unix as the first operating system to be ported to the LOCK TCB for three reasons: (1) Unix is relatively small. The kernel has about 17,000 lines of code; (2) Unix is widely used; and, (3) The Unix application base is large.

LOCK/ix Approach

The unmodified Unix operating system has a reputation for being insecure [19,20]. This is not an issue for LOCK because most of the operating system is treated as hostile code. Much of the operating system can be regarded as another application running on top of the LOCK TCB. This means that the operating system can be modified without affecting the security of the system.

Like any other operating system, Unix has security-related functions rooted in resource management. The SIDEARM and low-level, host-resident TCB code provides this functionality. Therefore some operating system internaIs must be removed or modified in porting it to LOCK. We expect these changes to be minimal and localized, however, to simplify the porting of applications.

LOCK/ix will provide “virtual machine-type” multi-level security. During any particular login session, a LOCK/ix subject can operate only at the security level granted at login. Users must log out and log back in to change a security level. A login session is equivalent to a Unix process family (figure 5). We chose this configuration for three reasons: (1) better performance; (2) minimization of necessary trusted code; and, (3) simplicity due to the relatively small number of Unix kernel modifications required [21].

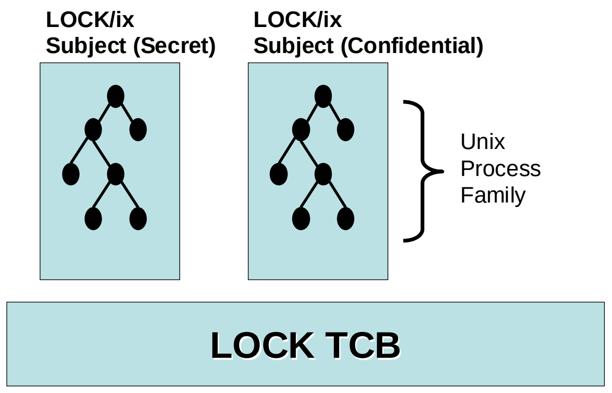

Figure 5 LOCK/ix Subjects and Processes

Compatibility goals summary

The LOCK/ix interface definition is based on Unix System V Release 1. This definition sets out the differences between LOCK/ix and Unix System V Release 1 and the reasons for them. These differences are a problem if they affect the porting of many applications.

Some system calls and functions must be changed to implement LOCK/ix securely. They fall into the three basic categories listed below and in figure 6.

I. The system call or function does not make sense in the LOCK implementation of Unix, and will be implemented for compatibility's sake as a no-op.

II. The system call or function requires changes to make it consistent with the LOCK environment.

III. The system call or function requires changes to implement security policy.

I. (no-op)	II. (env.)	III. (security policy)	
mount(2)	malloc(3)	access(2)	link(2)
stime(2)	sbrk(2)	chdir(2)	mknod(2)
sync(2)	times(2)	chmod(2)	mount(2)
umount(2)		chown(2)	open(2)
		creat(2)	read(2)
		exec(2)	setgid(2)
		fcntl(3)	wetuid(2)
		fopen(3)	stat(2)
		fread(3)	ustat(2)
		fwrite(3)	utime(2)
		ioctl(2)	write(2)
		kill(2)	

Figure 6 LOCK/ix Changes

The group I system calls that will be modified are those that are used for specific operations: to mount and unmount file systems, to update file systems, and to set the system date and time. File system operations are handled in LOCK/ix by a special File Server Subject. This server will not mount file systems; all portions of the file system hierarchy will be "mounted" at system generation time, in a manner which should satisfy all applications, Because the file systems are under the control of the TCB, a user will have access only to those file systems allowed by the security policy. The system time in a LOCK system is set outside of LOCK/ix, (by TCB code during bootup).

The group II system calls and functions designated for modification are those concerned with memory allocation and time measurement. A side effect of the LOCK addressing scheme limits the maximum object size to 16 megabytes. The Dual computer that we are LOCKing has a 32-bit address space and LOCK uses the uppermost eight bits for the offset into the subject translation table. The memory allocation system calls and functions must be changed to reflect this limitation. The LOCK TCB reports system clock time to the nearest second (versus 1/100 of a second in standard Unix) to non-TCB applications to prevent its use in covert channels. This solution is acceptable for non-real time systems.

The group III system calls and functions requiring modification address a wide range of uses. We have made a few assumptions when discussing these. First, the addition of security functionality does not in any way change what the system call or function does. If the user has the required access rights the call will work as expected.

Also, there are some changes that are a result of securing the system (for instance, user-id 0 will no longer grant "super-user" privileges). These modifications will require changes in some programs, but the use of the system calls themselves will not affect the porting of most applications.

LOCK/ix and setuid/setgid

One of the issues involving some of the group III system calls and functions is the use of the setuid (setgid) function. In standard Unix System V, there is the concept of a real user-id and group-id as well as an effective user-id and group-id. The setuid (setgid) feature is used to change the effective user-id (group-id) from the real user-id (group-id) to some other value. This feature provides a temporary way to change one's effective user-id (group-id) to accomplish a task, and then resets it to the real user-id (group-id) value. The LOCK security policy requires that a subject retain its identity throughout its lifetime.

The File System Server (FSS) supports setuid (setgid). If the user invokes a setuid program and its effective user-id is on the Access Control List (ACL), the FSS temporarily adds the user's real user-id to the ACL to allow access to the appropriate resources. When processing is complete, the FSS calls the TCB to have the previously created entry cleared.

We believe that the result will be a functional setuid (setgid) feature that will preserve compatibility in use of the calls. Given this scheme, there are only a few system calls within LOCK/ix which can be said to be incompatible, and their frequency of use is so low that we expect only a minor impact on the portability of System V Release l-compliant applications.

Relationship to other efforts

The Portable Operating System for Computer Environments (POSIX), and the Trusted Unix Working Group (Trusix) are two of the most visible groups trying to add security to Unix standards.

POSIX is the name for a standard operating system interface defined in the IEEE 1003.1 Standard. The interface standard is a subset of the features and functionality provided in Unix System V Release 3. POSIX is also the name given to the group that evaluates and accepts or rejects proposed changes to the standard. To do this, several subcommittees have spun off to handle the specialized areas of interest. One of the more recent additions has been the P1OO3.6 subcommittee on security. Their main task is to define security extensions to the POSIX standard targeted for the B3 level of security.

Trusix is a group working in the area of secure Unix and is closely tied to the POSIX effort. Trusix is developing a TCB interface specification, and companion documents, which give a rationale for the specification while addressing central issues in the design of B3 secure Unix systems. The body of this work will be evaluated for POSIX compliance. Trusix is not a standard, but the results of this effort are intended to be used as guidelines for the development of secure, POSIX-conforming, Unix-based systems.

Figure 7a shows the relationship among these three endeavors. POSIX is based on Unix System V Release 3, while LOCK/ix is based upon Release 1. LOCK/ix will not define a secure Unix standard, but it may be used as a guideline for porting future LOCK operating systems. LOCK /ix will be an example of a secure Unix implementation which will be as compatible as possible with the Unix System V Interface Definition for Release 1.

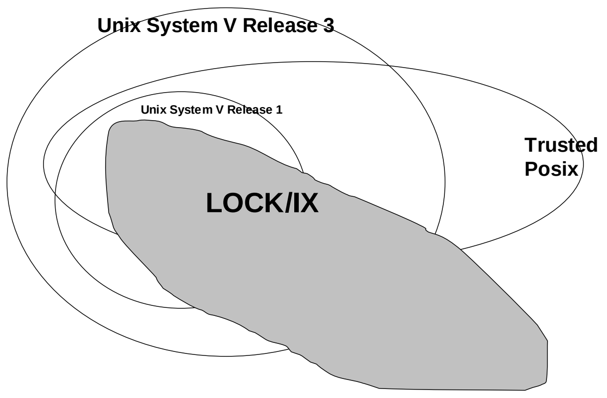

Figure 7 LOCK/ix and Secure Unix Standards

LOCK/ix does not support Inter-Subject Communication (ISC), for two reasons: (1) ISC does not exist in Unix System V Release 1, and (2) there is little need for ISC in LOCK/ix, because of the aggregation of Unix processes into one LOCK/ix subject as shown in figure 5.

The LOCK/ix solution for ACL's differs from that of Trusix. Some security features may differ due to the differences in the Unix releases upon which these two efforts are based. Other solutions may differ because these are concurrent efforts (figure 7b). LOCK/ix team participation in the Trusix group facilitates information sharing to reduce redundant work and to come to a common solution where possible.

In summary, LOCK/ix will be highly compatible with Unix System V Release 1. Changes will be required for programs that try to access or use resources that have been modified or eliminated to implement a consistent security policy. Where access is allowed, however, the strict use of system calls and functions will work as expected.

Future LOCK

We expect LOCK standard modules to help the development and evaluation communities. The use of these modules should decrease the vendors investment for secure system development. We expect to reduce both the development and evaluation time because the LOCK standard modules already will meet many of the security requirements. The modules should reduce risk and the resource expenditures for both communities.

No matter how technically strong, LOCK's success depends on industry's acceptance and use of the LOCK standard modules. We seek vendor participation in establishing standard interface control specifications for the LOCK modules. We will be asking for feedback on these documents through the LOCK Integration Needs Consortium group established for the purpose.

With industry's ownership, we expect to have several vendors develop and produce SIDEARM and BED modules. NCSC would then endorse them for sale to system integrators and would expect computer vendors to integrate the modules and conduct an accelerated evaluation. The resulting LOCKed versions of their systems would become part of their standard product line.

Module interface standardization and accelerated evaluation should allow the modules and the host systems to keep pace with advancing technology. This should reduce the cost of information security and allow integration into next-generation computer architectures.

LOCK's open architecture foundation will be common across different computers. We expect LOCK to increase the number of secure system developers and secure systems.

Postscript

Tying up loose ends.

The LOCK program created the planned prototypes. Ultimately, the notion of a hardware reference monitor was dropped because of both cost and the trend toward tightly integrating memory management unit functions into the central processing unit.

From a business perspective, LOCK modules never caught on. This was partially because of the tight integration trends cited above. Further, it became increasingly apparent that the evaluation community was unable to assess modules like SIDEARM and BED in such a way to make any guarantees regarding reduction in cost or time to complete evaluation. The cost, time, and uncertainty of evaluations were a general problem for all evaluated systems. It particularly hurt higher-assurance systems because of the increased requirements and the lack of experience base with such systems.

The social review process mentioned in the paper did take place, but the outcome was less than what was hoped for. We gave the referees a short fixed time to perform the review (about 2 weeks). The volume of the material to review was large and the review task was not structured in a way that allowed progress in the short allotted time. The panel spent much of the time debating and discussing the definitions of security and the general specification and proof approach. Little refereeing of the proof validity was accomplished. We learned that referees must be involved from the beginning and that the referee task needs to be structured to keep focus on the important aspects of the proofs that would benefit most from human review – such as the higher-level theorems and sample lower-level proofs supporting those theorems.

After delivery of the LOCK prototype, a follow-on project called Secure Network Server was initiated. SNS was a prototype effort to employ the general-purpose trusted computing base (TCB) to create a trusted guard. This application was within the larger Multilevel Information System Security Initiative addressing the Defense Message System requirement for trustworthy organizational messaging service.

The SNS ultimately reduced its target assurance level to B3. This placed it in competition with B1 and B2 commercial guards. Customers seemed to place more importance on functionality than assurance. Further, the functionality for which the SNS was used, involving guard functions, were highly complex and could not in principle achieve very strong properties. Because the more commercial B1 and B2 guards seemed more cost-effective, the SNS project was discontinued.

Concepts and mechanisms from the LOCK program went on to influence many subsequent designs. The type enforcement concept influenced several research operating systems and became an important feature of a successful commercial product called the Sidewinder firewall.

There were several lessons learned from the LOCK program.

Application security matters.

The Orange Book seemed to be based on the premise that application-level security does not matter. Well, it does. News flash: users do not actually even want operating systems; they tolerate operating systems because it enables their applications. It is through applications that organizations, military or otherwise, achieve their goals. If security policy[4] of the organization is subverted in the application, then it does not really matter if the operating system is highly assured in its own internal properties. This is not to say that operating system security does not matter and is not essential in supporting application-level security. Rather, I mean that application-level security, which ultimately requires some functional correctness, is at least as important. It is also worth pointing out that recent trends have swung the pendulum to the other extreme. Vendors are producing security features at the application level without the requisite supporting security in the operating system. This approach can lead users to a false sense of security.

Business case assumption validation is important.

The creation of the SIDEARM and BED modules had many interesting technical side-effects and research results, including the concept of policy-neutrality. At the same time, the business case was predicated on an assumption about the evaluation process that ended up being invalid, despite the strategic desire of the owning organization to make those assumptions come true. The technology and culture of evaluations simply did not support the notion of reduced evaluation time and cost by using pre-evaluated modules.

Covert channel control is hard.

If a system shares resources, it will have covert channels. It was extraordinarily difficult to find all of them because the system models do not reach down to the implementation detail where much resource sharing took place in subtle ways. Further, once found, closing channels was difficult to do without destroying the resource sharing that was an inherently desirable property of the design (e.g., multilevel security to handle the range of data an organization needed to process). The theory and limits of balancing sharing and covert channels has been inadequately developed to make good top-down design decisions on the trade-off.

[4] By this, I mean the full rich desired security policy, not the shadow that they specify because security functionality is so limited.

Return on investment in security engineering is often poorly understood and highly uneven.

Unfortunately, the Orange Book required the use of design process, tools and techniques that had really never been proven in terms of their value to the assurance or function of the system. Looking back, it is not clear that that was a particularly good idea.

We found that many tools and techniques added much value for little cost and that some added almost no value at high cost. For example, we found that inexpensive human sketches of proof about system properties identified a high number of potential design problems that were found and corrected early. Running those proofs in the gruesome detail needed to have the verification tools check those proofs cost substantially more and yielded few additional potential issues. At the same time, the mere obligation to be able to run the proofs created a strong incentive for thorough specifications and a careful journal level proof analysis. This effect had significant pay-off in system assurance.

We even found that some mechanisms, like the requirement for Discretionary Access Control, decreased overall assurance. This was so because they appeared to provide security control through user-set rules which really could not be guaranteed because of how discretionary rights can be propagated once they are granted.

Using immature and evolving techniques seriously complicates research engineering.

Design verification tools, such as Gypsy, were very immature and fell well-short of the promise of verifiable designs of any significant size. We often found ourselves struggling to get the tool to accept proofs that were obviously true. Much of the proof structure had to be re-done to accommodate the idiosyncrasies of the tool, which in turn decreased readability and thus the possibility of independent human review. Further, the tool kept evolving underneath because of serious bugs and improvements from ongoing research, requiring us to re-do quite a bit of work. On the other hand, a goal of the project was investigate A1 design approaches like formal methods, so these experiences, although difficult to manage, were part of the results of the research project.

As another example, in designing the BED module, we used a “pre-evaluated” cryptographic unit called Tepache. This device had just been created and we were its first consumers. It came with the promise of reduced cryptographic evaluation time and cost and easy integration. In retrospect, because of bugs in the design and the lack of buy-in of the module concept within the evaluation community, the use of this "standard" module complicated the design process much more than it helped it.[5]

Overall, LOCK was an important project in the history of trustworthy system design primarily for the important design principles that emerged from the experience. Principles such as Small-and-Real Beats Big-and-Theoretical, Focus on the Unknown, Theories of Security Come from Theories of Insecurity, and Enemy Inside the System, bring unique and important value and perspective to future generations of trustworthy system designers. It is these principles, along with the lessons learned described in this postscript that the reader would be well served to take away from reading this paper.

Acknowledgements

I would like to thank Tom Haigh, Earl Boebert, and Dan Thomsen for their review of the prescript and postscript. Their thoughts prompted a better paper.

References

[1] Saydjari, O. Sami, Joseph M. Beckman, Jeffrey R. Leaman, "LOCKing Computers Securely", 10th Proceedings of the National Computer Security Conference, Ott 1987, pp. 129-141.

[2] Boebert, W. E., "The LOCK Demonstration", Proceedings of the 11th National Computer Security Conference, LOCK Supplement, Ott 1988.

[3] Neumann, P. G., et al, "A Provably Secure Operating System The System, Its Applica-tions, and Proof s," Computer Science Labora-tory Report CSL-1 16, SRI International, Menlo Park, CA, 7 May 1980.

[4] Ford Aerospace and Communications Corporation. "Provably Secure Operating System (PSOS) Final Report." Contract MDA904-80-C-0470. June 1981.

[5] Kemmerer, R., McHugh, J., Haigh, J. T., Young, W. D., "An Experience Using Two Covert Channel Analysis Techniques on a Real System Design." Proceedings of the 1986 Symposium on Security and Privacy, IEEE, April 1986.

[6] Boebert, W. E., Kain, R.Y. and Young, W.D., "Secure Computing: The Secure Ada Target Approach", Scientific Honeyweller, June 1985.

[7] Saltzer, J. D., and Schroeder, M. D., "The Protection of Information in Computer Systems", Proceedings of the IEEE, Vol 63 No 9, March 1975, pp. 1278-1308.

[8] Boebert, W. E., and C. T. Ferguson. "A Partial Solution to the Discretionary Trojan Horse Problem." Proceedings of the 8th National Computer Security Conference. Oct. 1985.

[9] Ferguson, Chuck, and C. B. Murphy. "A Proposed Policy for Dynamic Security Lattice Management." Proceedings of the 9th National Computer Security Conference (Addendum). Sept 1986.

[10] Boebert, W. E., "A Practical Alternative to Hierarchical Integrity Policies", Proceedings of the 8th National Computer Security Confer-ence, pp. 18-27, 30 Sept-3 Ott 1985

[11] W. E. Boebert, "On the Inability of an Unmodified Capability Machine to Enforce the *-Property", 7th Proceedings of the DoD/NBS Computer Security Conference, Sept 24-26 1984, pp. 291-293.

[12] Kain, R. Y.,and Landwehr, C. E., "On Access Cheeking in Capability-Based Systems; IEEE Transactions on Software Engineering, Vol. SE-13, No. 2, Feb 1987, p. 202.

[13] Fabry, R. S., "Capability-Based Addressing", Communications of the ACM, July 1974, Vol. 17, No. 7.

[14] Chen, Thomas M., "SAT Design Overview." Mitre Bedford Working Paper #25867 under contract F19628-84-C-0001, December 18, 1984.

[15] Young, William D. et al., "A Verified Labeler for the Secure Ada Target," Proceedings of the 9th National Computer Security Conference, Sep. 1986.

[16] McHugh, John, "An Emacs-Based Downgrader for the SAT," Proceedings of the 8th National Computer Security Conference, Oct. 1985.

[17] DeMillo, R. A., Lipton, R. J., Perlis, A. J., "Social Processes and Proofs of Theorems and Programs,", Communications of the ACM, Vol 22, No 5, May 1979, p. 271.

[18] Haigh, Tom, and W. D. Young. "Extending the Non-interference Version of MLS for SAT." Proceedings of the 1986 Symposium on Security and Privacy. IEEE, April 1986.

[5] Interestingly, this problem with evaluation of standard modules foreshadowed the same problem we later encountered within the evaluation community for our own higher-level modules.

[19] Sutton, Steve, “Can Unix Be Trusted?”, /usr/group Communications, September/Oct-ober 1987.

[20] Grampp, F. T., and Morris, R. H., “Unix Opera-ting System Security”, AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2, October 1984, pp. 1649-1672.

[21] Schaffer, Mark A., and Walsh, Geoff, “LOCK/ix: On Implementing Unix On The LOCK TCB”, 11th Proceeding of the National Computer Security Conference, October 1988.

[*22] Saydjari, O. Sami, Joseph Beckman, and Jeffrey Leaman, “LOCK Trek”, Proceedings of the IEEE Security and Privacy Conference, April 1987.*