In Foundations of Data Organization, eds. W. Litwin and H.-J. Shek. Springer, 1990, pp. 2-15 (Preprint)

ITOSS: An Integrated Toolkit For Operating System Security.[1]

(Extended Abstract)

Michael Rabin
Aiken Computation Laboratory
Harvard University
Cambridge, MA 02138

J. D. Tygar
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Public and private organizations maintain large systems of files to be accessed by many users. Clearly, access to information in these files must be regulated so that specific items are made available to specific users, in accordance with rules and limitations deemed appropriate by management. The totality of these rules and limitations constitute the (information) *security policy* of the organization.

Nowadays such systems of files reside within computer systems, usually on secondary memory devices such as magnetic or optical disks or magnetic tapes. The files are accessed through computers, on terminals or printers, in a time-shared mode on a single computer, or in a distributed system of computers and file servers linked together. Computer systems are governed by operating systems which, among other tasks, implement and regulate the total user interaction with the system, including user access to the file system. Any hostile impingement on the integrity of the operating system poses grave dangers to the security of the file system, as well as to the proper intended behavior of the computer. Thus security requirements include the protection of operating systems from unauthorized incursion and subversion.

We feel that a method or *model* for implementing secure operating systems should posses the following attributes:

1. It should include mechanisms allowing the transaltion of any desired well specified security policy into the behavior of the given operating system.

[1]This research was supported in part under NSF research grant DCR-81-21431 at Harvard University. The second author received additional support from graduate fellowships from IBM and NSF.

2. It should provide software support and tools for facilitating the above process of translation.

3. It should ensure the proper intended behavior of the system even under malicious attacks.

4. Finally, despite its vital importance, computer security must be achieved at just a modest cost of system performance degradation.

Our approach to the problems of computer security, while bringing to bear some sophisticated algorithms, is at the same time pragmatic. We do not axiomatize the notions and mechanisms of security. Neither do we propose to conclusively demonstrate that a large software system possess certain security properties by carrying out formal verification. In fact, there are solid scientific reasons to believe that such global verification is not possible, and in practice formally verified secure system kernels have been found to have serious weaknesses [Benzel 84], [DeMillo-Lipton-Perlis 79], [Jelen 85], [McLean 85], [McLean 86], [Thompson 84].

What we rather do is to prepare a list of desired and essential properties that a computer system should possess so that a security policy can be reliably and conveniently implemented by users. The issue of security is viewed as that of controlling the access of users to files and of protecting the integrity of the operating system. This is effected through a series of new concepts, mechanisms, calculi, and software support constructs. Taken together these tools (the word is used in the colloquial sense rather than as in "software tools") comprise the *ITOSS (Integrated Toolkit for Operating System Security) model* for computer security. This model is general in scope and applicability and was also implemented in detail for the UNIX BSD 4.2 operating system.

Users act in the system through computing processes which make system calls requesting access to files. In our system, processes $\mathcal{P}$ have privileges V, and files $\mathcal{F}$ have protections T. We develop a formal calculus for representing privileges and protections by *security expressions*, and a semantics for the interpretation of such expressions as having values which are certain set constructs. We define the relation of a *privilege* V *satisfying a protection* T ($V \Rightarrow T$). By stipulation, a process $\mathcal{P}$ with privileges V will be allowed to access a file $\mathcal{F}$ with protection T if $V \Rightarrow T$. An efficient algorithm is developed for determining, for given expressions V and T, whether $V \Rightarrow T$.

Our treatment of privileges and protections includes *non-monotone* privileges, a construct of *indelible* protections; and a mechanism for enforcing *confinement* (see [DOD 85], [Lampson 73]).

The next tool of ITOSS is that of *incarnations*. From an organizational point of view, the significant entity is not the individual user as a person but the *role*, such as department head or bank teller, in which he interacts with the computer system. We therefore enable the organization to dynamically specify a set of entities $I_1, I_2, \ldots$ called *incarnations*,

where each I_i is endowed with the privilege V_i deemed appropriate for the organizational role that I_i represents. The privilege V_i is passed to the processes created on behalf of I_i. A particular user (person) $\mathcal{U}$ will have specific incarnations $I, I', \ldots$, associated with him. When $\mathcal{U}$ logs in he chooses the incarnation representing the role in which he intends to interact with the file system. By use of windows, he can simultaneously interact in several roles, without danger of security tresspasses.

The incarnations mechanism has several benefits. It allows precise tailoring of access privileges to needs of the work to be done by a user in a computer session. Access privileges can be specified according to work roles and can be effortlessly shifted around by system managers simply by removing or adding incarnations to a user.

Up to now we have described passive protection mechanisms. A *sentinel* S is a pointer to an executable file (program), call it F_S. The sentinel S may be incorporated as part of the protection header of a file $\mathcal{F}$. The code in F_S, and the decision whether to protect a particular file F by S, are made by system managers. The operating system includes code which, upon a system call to open a file $\mathcal{F}$, scans the header of $\mathcal{F}$ and creates sentinel processes $\mathcal{S}_1, \mathcal{S}_2, \ldots$, where $\mathcal{S}_i$ runs the code F_{S_i}, corresponding to the sentinels $S_1, S_2, \ldots$, listed in the header. Actually, the sentinels appear in the header in sentinel *clauses* of the form $C \rightarrow S$, where C is a *triggering condition*. The operating system tests C and invokes $\mathcal{S}$ as a process if and only if C is true.

Sentinels have many security and system applications. For example, we may write an audit program $F_{S_{\text{audit}}}$ which will record details of accesses to file $\mathcal{F}$ in a file $\bar{\mathcal{F}}$. For a sensitive file $\mathcal{F}$, one may include the sentinel S_{audit} in the protection. When such a file $\mathcal{F}$ is opened, the sentinel $\mathcal{S}_{\text{audit}}$ is invoked. The system passes to $\mathcal{S}_{\text{audit}}$ certain parameters which may include the file name of $\mathcal{F}$, an identifier of the process $\mathcal{P}$ which made the call to open $\mathcal{F}$, the name of the file $\bar{\mathcal{F}}$ into which the access to $\mathcal{F}$ by $\mathcal{P}$ will be recorded, etc.

It is important to emphasize that, as a rule, the decision to have any particular sentinel S, the code of F_S, the list parameters to be passed to $\mathcal{S}$, and the decision as to which file $\mathcal{F}$ will be protected by a clause $C \rightarrow S$ (and with which C), are all made by the system management. ITOSS provides the general sentinel *mechanism* as a *tool* which may then be employed by management for any purposes deemed useful.

In every computer installation there is a group of users, the system programmers, each of whom necessarily possesses extensive access privileges. Thus for UNIX, system programmers have "root", i.e. total, privileges. This poses very serious security threats. In ITOSS we have the *secure committee* tool. This mechanism allows management, if they so desire, to subject an incarnation I_{comm} to a committee of n users $\mathcal{U}_1, \ldots, \mathcal{U}_n$ (more accurately, n incarnations $I_1, \ldots, I_n$ of these users), a quorum of q of whom are required to invoke and later control I_{comm}.

Secure committees with any specified privileges are created by system management according to need, and the system can support any number of such committees. We shall explain later how to initiate this process so as to ensure security from the start. Secure

committees have additional security applications beyond "watching the guards".

Additional security tools of ITOSS are *fences*. These include a method of "fingerprinting" system calls and files, and comparing fingerprints at appropriate points during the progress of the computation. A discrepancy between a pair of fingerprints which ought to be the same is an indication that some inadvertent or maliciously induced departure from the intended course of the computation has occurred. Fences are incorporated as modules into the kernel and run as part of the kernel code. They serve as a second line of defense against possible security weaknesses. Since it is possible to incorporate flexible variants of fences into operating system kernels *after* their completion, there is a good chance that consequences of any security error in a given kernel will be blocked at runtime by the fence.

In our implementation of ITOSS in conjunction with UNIX 4.2, we introduced fingerprinting at the top level of system calls, and again at the device driver level. This fence uncovered a hitherto unknown security breach arising from a possible race between `link()` and `chmod()` calls referring to a file. Even if the original code were left uncorrected, the unintended change of protection would be consistently blocked by the fence, and the attempted, unauthorized change of protetction of a file would be brought to our attention whenever arising during run time.

A centrally important feature of ITOSS is that when incorporated into an operating system it provides for more than just one option of a security policy. The tools of ITOSS allow the specification and implementation of a wide variety of organizational security policies.

Current proposals for high-security operating systems entail a serious degradation of system performance as a price for enhanced security. An important focus of our work is this issue of efficiency. The overall approach adopted by us, coupled with carefully chosen data structures and very fast algorithms for the frequently repeated security functions, results in a highly efficient system. In tests comparing our system with pure UNIX 4.2, we found no more than a 10% degradation of performance resulting from incorporating ITOSS.

Altogether, we feel that the tools developed in this work lead to the following security advantages:

1. ITOSS provides user-privilege and file-protection structures rich and fine-grained enough to faithfully express and implement any desired security policy. Furthermore, our formalism allows convenient and succinct expression of privileges and protections.

2. The coarseness of privilege/protection structures in existing security systems forces system designers, in certain instances, to confer excessive privileges of access on user and computing processes. These excessive privileges may then be employed by users and processes to subvert file and system security. The rich structure of privileges

and protections of ITOSS, coupled with the tool of incarnations, allows tailoring of user and process privileges to the exact access requirements of the tasks to be performed. This circumvents many possible security pitfalls and dangers.

3. Sentinels provide convenient and reliable mechanisms for guarding against inadvertent or malicious failures of a user to follow security procedures, and for monitoring, auditing, and controlling access to sensitive files.

4. We provide system defenses against so-called "Trojan Horses" in application programs. These "Trojan Horses" are programs submitted by an unscrupulous party for general use within the system, and containing code which when run by an unsuspecting user will illegally read, modify, or destroy his files. These defenses may be effected through the use of incarnations, fences, sentinels, non-monotonic protections, indelible protections and other ITOSS mechanisms.

5. Fences provide a "second line of defense", i.e. measures for detecting unauthorized changes of files and attempts to perform illegal accesses to files, reporting on the former and preventing the latter.

6. In existing computer systems there are always individual users, such as system programmers, with total access privileges. No system safeguards are provided against such pivotal individuals, should they wish to subvert the system's security. The mechanism of secure committees solves this problem.

7. The overhead cost in performance degradation resulting from running ITOSS is small (less than 10% slowdown over pure UNIX 4.2 operation.)

8. By implementing suitable changes, ITOSS can be incorporated into any existing operating system with a relatively modest programming effort, and will confer on that system the security protections of ITOSS.

Finally, a word on how ITOSS will be used. This system is intended for installations and organizations, such as banks, hospitals, corporations, and government departments, where security of information is an important issue. We envision that in such organizations, management will formulate a security policy, i.e. define organizational roles as well as classes of files (depending on security considerations) and specify, based on work needs, for every role, the set of files that can be accessed by users acting in that role.

The computer installation will have "security engineers" who are system analysts and programmers familiar with the tools and provisions of ITOSS, and also with the organization served by the system. These security experts will assist management in formulating the organization's security policy.

Actually we expect that with time and experience, security policies typical to various industries such as banking, retailing, and government, will emerge. Management of a

particular organization will adopt and adapt a standard policy to its needs, without having to start from scratch.

The security engineers will implement the organization's security policy. This will include the creation of classes of privilege and protection expressions; the creation of incarnations; the assignment of privileges to incarnations and of protections to files. After proper system initialization, the assignment of protections to newly created files will as a rule be automated. Security engineers will decide which sentinel programs to choose from a standard library, and what new sentinels to create. They will create and install secure committees for performing extra sensitive tasks.

The day to day maintenance of security will also be in the hands of the security engineers (acting as a secure committee, if so prescribed by policy). They will initially introduce users into the system and assign incarnations to those users according to instructions from management. The security engineers, together with other system programmers (also acting as a secure committee), will be responsible for ongoing changes in the operating system, the file system modifications, and the updating of security features.

The features and mechanisms of ITOSS will be mostly transparent to the ordinary user. The user will be presented, after he logs in using his personal password, with a menu of the incarnations available to him. These will be stated in everyday terms such as: "Manager of Loans", "Member of Committee on Salaries", "Personal Matters." Once the user makes a selection of an incarnation, his access privileges and the protections for the files that he creates are automatically determined. He is oblivious to the details of privileges and protections, and to the existence of sentinels which may guard files that he accesses. The flexibility of ITOSS allows management to depart from the transparent-to-user mode, if so desired. For example, security policy may permit a user to set and modify protections, including sentinels, to his private files which he accesses in the Personal Matters role (incarnation) that may be available to him. All such details are questions of policy, and any kind of policy is implementable in ITOSS.

We feel that this division of labor and responsibilities for security between management, expert security engineers, and users, coupled with the tools and flexibly employable protections of ITOSS, will provide usable, reliable, and appropriate protection for information systems.

Privileges and Protections

From management's point of view the issue of the security of information can be expressed as follows: We have a group of users and a dynamically changing body of information, which for the purposes of this work will be thought of as being organized in units called files. Management wants to define and enforce a regime specifying, for every user $\mathcal{U}$ and every file $\mathcal{F}$, whether $\mathcal{U}$ is allowed to access $\mathcal{F}$.

We view the security problem in the context of an operating system. In this environment the files reside in some kind of storage. Users have computing processes acting on

their behalf.

Thus the security problem is reduced to being able to *specify* for every process $\mathcal{P}$ present in the system and every file $\mathcal{F}$ whether $\mathcal{P}$ will be allowed to access $\mathcal{F}$ and being able to *enforce* the specified regime. On the most general level, such a regime can be specified and enforced in one of the following three equivalent ways: We can create and maintain an *access matrix* M in which $M[i,j] = 1$ if and only if process $\mathcal{P}_i$ is allowed access to file $\mathcal{F}_j$. [Lampson 74] Alternatively, each process $\mathcal{P}_i$ may be provided with a *capability list* $L_i = (j_1, j_2, \ldots)$ so that $\mathcal{P}_i$ may access $\mathcal{F}_j$ if and only if j appears in L_i. The dual to this approach provides each file $\mathcal{F}_j$ with an *access control list* $L'_j = (i_1, i_2, \ldots)$ so that $\mathcal{P}_i$ may access $\mathcal{F}_j$ if and only if i appears in L'_j. When a process attempts access to a file, the operating system checks the access matrix, the capabilities list, or the access control list to see if this access should be permitted. The dynamically changing nature of the ensembles of processes and files and the large number of objects involved render such a regime difficult to specify and to update. Also, large capability lists for processes or long access control lists for files give rise to storage and runtime inefficiencies.

Our approach is to approximate these most general schemes by associating with every process $\mathcal{P}$, a list of *privileges* $\mathcal{V}$ called the *combined privileges* and with every file $\mathcal{F}$ a list of *protections* $\mathcal{T}$ called the *combined protections*. Access of $\mathcal{P}$ to $\mathcal{F}$ is allowed if $\mathcal{V}$ *satisfies* (is sufficient for overcoming) $\mathcal{T}$. We shall do this so as to satisfy the following criteria:

1. The privilege/protection structure must be sufficiently rich and fine grained to allow modelling of any access-control requirements arising in actual organizations and communities of users.

2. A formalism must be available so that users can rapidly and conveniently specify appropriate privileges for processes and protections for files.

3. There must be a rapid test whether a combined privilege $\mathcal{V}$ satisfies a combined protection $\mathcal{T}$.

When thinking about access of a process $\mathcal{P}$ to a file $\mathcal{F}$, we actually consider a number of access modes. For the purpose of this work we concentrate on the following modes:

1. Read

2. Write

3. Execute (i.e. run as a program)

4. Detect (i.e. detect its existence in a directory containing it)

5. Change Protection

This list of modes of accesses can be readily modified or extended to include other modes, according to need. Of these modes, Read, Execute, and Detect are henceforth designated as *non-modifying*, while Write and Change Protection are henceforth designated as *modifying.*

Both the combined privileges and the combined protection each consist of five privileges or protections, respectively, one for each of the access modes. Thus a process $\mathcal{P}$ has five privileges $(V_{\rm rd}, V_{\rm wr}, V_{\rm ex}, V_{\rm dt}, V_{\rm cp})$ associated with it, where $V_{\rm rd}$ is the read privilege, $V_{\rm wr}$ is the write privilege, etc. Similarly, a file $\mathcal{F}$ has five corresponding protections $(T_{\rm rd}, T_{\rm wr}, T_{\rm ex}, T_{\rm dt}, T_{\rm cp})$ associated with it. When process $\mathcal{P}$ makes a system call to read $\mathcal{F}$, the system will check whether $V_{\rm rd}$ satisfies $T_{\rm rd}$ before allowing the access. The other access modes are handled similarly. The detailed implementation is somewhat more complicated, and an check of a privilege may involve looking at several components of the protection. The reason for this complication is to provide for confinement.

From now on we shall treat just a single privilege/protection pair $\langle V, T\rangle$, which can stand for any of the pairs $\langle V_{\rm rd}, T_{\rm rd}\rangle$, etc. In fact, $\langle V, T\rangle$ can stand for $\langle V_X, T_X\rangle$, where X is any additional access mode that an operating system designer may wish to single out.

As will be seen later, each privilege and protection have a finer detailed structure. Thus a privilege V_X, where X is a mode of access, will have several components.

Sentinels: An Overview

The mechanisms described above form a passive, *pure access scheme;* they describe when a process $\mathcal{P}$ can access a file $\mathcal{F}$ but the mechanisms perform no other action. While the pure access scheme gives us much power, there are basic security functions that it can not support. Thus, if we wanted to audit accesses to $\mathcal{F}$, that is record the names of all users who accessed a file $\mathcal{F}$, we would have to either modify the operating system kernel or modify all application code that might access $\mathcal{F}$. Since modifying and testing new code would be a laborious and dangerous process, this solution could only be rarely used and then with great difficulty. We extend the pure access scheme with *sentinels.* Sentinels allow us to conveniently add functions such as auditing. These functions can be customized to reflect any special needs an organization may have.

Here is how a sentinel for a file $\mathcal{F}$ works: Roughly speaking, a sentinel S is a name, listed in a $\mathcal{F}$'s protection header, of an executable file F_S. When any process $\mathcal{P}$ opens $\mathcal{F}$, the operating system schedules F_S for execution as a process $\mathcal{S}$. The process $\mathcal{S}$ is passed some parameters that allow it to perform various operations. The sentinel programs can be written to perform any desired action.

For example, if the security engineers wish to have an audit fucntion, they create a sentinel named, say, $S_{\rm audit}$ and write an appropriate program $F_{S_{\rm audit}}$. The program $F_{S_{\rm audit}}$ will accept as parameters the identity of the process $\mathcal{P}$ accessing a file $\mathcal{F}$, the name for $\mathcal{F}$ and for the file $\bar{\mathcal{F}}$ *into* which the access to $\mathcal{F}$ is recorded, and any other specified paremters. The code of $F_{S_{\rm audit}}$ will realize the desired manner in which the access to $\mathcal{F}$

will be recorded in $\bar{\mathcal{F}}$.

Suppose we only wished to record access when a member of a certain class of users accessed $\mathcal{F}$. We could attach a sentinel S which checked whether the user accessing $\mathcal{F}$ belonged to the class. If he did, S would record the fact; if he did not, S would abort. While this is an adequate solution, it would be even better if we could keep overhead costs down by keeping process creation to a minimum. To allow this, we further extend the form in which a sentinel appears in a protection header to include a *trigger* condition R. Then sentinel S is only scheduled for execution by the operating system kernel when $\mathcal{F}$ is accessed *and* the process meets the trigger condition R.

Enhanced functionality is achieved by separating sentinels into different classes. Auditing simply requires that S make the necessary records in the audit file when $\mathcal{P}$ reads $\mathcal{F}$. A more sophisticated form of sentinels will take action only when certain records are read. For example, certain records in $\mathcal{F}$ might have a keyword `secret` attached to them. We may wish to have a sentinel S guarding $\mathcal{F}$ which will allow most accesses, but restrict access to `secret` records. The first example requires only an asynchronously running process, but the second example requires that S be able to continually monitor and approve individual I/O operations between $\mathcal{P}$ and $\mathcal{F}$. To achieve such mode of operation, we introduce two types of sentinel mechanisms: *asynchronous* sentinels which run independently of $\mathcal{P}$ and *synchronous sentinels* (also called *funnels*) which "lie between" $\mathcal{P}$ and $\mathcal{F}$. By "lie between", we mean that S is able to inspect and approve all I/O operations from $\mathcal{P}$ to $\mathcal{F}$. (In our implementation, funnels utilize UNIX *named pipes*. [Ritchie-Thompson 74], [Kernighan-Plauger 76], [Leffler-Fabry-Joy 83])

A very sophisticated attack might try to read a file $\mathcal{F}$ protected by an asynchronous sentinel S and then crash the operating system before S can perform its function. To prevent this, we further specialize sentinels to allow *Lazarus* processes which will, if interrupted by a system crash, be rerun when the operating system is rebooted.

When a sentinel S runs as a process, it must, as all processes, have some privileges. One choice an implementor could pick would be to have all sentinels run as the system user. This would not conform to the philosophy of this work which always calls for tailoring privileges to be the minimal ones necessary to perform the task at hand. In addition this would mean that we could not allow individual users to attach sentinels for their own purposes. To allow sentinels to be used in the widest possible context, we give each sentinel file F_S an assigned privilege V_S. When the sentinels is scheduled by the operating system, it will run with privilege V_S. A sentinel file is a special case of an exectuable file.

It should be emphasized that we introduce sentinels as an operating system supported mechanism or tool. A particular version of ITOSS may come with a repertory of some standard sentinels which security engineers may use. But the sentinel tool allows security engineers to write any additonal sentinel files to perform useful security functions, and to routinely and effortlessly deploy sentinels as protections for files.

A *sentinel* S is an ordered tuple $\langle F_S, t \rangle$ where F_S is the name of a file, and t is the type, 1, 2, or 3 indicating that the sentinel S is

1. *Asynchronous*,
2. *Synchronous*,
3. *Lazarus*.

Suppose a process $\mathcal{P}$ attempts to access a file $\mathcal{F}$. Suppose further that $\mathcal{P}$'s privilege for that access mode is V_P and $\mathcal{F}$'s protections for that access type are T including a sentinel S. First, the operating system tests whether $V_P \Rightarrow T$. This determines whether $\mathcal{P}$ can access $\mathcal{F}$. But regardless of the result, S can be executed as a sentinel. The operating system checks to make sure that F_S exists and is executable. If it is, $\mathcal{S}$ is created by the operating system with privileges V_S and executes according to type t.

To reduce unnecessary executions of F_S from being activated, we would like to express a trigger condition R specifying when S should be run by the operating system. There are several possibilities for expressing conditions R. We chose to use our privilege/protection scheme for this purpose, but it is easy to imagine other good choices for expressing trigger conditions.

Incarnations and Secure Committees: Overview

A centrally important issue in secure operating and file systems is the correct and prudent management of privileges and protections, i.e., of access rights to files. There are several problems and dangers arising from the way access rights are assigned to *users* in existing systems.

A person often has a number of roles in which he is active within an organization, and which give rise to his interaction with the file system. In existing systems a user is usually presented to the operating system by a single entity through his login name, and this entity determines his access rights without regard to the purpose of his current computer session.

Assume that a user wishes to play a computer game. The program that he runs may have been inadequately tested and may contain Trojan Horse code. But the user's process running the program has access to *all* the files available to that user, so that the Trojan Horse code can cause serious damage.

Another difficulty arises when we have to add or revoke access rights. Assume that person $\mathcal{A}$ is in charge of a certain department and is chairman of a certain committee. These roles require access rights to two, possibly overlapping, sets of files. When person $\mathcal{A}$ is replaced by person $\mathcal{B}$ as chairman, the revocation of his access rights to the relevant files, and the assignment of these rights to $\mathcal{B}$ is a cumbersome and error prone process in existing systems.

In ITOSS the basic entities on whose behalf computing processes run are abstracted as *incarnations*. Each incarnation has combined privileges which will be attached to the

processes created by it, and a scheme for attaching combined protections to the files created by these processes.

The set of incarnations is specified and dynamically updated by management, and generally reflects the organizational work roles. Each user has a number of associated incarnations with his login. When he logs in, he selects an appropriate incarnation from a menu of the incarnations available to him.

Consider the user $\mathcal{A}$ mentioned before. The security engineers create incarnations $\mathcal{I}_{\text{head}\,X}$, $\mathcal{I}_{\text{chmn}\,Y}$ to represent the roles of head of Department X and chairman of Committee Y, and an incarnation $\mathcal{I}_{\text{min}}$ with minimal privileges. If user $\mathcal{A}$ wishes to run the game program, he invokes the incarnation $\mathcal{I}_{\text{min}}$ which has no access rights to any significant files. If $\mathcal{A}$ is replaced by $\mathcal{B}$ as chairman of Committee Y, the security engineers remove $\mathcal{I}_{\text{chmn}\,Y}$ from the incarnations available to $\mathcal{A}$ and associate it with $\mathcal{B}$.

A concern equally important to the aforementioned control of privileges of processes is the correct assignment of protections to files. ITOSS proves means for *automatic assignment* of headers, including combined extended protecteions, to files. This stands in contrast to most previous schemes which placed the brunt of responsibility for protection files on the individual user who created the file. In practice, the prevailing approach worked poorly: non-expert users who did not fully understand the working of the operating system would give incorrect security specifications. ITOSS, with its far richer ensemble of security mechanisms, would be an even greater challenge to non-expert users. Moreover, even well-informed users acting in an uncoordinated fashion could not properly manager the assignment of the global resource or individual securons.

Our innovation allws us to shift this responsibility to the security engineers. The protection assignment is performed by an operating system mechanism which takes into account the context in which a given file was created, insuring that filew will always be created with appropriate protections.

A prevalant difficulty with system security features is that they are generally unused by users of the computing system. The automated nature of ITOSS solves this problem.

In existing systems there are always some users, such as the system programmers, who have access to all of the system's resources and files. This arrangment poses obvious security dangers. In ITOSS, every role is represented by an incarnation. Usually an incarnation is assigned to a user who may in addition control several other incarnations, i.e., one user controls several incarnations. Dually, the *secure committee* tool allows management to subject control of any incarnation $\mathcal{I}$ to a *committee* of n users, so that a *quorum* of q committee members is required to invoke the incarnation $\mathcal{I}$ and execute commands through it.

Assume that the security engineers decide to create a secure committee SCOMM with certain access privileges, and wish to have SCOMM governed by a quorum of at least q of the users $\mathcal{U}_1, \ldots, \mathcal{U}_n$. They create a *committee incarnation* $\mathcal{I}_{\text{comm}}$ and *committee-member* incarnations $\mathcal{I}_1, \ldots, \mathcal{I}_n$ where $\mathcal{I}_j$ is assigned to user $\mathcal{U}_j$. If $k \geq q$ users $\mathcal{U}_{i_1}, \ldots, \mathcal{U}_{i_k}$ need

to invoke $\mathcal{I}_{\text{comm}}$, then each $\mathcal{U}_{i_j}$ selects his committee-member incarnation $\mathcal{I}_{i_j}$. Every $\mathcal{U}_{i_j}$, $1 \leq j \leq k$ then makes through $\mathcal{I}_{i_j}$ a system call to invoke $\mathcal{I}_{rmcomm}$. These calls by $\mathcal{I}_{i_1}, \ldots, \mathcal{I}_{i_k}$ may involve, for greater security, "pieces" of a secret password in a manner explained in Section 4.3. The system creates $\mathcal{I}_{\text{comm}}$ only after having such $k \geq q$ calls from SCOMM members.

The operating system records the identifiers of the incarnations $\mathcal{I}_{i_1}, \ldots, \mathcal{I}_{i_k}$ (here $q \leq k$) representing the users participating in the session. After the incarnation $\mathcal{I}_{\text{comm}}$ is invoked, every command by a joint shell owned by $\mathcal{I}_{\text{comm}}$ must be approved by all committee members participating in the session. Each participating member acts from his terminal and gets every system command for his inspection and approval.

The creation of secure committees, like all other security functions in ITOSS, is entrusted to the security engineers. As mentioned in the introduction, the security engineers and system programmers may also be organized into various secure committees, with possibly different quorums depending on the tasks and access privileges of the committee in question. **Validation**

Once an implementor has specified a language for expressing security constraints and provided a mechanism for enforcing them, the task he faces is to *validate* the resulting system so to show that it is free of errors. Validation is an important issue not just for security but for software engineering in general, and a large number of methods, such as formal verification, testing, structured walkthroughs, have been proposed for dealing with this problem. In practice, none of these methods guarantee software without errors; they merely increase the confidence a user has in the system.

Validation for security is special because in many cases we are trying to prohibit some event from occurring. We propose a general method, *fences*, for providing a "second test" of security conditions.

The term "fence" was first applied to the IBM 7090 computer to describe a memory protection mechanism. [Bashe *et al* 86] In this context, a fence was a pointer into memory which separated user and system memory. Memory beyond the fence was accessible only in system mode, and this was enforced by independent hardware.

In our usage, a fence is any low-overhead hardware or software feature which enforces security conditions by testing values independently of the main stream of execution, allowing operations to be performed only if they do not violate security conditions.

Fingerprints

In the course of his research on string matching, the first author proposed a special hash function, called a *fingerprint*. [Rabin 81] His fingerprint function $F_K(x)$ hashes a n-bit value x into a m-bit value ($n > m$) randomly, based on a secret key K. The interesting point is that given a y, if K is unknown, then no one can find an x such that $F_K(x) = y$ with probability better than 2^{-m}.

(Briefly, Rabin's algorithm picks an irreducible polynomial p of degree m over the integers modulo 2. The coefficients of p, taken as a vector, form the key K. The bits in the input x are taken as the coefficients of a $n-1$ degree polynomial q. Let r be the residue of q divided by p in $Z_2[x]$. r is a $m-1$ degree polynomial, and its coefficients, taken as a vector, form $F_K(x)$. A software implementation of this algorithm merely consists of a sequence of very fast XOR operations. [Rabin 81] gives this algorithm in greater detail. [Fisher-Kung 84] describes a very fast systolic hardware implementation of this algorithm.)

With the fingerprinting algorithm, we can install powerful fences. Suppose we wish to guarantee that a file $\mathcal{F}$ has not been tampered with. One way we could protect against this is by installing a synchronous sentinel to guard $\mathcal{F}$. However, $\mathcal{F}$ would still be vulnerable to attacks on the physical disk. As a second-tier protection, we could have the synchronous sentinel guarding $\mathcal{F}$ keep an independent fingerprint of $\mathcal{F}$ elsewhere in the operating system. If $\mathcal{F}$ was changed illicitly, the sentinel would instantly detect it unless the fingerprint was also changed. Since the fingerprint is provable impossible to forge with accuracy greater than 2^{-m} unless the key K is known, it is impossible for the opponent to change $\mathcal{F}$ without eventually coming to the attention of the sentinel.

Bibliography

[Bashe 86] Bashe, C. J., L. R. Johnson, J. H. Palmer, and E. W. Pugh. *IBM's Early Computers* MIT Press, Cambridge, Massachusetts, 1986.

[Benzel 84] "Analysis of a Kernel Verification." *Proceedings of the 1984 Symposium on Security and Privacy*, Oakland, California, May 1984, pp. 125–131.

[Daley-Dennis 68] Daley, R. C., and Dennis, J. B. "Virtual Memory, Processes, and Sharing in MULTICS." *Communications of the ACM,* **11:**5, pp. 306–312 (May 1968).

[DeMillo-Lipton-Perlis 79] DeMillo, R. A., R. J. Lipton, and A. J. Perlis. "Social Processes and Proofs of Theorems and Programs." *Communications of the ACM,* **22:**5, (May 1979).

[Dijkstra 68] Dijkstra, E. W. "The Structure of the 'THE' Multiprogramming System." *Communications of the ACM,* **11:**5, pp. 341–346 (May 1968).

[DOD 85] *Trusted Computer System Evaluation Criteria.* Computer Security Center, Department of Defense, Fort Meade, Maryland. (CSC-STD-001-83) March 1985.

[Grampp-Morris 84] Grampp, F. T., and R. H. Morris. "UNIX Operating System Security." *AT&T Bell Laboratories Technical Journal,* **63:**8b, pp. 1649–1672 (October 1984).

[Jelen 85] Jelen, G. F. *Information Security: An Elusive Goal.* Program on Information Resources Policy, Harvard University, Cambridge, Massachusetts. June 1985.

[Lampson 73] Lampson, B. W. "A Note on the Confinement Problem." *Communications of the ACM,* **16:**10, pp. 613–615 (October 1973).

[Lampson 74] Lampson, B. W. "Protection." *ACM Operating Systems Review*, **19**:5, pp. 13–24 (December 1985).]

[McLean 85] McLean, J. "A Comment on the 'Basic Security Theorem' of Bell and La-Padula." *Information Processing Letters,* **20:**3, pp. 67–70 (1985).

[McLean 86] McLean, J. "Reasoning About Security Models." Personal Communication, 1986.

[Organick 72] Organick, E. I. *The Multics System.* MIT Press, Cambridge, Massachusetts, 1972.

[Rabin 81] Rabin, M. O. "Fingerprinting by Random Polynomials." TR-15-81. Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts. 1981.

[Ritchie-Thompson 74] Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System." *Communications of the ACM,* **17:**7, pp. 365–375 (July 1974).

[Schroeder-Saltzer 72] Schroeder, M. D., and J. H. Saltzer. "A Hardware Architecture for Implementing Protection Rings." *Communications of the ACM,* **15:**3, pp. 157–170 (March 1972).

[Shamir 79] Shamir, A. "How to Share a Secret." *Communications of the ACM,* **22:**11, pp. 612–613 (November 1979).

[Thompson 84] Thompson, K. "Reflections on Trusting Trust." *Communications of the ACM,* **27:**8, pp. 761–763 (August 1984).

[Wulf-Levin-Harbison 81] Wulf, W. A., R. Levin, S. P. Harbison. *HYDRA/C.mmp.* McGraw-Hill, New York, NY, 1981.