J. D. Tygar, Ph.D. dissertation, Harvard University

An Integrated Toolkit for Operating System Security

A thesis presented

by

Justin Douglas Tygar

to

The Division of Applied Sciences

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

August 1986

©1986 by Justin Douglas Tygar

All rights reserved.

Abstract

This thesis reports on joint research with Michael Rabin.

Issues of operating system security occupy a central role in applied computer science; yet there has been no satisfactory complete solution to the problem of computer security. In the Security Toolkit Project we have developed a number of novel interlocking techniques which can be combined in many ways to provide tremendously enhanced security. The security toolkit, ITOSS, uses little overhead and is flexible; a security engineer can tailor a particular configuration to exactly satisfy the security needs demanded by the organizational structure at the site. It opens the way for further experimental work, rapid development, and simulation of new security schemes. The project consists of four phases: creating a new model of security, implementing the model by modifying UNIX (4.2 BSD) on a SUN-2 computer, inventing new algorithmic fences to validate the software by insuring that security violations are computationally infeasible, and developing software to permit easy exploitation of ITOSS's features.

Acknowledgements

This thesis reports on joint research with Michael Rabin.

Michael Rabin, more than any other person, deserves thanks for his help on the security toolkit project. It has been a special privilege to work with Michael, who not only served as my advisor, but also vigorously collaborated in this research. His influence can be seen on every page of this thesis.

It is a pleasure to acknowledge the help of:

Ugo Gagliardi, for serving on my research committee and pointing out certain duality conditions in operating systems; John Reif, for serving on my research committee and helping me explore connections with number theory and randomness; and Leslie Valiant, for serving on my research committee and discussions on complexity theory.

Danny Krizanc, for listening to many versions of this research and providing me with careful criticism; Harry Lewis and Peter McKinney, for their help in clearing legal paperwork involved in obtaining access to UNIX software; Silvio Micali, for showing me some literature on cryptography; David Shmoys, for his suggestions on the presentation of this material; and Straff Wentworth, for several long explanations of conventional approaches to computer security.

Ron Ellickson, Lou Scheffer, and Curt Widdoes, for giving me an opportunity to test my skills in an industrial environment; Bart Miller, for introducing me to the internals of UNIX; and Steve Sargeant, for his explanations of the obscure corners of UNIX.

I am indebted to my associates at Aiken Laboratoy for providing a warm, collegial environment for doing research. Special thanks to Glen Dudek, Carol Harlow, Nike Horton, and Eleanor Sacks for clerical and technical support. Thanks to Lisa Neal for her help in the final preparation of this thesis.

I dedicate this thesis to my parents and my sister.

Financial support for this thesis was provided by a National Science Foundation Graduate Fellowship, an IBM Graduate Fellowship, and National Science Foundation Grant MCS 81-21432 at Harvard Univeristy.

UNIX is a trademark of AT&T; SUN is a trademark of Sun Microsystems.

Contents

1 What is Security? 7

1.1 ITOSS . . . 7

1.1.1 The Security Problem . . . 7

1.1.2 Outline of Thesis . . . 9

1.2 Previous Work . . . 10

1.2.1 Capability Lists . . . 10

1.2.2 Simple Hierarchies . . . 10

1.2.3 Formal Verification . . . 11

2 Pure Access Control 13

2.1 Privileges and Protections . . . 13

2.2 The Tree of Securons . . . 15

2.3 Security Expressions . . . 17

2.4 An Algorithm for Testing $V \Rightarrow T$ . . . 20

2.5 Negative Privileges and Protections . . . 22

2.6 Indelible Protections and Confinement . . . 23

2.7 Bypass Privileges . . . 28

2.8 Gradations of Actions on Files . . . 29

3 Sentinels 32

3.1 Overview . . . 32

3.2 Semantics of Sentinels 34
3.3 Types of Sentinels 36
3.4 Examples 37
3.5 Previous Work 38

4 Incarnations and Secure Committees 40
4.1 Overview 40
4.2 Incarnations 41
4.3 Secure Committees 42

5 Fences 45
5.1 Validation 45
5.2 Fingerprints 46
5.3 System Call Fingerprints 47

6 A Prototype Implementation 49
6.1 Performance 49
6.2 How to Initially Configure ITOSS 50
6.3 Some Implementation Details 51
6.3.1 Peculiarities 51
6.3.2 File System Alterations 52
6.3.3 Internet 52
6.3.4 Hooks for Fingerprints 52
6.3.5 System Calls 53
6.3.6 Compatibility Mode 53

7 Open Problems 54

8 Bibliography 57

Chapter 1

What is Security?

1.1 ITOSS

ITOSS, the Integrated Toolkit for Operating System Security, is a prototype implementation of a set of tools addressing several basic security problems arising in the design of operating systems. While this thesis is concerned with a specific realization of these tools, the underlying model and techniques can be used in a wide variety of operating systems. Whenever possible, this thesis will present the methods in the most abstract form available.

1.1.1 The Security Problem

We presume two types of entities: *users*, denoted $\mathcal{U}_1, \mathcal{U}_2, \ldots$, and *files*, denoted $\mathcal{F}_1, \mathcal{F}_2, \ldots$. User $\mathcal{U}_i$ may deploy processes to attempt to find, use, or change the contents of file $\mathcal{F}_j$. The set of possible operations a user may attempt to apply to a file is determined by a fixed set of access types. Examples of access types are *read*, and *write*.

Security problems revolve around the issues of permitting or denying a particular user $\mathcal{U}_i$ access to a particular file $\mathcal{F}_j$ under an access type X.

Broadly, security can be viewed as several layers of mechanisms designed to enforce proper use of a given system. Here are five layers of interest to us:

Application: The ultimate purpose of an operating system is to allow users to run application programs. The security problem at this layer is to insure that application programs perform as required. (For example, issues of statistical data base security, which address methods to keep users from inferring information about specific subjects from a statistical data base, fall into this category.)

Supervision: Operating systems are manipulated by humans. At this layer, we are concerned with providing straightforward tools must be provided which easily allow appropriate authorities to correctly specify security conditions. Tools may also be needed to report on actions that users take while executing programs in the operating system. These tools may be used by managers, to specify broad conditions which the operating system must satisfy, or by individual users, to specify accesses permitted to individual files. These tools may be manual, requiring specific execution, or automatic, being triggered without specific execution.

Access Control: Security issues at this layer focus on declaring the conditions under which a file may be accessed. The primitives applied at this layer may be used at the Supervision layer to allow broad structures to be realized.

Kernel: Abstract mechanisms provided at the access control layer must be implemented. At this layer we are concerned with the efficient implementation of access control structures. We also need ways of insuring that the implementation does not contains errors which provide security holes.

Physical: Without physically securing a computer, it is not possible to insure any higher level of security will function correctly. For example, it is meaningless to speak of protections keeping a user from reading a file if he can simply walk away with the disk and examine it at his leisure. Physical security implies that all processors, peripheral devices, and wires are protected against unauthorized surveillance and tampering.

1.1.2 Outline of Thesis

In this thesis we presuppose physical security and concern ourselves with security issues at the kernel, access control, and supervision layers. The tools we provide to address these problems can also be used to help with security at the application layer.

Chapter 2, presents an access control model which does not presume any specific structure on the set of users which can specify a broad range of security structures. The range of structures our access control model can embrace includes all conventional security structures. This chapter describes algorithms for efficiently implementing the access control model.

Chapter 3 describes a security tool which can monitor user activities and arbitrarily extend our access control mechanism. This mechanism can provide automatic actions which enforce security.

Chapter 4 explores how one could structure a front end which allows manual supervision of security. This chapter introduces the notion of collective security decision making. Further, it indicates how separate user activities can be made secure by independent security structures.

Chapter 5 describes some methods for detecting some types of hidden errors in an implementation. While these methods do not guarantee that the resulting implementation is free of error, they have been successfully used to find bugs which were not discovered by more traditional validation approaches.

Chapter 6 summarizes the results of building a prototype system which extended UNIX (4.2 BSD)[1] to include our security model and tools. This chapter also discuss how this system may be used in practice.

Chapter 7 lists future work and open problems.

---

[1]See [Berkeley 84].

1.2 Previous Work

Many researchers have attacked special parts of the operating security problem. We will focus our attention on three approaches: (1) *capability lists*; (2) *simple hierarchical approaches*; (3) *formally verified approaches.*

1.2.1 Capability Lists

Capability lists enumerate files a process may access. Since any pattern of accesses can be stored in a capability list, they provide a very general mechanism for providing security. The most sophisticated of the security structures supported under capability lists was provided by the Hydra operating system at Carnegie-Mellon [Wulf-Levin-Harbison 81] which showed how wide classes of classical security problems could be addressed in a capability-based operating system.

Unfortunately, capability lists do not by themselves provide an effective solution to the security problem since they only address the access control layer of security, ignoring the supervision layer and kernel layer. At the supervision layer, [Lampson-Sturgis 76] have criticized Hydra and other capability-based systems for "drastically underestimating the work required to make the system usable by ordinary programmers." Because of the level of detail that information must be managed, it is difficult to correctly configure a capability-based operating system. At the kernel layer, algorithms for manipulating capability lists require time quadratic in the length of the lists and, as [Wulf-Levin-Harbison 81] shows, can result in high overhead.

1.2.2 Simple Hierarchies

A completely different approach to security is to make it as simple as possible. MULTICS is one of the most elegant and successful examples of this approach. [Organick 72], [Schroeder-Saltzer 72]. MULTICS uses a ring structure to impose a

hierarchy of security: very secret files, secret files, . . . , unsecret files. This approach meets its goals but limits the range of security structures that may be imposed.

As we shall see in Chapter 3, this proves to be a serious limitation and tends to generate security holes.

1.2.3 Formal Verification

In the formal verification approach to security, we start from the assumption that only mathematical proof can establish that operating system code is free from error. A very simple hierarchical security system is established, and a small section of critical code[2] is formally verified to "prove" that it satisfies the conditions established. The Department of Defense has proposed a simple model to serve as a standard for evaluating formally verified systems. [DOD 85]

Unfortunately, the technology for establishing formal verification is not yet at the state where a significant length of code can be verified. In the most sophisticated attempt to date, described in [Benzel 84], at least seven security holes slipped through the formal verification process. In his ACM Turing Award lecture, Ken Thompson showed how to include a security hole which can not be detected by *any* source code verification process. [Thompson 84] John McLean has shown substantial defects in the standard security model. As an example, there is nothing to prevent such a model from spontaneously removing all security constraints. He also points out that all available tools for verifying systems are formally unsound, that is, one can derive false assertions from the tools. [McLean 85], [McLean 86]. (See [DeMillo-Lipton-Perlis 79] for a discussion of limitations of formal verification.)

Even more serious, the formal verification approach introduces serious performance degradation. A typical example is an attempt to make VM/370 secure. The resulting operating system was so slow that even in unsecure operation data was

[2] This small section of code is called the "security kernel". This terminology is confusing since the security kernel is not necessarily related to the operating system kernel. [Jelen 85]

transmitted slower than 100 bits/second; meeting government standards for the maximum rate at which information could be securely transmitted. [Gold-Linde-Cudney 84] Formal verification typically produces an order of magnitude slowdown.

Chapter 2

Pure Access Control

2.1 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 memory (usually in secondary storage from which they are called on demand). 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 two 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 $\mathcal{F}_j$ iff j

appears in L_i When a process attempts access to a file, the operating system checks the access matrix or the capabilities list to see if this ought to 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 inefficient to enforce.

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

1. The privilege/protection structure must be sufficiently rich 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 privilege V satisfies a protection 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

(Of these modes, Read, Execute, and Detect are designated as *non-modifying*, while Write and Change Protection are designated as *modifying.*)

The total privilege/protection structures are split into five segments, one for each of the access modes. Thus a process $\mathcal{P}$ has five privileges $(V_{\mathrm{rd}}, V_{\mathrm{wr}}, V_{\mathrm{ex}}, V_{\mathrm{dt}}, V_{\mathrm{cp}})$ associated with it, where V_{rd} is the read privilege, V_{wr} is the write privilege, etc. Similarly, a file $\mathcal{F}$ has five corresponding protections $(T_{\mathrm{rd}}, T_{\mathrm{wr}}, T_{\mathrm{ex}}, T_{\mathrm{dt}}, T_{\mathrm{cp}})$ associated with it. When process $\mathcal{P}$ makes a system call to read $\mathcal{F}$, the system will check whether V_{rd} satisfies T_{rd} before allowing the access. The other access modes are handled similarly.

From now on we shall treat just a single privilege protection pair $\langle V, T\rangle$, which can stand for any of the pairs $\langle V_{\mathrm{rd}}, T_{\mathrm{rd}}\rangle$, 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 one of the above five modes of access, will have several components.

2.2 The Tree of Securons

The basic atomic units out of which all privileges and protections are composed are nodes, called *securons*, of a specific tree. Since customers such as government departments or corporations, usually have tree-like hierarchical organizational structures, the tree of securons is a natural domain into which to map the desired security structure of the organization in question.

Definition 2.1 *The* securon trees *of* width n *and* depth h *is the set* $\mathbf{str}(n,h)$ *of all strings* $0.i_1.i_2.\cdots.i_k$ *where* $i_j \in [0\ .\ .\ n-1]$*, and* $k \leq h$*.*

If $x, y \in \mathbf{str}(n,h)$ *and* $0 \leq i \leq n-1$*, then* y *is the* i*th* child *of* x *if* $y = x.i$*. The strings* x *and* y *are* related *if* x *is an initial sequence of* y *(denoted by* $x \leq y$*) or*

$y \leq x$.

The depth *of the securon* $x = 0.i_1.\cdots.i_k$ *is* $d(x) = k$.

In every particular implementation of a specific tree $\mathbf{str}(n, h)$ is used. In the current version of ITOSS, $n = 256$ and $h = 15$. We shall henceforth denote by **str** the fixed securon tree used in the architecture of our secure system.

As a first approximation we are tempted to use the securons themselves as privileges and protections. Thus we may assign to a manager the securon $x = 0.15.19.7$ and to his 15 subordinates the securons $0.15.19.7.i$, $0 \leq i \leq 14$. If a subordinate creates a file $\mathcal{F}$, then his securon y is attached as the protection $T = y$ of this file. A process $\mathcal{P}$ owned by the manager will have the privilege $V = x$. We can then stipulate that when the process $(\mathcal{P}, V)$ tries to access the file $(\mathcal{F}, T)$, permission will be granted only if $x \leq y$ (x is an initial sequence of y). We feel that such arrangements, and their obvious extensions and modifications, are not powerful and rich enough in structure to reflect the security structure we need. In almost all existing security models, however, privileges and protections are linearly ordered (public, restricted, secret, etc.), i.e., even more limited than the above arrangement. We want to have more sophisticated objects for expressing privileges and protections and a more flexible definition for the notion of a privilege satisfying a protection.

Definition 2.2 Privileges *are sets* $V \subseteq \mathbf{str}$ *of securons and* protections T *are sets of sets of securons (i.e.* $T \subseteq \mathbf{P}(\mathbf{str})$*, where* $\mathbf{P}(S)$ *denotes the set of all subsets of* S*).*

A privilege

$$V = \{s_1, \ldots, s_m\}, \quad s_i \in \mathbf{str}$$

satisfies *(is sufficient to overcome) the protection*

$$T = \{U_1, \ldots, U_m\}, \quad U_i \subseteq \mathbf{str}$$

if for some j, $1 \leq j \leq k$ *we have* $U_j \subseteq V$. *We shall use the notation*

$$V \Rightarrow T$$

to denote that V *satisfies* T.

These concepts allow us to create very detailed privilege/protection schemes. Thus if we want to express the relation between manager and subordinates described above, we assign to the manager's processes the privilege

$$V = \{0.15.19.7.i \mid 0 \leq i \leq 14\},$$

and to files of the ith subordinate, the protection $T_i = \{\{0.15.19.7.i\}\}$ (not $\{0.15.19.7.i\}$!) Now $V \Rightarrow T_i$ holds for all subordinates. If we wish to make subordinate 1's files accessible to an additional user $\mathcal{M}$, we can assign to $\mathcal{M}$ a securon s and define his set of securons to be $V_{\mathcal{M}} = H \cup \{s\}$, where H represents privileges required by $\mathcal{M}$ in other contexts. We also set $T_1' = T_1 \cup \{\{s\}\}$. Now we have $V \Rightarrow T_1'$ as well as $V_{\mathcal{M}} \Rightarrow T_1'$. We can, in fact, realize any access matrix by means of a sufficiently detailed assignment of privileges and protections.

2.3 Security Expressions

The above apparatus for implementing privileges and protections is indeed powerful but, as it stands, cumbersome. If privileges $V \subseteq \mathbf{str}$ and protections $T \subseteq \mathbf{P}(\mathbf{str})$ are to be specified by enumeration, then both the assignment of privileges and protections, and the testing of whether $V \Rightarrow T$ will be too difficult to be of practical use. What we need is a formal calculus which will allow us to write quickly and economically compact formal expressions denoting rich and complicated privilege and protection sets. The first need is to describe large subsets of **str**. This will be done by introducing securon terms which will also be the atomic terms for building expressions.

Definition 2.3 *Let* $s \in \mathbf{str}$ *and* $0 \leq i \leq j \leq \mathit{depth}(\mathbf{str})$*.* Securon terms *are:*

1. s

2. $s[i \ \mathbf{downto}\ j]$

Definition 2.4 *The set* $SET(t)$ *defined by a term* t *is* $SET(s) = \{s\}$ *for* $s \in \mathbf{str}$*, and*

$$SET(s[i \ \mathbf{downto}\ j]) = \{u \mid u \in \mathbf{str}, u \text{ related to } s, i \leq d(u) \leq j\}$$

Recall that u related to s means u is an ancestor, descendent, or is equal to s. Thus $SET(0[0 \ \mathbf{downto}\ \mathit{depth}(\mathbf{str})]) = \mathbf{str}$. And

$$SET(s[d(s)+1 \ \mathbf{downto}\ d(s)+1])$$

is the set of all children of the securon s.

Definition 2.5 *A* privilege expression *is defined by*

1. *Any securon term is a privilege expression.*

2. *If* E_1 *and* E_2 *are privilege expressions then so are* $E_1 \wedge E_2$*.*

Definition 2.6 *A* protection expression *is defined by*

1. *Any securon term is a protection expression.*

2. *If* E_1 *and* E_2 *are protection expressions then so is* $E_1 \wedge E_2$ *and* $E_1 \vee E_2$*.*

A *(security) expression* is a privilege or protection expression. Note that every privilege expression is a protection expression but not *vice versa.*

We must give semantics for these formal expressions, i.e., rules for associating a set $V \subseteq \mathbf{str}$ with a privilege expression, and for associating a set of sets $T \subseteq \mathbf{P}(\mathbf{str})$ with a protection expression. We shall use the notation $v_{\mathrm{priv}}(E)$ to denote the privilege defined by the expression E, and $v_{\mathrm{prot}}(E)$ to denote the protection values.

If t is a securon term then we want to include in the corresponding privilege all the securons in $SET(t)$. If $E = E_1 \wedge E_2$ is a privilege expression, we want the corresponding privilege to be the weakest privilege stronger than the privilege corresponding to E_1 as well as the privilege corresponding to E_2. This motivates the following definition:

Definition 2.7 *The function $v_{\text{priv}}(E)$, giving the privilege corresponding to the expression E, is defined by*

1. $v_{\text{priv}}(s) = \{s\}$, *for* $s \in \mathbf{str}$.

2. $v_{\text{priv}}(s[i \textbf{ downto } j]) = \{u \mid u \in \mathbf{str},\ i \leq d(u) \leq j\} = SET(s[i \textbf{ downto } j])$.

3. $v_{\text{priv}}(E_1 \wedge E_2) = v_{\text{priv}}(E_1) \cup v_{\text{priv}}(E_2)$.

When it comes to protections, we want the protection corresponding to a term t to be the set containing *all singleton sets* $\{u\}$ for $u \in SET(t)$. We want the protection $E = E_1 \vee E_2$ to mean that any privilege satisfying E_1 or E_2 also satisfies E. Finally, having the protection $E = E_1 \wedge E_2$ should mean that $V \Rightarrow E$ if and only if $V \Rightarrow E_1$ and $V \Rightarrow E_2$.

For sets (of sets) $T_1, T_2 \subseteq \mathbf{P}(\mathbf{str})$ we introduce the notation of cartesian product given by

$$T_1 \times T_2 = \{U_1 \cup U_2 \mid U_1 \in T_1,\ U_2 \in T_2\}$$

Notice that $T_1 \times T_2$ is again a set of subsets of $\mathbf{str}$.

Definition 2.8 *The function $v_{\text{prot}}(E)$ giving the protection corresponding to the expression E is defined by*

1. $v_{\text{prot}}(s) = \{\{s\}\}$.

2. $v_{\text{prot}}(s[i \textbf{ downto } j]) = \{\{u\} \mid u \in SET(s[i \textbf{ downto } j])\}$.

3. $v_{\text{prot}}(E_1 \vee E_2) = v_{\text{prot}}(E_1) \cup v_{\text{prot}}(E_2)$.

4. $v_{\rm prot}(E_1 \wedge E_2) = v_{\rm prot}(E_1) \times v_{\rm prot}(E_2)$.

If V is a privilege *expression* and T is a protection *expression* then V *satisfying* T ($V \Rightarrow T$) will of course be defined to mean $v_{\rm priv}(V) \Rightarrow v_{\rm prot}(T)$, where the latter relation was explained in Definition 2.2. From now on, we shall talk about privileges and protections as either expressions or sets, and the intended meaning, when not specified, will be clear from context.

2.4 An Algorithm for Testing $V \Rightarrow T$

The formalism of security expression developed in Section 2.3 allows us to specify, by writing concise expressions, large, complicated privilege and protection sets. We need an efficient algorithm for testing whether a privilege satisfies a protection. Let $V = E_1 \wedge \ldots \wedge E_m$, where each E_i is a securon term, be a privilege (expression) and let T be a protection. Thus $T = G_1 \vee G_2$, or $T = G_1 \wedge G_2$, or T is a securon term. In the first case we test whether $T \Rightarrow G_1$, and if this fails we test whether $T \Rightarrow G_2$. In the second case we test whether $T \Rightarrow G_1$, if this fails then $T \not\Rightarrow V$, otherwise we continue to test whether $T \Rightarrow G_2$. Note that we adopt the so called *short-circuit evaluation* mode, where irrelevant branches are not pursued.

Thus, recursively, our problem is reduced to testing whether $T \Rightarrow t$ where t is a term, say, $t = s[i \ \textbf{downto} \ j]$. The case $t = s$ is, trivially, a special instance of the former case. Let

$$T = s_1[i_1 \ \textbf{downto} \ j_1] \wedge \ldots \wedge s_m[i_m \ \textbf{downto} \ j_m].$$

Then $T \Rightarrow s[i \ \textbf{downto} \ j]$ if and only if for some $k, 1 \leq k \leq m$,

$$SET(s_k[i_k \ \textbf{downto} \ j_k) \cap SET(s[i \ \textbf{downto} \ j]) \neq \emptyset. \qquad (2.1)$$

To test (2.1) we introduce, for any $u \in \textbf{str}$ and integer $\ell \leq d(u)$, the notation

$INIT(u, \ell)$ to denote the unique string v satisfying:

$$INIT(u, \ell) = \begin{cases} v & \text{where } v \leq u \text{ and } d(v) = \ell, \text{ if } \ell < d(u) \\ u & \text{if } d(u) \leq \ell \end{cases}$$

Denote $\bar{s} = INIT(s, i)$, $\bar{s}_k = INIT(s_k, i_k)$. It is now readily seen that (2.1) is equivalent to the following easily checked condition:

$$(s_k \leq s \vee s \leq s_k \vee (\bar{s} \leq s_k \wedge \bar{s}_k \leq s)) \wedge ([i, j] \cap [i_k, j_k] \neq \emptyset) \qquad (2.2)$$

Here $[i, j]$ denotes the interval of integers ℓ such that $i \leq \ell \leq j$.

A condition of the form (2.2) has to be tested for at most every term in T and every term in V. Hence

Theorem 2.1 *We can test whether $V \Rightarrow T$ holds in time $length(V) \cdot length(T)$.*

Let us indicate how to improve the algorithm. It follows from (2.2) that for $s_k[i_k \textbf{ downto } j_k] \Rightarrow s[i \textbf{ downto } j]$ to hold, $\bar{s}_k = INIT(s_k, i_k)$ and $\bar{s} = INIT(s, i)$ must be related. By precomputing, we store the privilege V in an array

$$\left[\langle \bar{s}_1, s_1[i_1 \textbf{ downto } j_1]\rangle, \ldots, \langle \bar{s}_m, s_m[i_m \textbf{ downto } j_m]\rangle\right]$$

so that for $i < j$,($\bar{s}_i$ LEX $\bar{s}_j$), where LEX is a binary relation specifying the lexicographic ordering on strings (securons). To test whether $V \Rightarrow s[i \textbf{ downto } j]$ we use binary search, i.e., we check whether ($\bar{s}_k$ LEX $\bar{s}$) for $k = \lceil m/2 \rceil$. Assume this holds, then we check whether $\bar{s}_k \leq \bar{s}$. If the latter does not hold, then $\bar{s}$ is not related to any $\bar{s}_\ell$, $1 \leq \ell \leq \lceil m/2 \rceil$, and the problem has been reduced by half. If $\bar{s}_k \leq \bar{s}$ does hold, then we check (2.2), and if that fails we continue binary search on both $[1, \lceil m/2 \rceil - 1]$ and $[\lceil m/2 \rceil + 1, m]$.

In practice, the combination of this binary search on V with the fact that the recursion on the structure of T usually does *not* lead to testing $V \Rightarrow t$ for every term t in T, results in running time about linear in $\log(length(V))$ and sublinear in $length(T)$. In actual experiments, the algorithm is very fast.

2.5 Negative Privileges and Protections

As formulated thus far, the power of privileges is monotonic, i.e., if V_1 and V_2 are privilege expressions and $v_{\text{priv}}(V_1) \subseteq v_{\text{priv}}(V_2)$ then whenever $V_1 \Rightarrow T$ it is also the case that $V_2 \Rightarrow T$. The planners of a secure file system may wish to make, in certain instances, access to some file $\mathcal{F}_1$ to be incompatible with access to file $\mathcal{F}_2$ because a user who possesses the combined information in $\mathcal{F}_1$ and $\mathcal{F}_2$ will have dangerous power.

To enable planners to implement such policies, and for other potential applications, we introduce the constructs of negative protections and negative components of privileges. The total effect will be to make the power of privileges *non-monotonic.*

Definition 2.9 Protections *have the structure* $T = \langle T_1, T_2 \rangle$ *where* T_1 *and* T_2 *are protection expressions. The first and second components of* T *are called, respectively, the* positive *and the* negative *protections of the file.*

Similarly, privileges will have the structure $V = \langle V_1, V_2 \rangle$*.*

A privilege $V = \langle V_1, V_2 \rangle$ satisfies $T = \langle T_1, T_2 \rangle$*, again written as* $V \Rightarrow T$*, if*

$$V \Rightarrow T_1 \text{ and } V \not\Rightarrow T_2.$$

When talking about a privilege $V = \langle V_1, V_2 \rangle$, we shall often use $V^+ = V_1$ and $V^- = V_2$ to denote the positive and negative components of V, and similarly for protections.

By way of illustration, if in the above example $\mathcal{F}_i$ was protected in the old sense by expression T_i, $i = 1, 2$, then protecting F_i by $\langle T_i, T_1 \wedge T_2 \rangle$, and using privileges of the form $\langle V, V \rangle$, will exactly enforce the desired discipline.

The second component of T can be left empty and then, by convention, $V \Rightarrow T$ if $V^+ \Rightarrow T^+$.

2.6 Indelible Protections and Confinement

Two additional enhancements of the privilege and protection structures are needed for addressing problems arising in file system security arrangements.

One of the modes of access to a file is the *Change Protection* mode which enables the user to change one or more of the file's protections. Since changing protections of a file may have far reaching and sometimes unforseen consequences, we want to have a mechanism for limiting the change of protection even when allowed.

To this end we introduce the notion of *indelible* security expression $!E$, where E is an expression. The intention is that the indelible component of the protection on a file cannot, as a rule, be changed by a process even if the process has Change Protection rights with respect to that file. It will, however, be seen later that a "bypass" exception to this preservation of indelible protections rule is needed.

Since the intention is to provide a floor below which a protection cannot be lowered, it is readily seen that incorporation of indelible components in a protection $\langle T^+, T^- \rangle$ should be in the manner

$$T = \langle T_1 \wedge !E, T_2 \vee !G \rangle. \tag{2.3}$$

In this way, any modification of T^+ leaves at *least* $!E$, and any modification of of T^- leaves at *most* $!G$.

The concept of indelible protections is closely related to the issue of *confinement* of information. [Lampson 73] To illustrate confinement, assume that a file $\mathcal{F}$ has the Read protection $\langle !E, !G \rangle$, i.e., no mutable component is present. Let another file $\bar{\mathcal{F}}$ have the Read protection $\langle !\bar{E}, !\bar{G} \rangle$. Assume that some process $\mathcal{P}$ can read $\mathcal{F}$ and can *write* into $\bar{\mathcal{F}}$. The process $\mathcal{P}$ may then copy information from $\mathcal{F}$ into $\bar{\mathcal{F}}$. As matters now stand, there may be another process $\mathcal{P}_1$ which cannot read $\mathcal{F}$ but can read $\bar{\mathcal{F}}$. This process $\mathcal{P}_1$ may gain access to information in $\mathcal{F}$ via its copy placed in $\bar{\mathcal{F}}$ by $\mathcal{P}$. Our intention in providing indelible protections was to avoid unforseen declassification of information in a file through change of protection. It is

reasonable to assume that for files guarded by indelible protections, we would also want to avoid inadvertent leakage of information in the manner described just now. To this end we introduce the concept of *confinement* and a mechanism to enforce it.

Define $READS(\mathcal{F})$ by

$$READS(\mathcal{F}) = \{\mathcal{P} \mid \mathcal{P} \text{ can read } \mathcal{F} \text{ based on indelible privileges/protections}\}.$$

Similarly, define $WRITES(\mathcal{F})$ by

$$WRITES(\mathcal{F}) = \{\mathcal{P} \mid \mathcal{P} \text{ can write } \mathcal{F} \text{ based on indelible privileges/protections}\}.$$

Confinement can now be formally be defined as: For any process $\mathcal{P}$ and files $\mathcal{F}, \bar{\mathcal{F}}$

$$\mathcal{P} \in READS(\mathcal{F}) \wedge \mathcal{P} \in WRITES(\bar{\mathcal{F}}) \rightarrow READS(\bar{\mathcal{F}}) \subseteq READS(\mathcal{F}). \qquad (2.4)$$

Let us emphasize that only the indelible portions of the protections of $\mathcal{F}$ and $\bar{\mathcal{F}}$ play a role in (2.4).

The above discussions lead to natural extensions of our previous privilege and protection structures. Our positive and negative protections will (optionally) have indelible components so that a typical protection with have the form

$$T = \langle T_1 \wedge !E, T_2 \vee !G \rangle \qquad (2.5)$$

where T_1, T_2, E, and G are any protection extensions. By way of abbreviation, we shall use the notation $T = \langle T^+, T^- \rangle$ to describe (2.5) so that $T^+ = T_1 \wedge !E_1$ and $T_2 \vee !G$. We shall call T_1 and $!E$, respectively, the *mutable* and *indelible* components of T^+, and similarly for T^-.

The semantics for $v_{\text{prot}}(!H)$ will depend on the context, i.e. on whether $!H$ is part of T^+ or T^-. We according introduce the notations v^+_{prot} and v^-_{prot} to mark this distinction.

To help simplify the specification of $v^+_{\text{prot}}(!E)$ and $v^-_{\text{prot}}(!G)$, we extend the notion $SET(t)$ introduced in Definition 2.4. for terms to the case of general security expressions H. Recall that $v_{\text{prot}}(H) = \{U_1, \ldots, U_k\}$, $U_i \subseteq \mathbf{str}$.

Definition 2.10 *For any security expression H define*

$$SET(H) = \bigcup_{U \in v_{\text{prot}}(H)} U$$

Thus $SET(H)$ consists of all the securons "appearing" in $v_{\text{prot}}(H)$. It follows that for a *privilege* expression E, $v_{\text{priv}}(E) = SET(E)$.

Definition 2.11 *Let H be an expression. Define*

$$v^+_{\text{prot}}(!H) = \{SET(H)\},$$

$$v^-_{\text{prot}}(!H) = \{\{s\} \mid s \in SET(H)\},$$

$$v^+_{\text{priv}}(!H) = v_{\text{priv}}(!H) = SET(H).$$

The intention is that in $\langle T_1 \wedge !E, T_2 \vee !G \rangle$, $!E$ will give maximal positive protection and $!G$ will give maximal negative protection. Accordingly we defined $v^+_{\text{prot}}(!E)$ to be *all* of $SET(E)$ and $v^-_{\text{prot}}(!G)$ to be it any of $\{s\}$, $s \in SET(G)$.

Note that $v^+_{\text{prot}}(!E)$ always has the form $\{\{s_1, s_2, \ldots\}\}$ and $v^-_{\text{prot}}(!G)$ always has the form $\{\{s_1\}, \{s_2\}, \ldots\}$, where $s_1, s_2, \ldots$ are securons. For privileges, $v^+_{\text{priv}}(!E)$ and $v^-_{\text{priv}}(!G)$ are, as before, sets $\{s_1, s_2, \ldots\}$ of securons.

We now turn to the specification of privileges incorporating indelible expressions. Privileges for non-modifying actions (i.e. Read, Execute, and Detect) will have components whose functions will be to enforce confinement.

Definition 2.12 Privileges *for non-modifying modes of access have the form*

$$V = \langle V_1, V_2, M_1, M_2 \rangle \tag{2.6}$$

where V_1 and V_2 are privilege expressions, and M_1 and M_2 are any expressions. In (2.6), V_1 and V_2 are positive negative components of V, and M_1 and M_2 are called the positive and negative (indelible) mediating components of V.

We must also extend the notion of satisfaction for security expressions to the case where indelible expressions are included.

Definition 2.13 *Let V be a privilege expression and $T^+ = T_1 \wedge !E$, $T^- = T_2 \wedge !G$ be protection expressions. Define*

$$V \Rightarrow T^+ \ \text{ if } \ V_1 \Rightarrow T_1 \ \text{ and } \ v_{\text{priv}}(V) \Rightarrow v^+_{\text{prot}}(E), \tag{2.7}$$

$$V \Rightarrow T^- \ \text{ if } \ V_1 \Rightarrow T_2 \ \text{ or } \ v_{\text{priv}}(V) \Rightarrow v^-_{\text{prot}}(G), \tag{2.8}$$

In (2.7) and (2.8), the satisfaction relation on the right hand side is that of Definition 2.2, and its extension to expressions, explained after Definition 2.8.

Definition 2.14 *Let V_{rd}, with notation as in (2.6), and $T_{\text{rd}} = \langle T_1 \wedge !E, T_2 \vee !G \rangle$ be, respectively, a Read privilege and a Read protection. We shall say that V satisfies T ($V \Rightarrow T$) if*

1. $V_1 \Rightarrow T^+$,
2. $V_2 \not\Rightarrow T^-$,
3. $SET(E) \subseteq SET(M_1)$.
4. $SET(G) \subseteq SET(M_2)$.

The notation $V \Rightarrow T$ is similarly defined for the other non-modifying access privileges.

Definition 2.15 *Let $T = \langle T_1 \wedge !E, T_2 \vee !E \rangle$ and $\bar{T} = \langle \bar{T}_1 \wedge !\bar{G}, \bar{T}_2 \vee !\bar{G} \rangle$, we shall say that $\bar{T}$* indelibly dominates *T if*

$$SET(E) \subseteq SET(\bar{E}), \quad SET(G) \subseteq (\bar{G}).$$

Lemma 2.1 *If $T_{\text{rd}} = \langle T_1 \wedge !E, T_2 \vee !G \rangle$ and $\bar{T}_{\text{rd}} = \langle \bar{T}_1 \wedge !\bar{E}, \bar{T}_2 \vee !\bar{G} \rangle$ are Read protections for files $\mathcal{F}$ and $\bar{\mathcal{F}}$ respectively and $\bar{T}_{\text{rd}}$ indelibly dominates T_{rd}, then it is the case that $READS(\bar{\mathcal{F}}) \subseteq READS(\mathcal{F})$.*

Proof: By definition, only the indelible components of privileges play a role in determining $READS(\mathcal{F})$ and $READS(\bar{\mathcal{F}})$. Let $\mathcal{P}$ be a process with (indelible) Read privilege V_{rd} as in (2.6), such that $\mathcal{P} \in READS(\bar{\mathcal{F}})$. Then, by definition,

$$v_{\mathrm{priv}}(V_1) \Rightarrow v^{+}_{\mathrm{prot}}(\bar{E}), \tag{2.9}$$

$$v_{\mathrm{priv}}(V_2) \not\Rightarrow v^{+}_{\mathrm{prot}}(\bar{G}), \tag{2.10}$$

$$SET(\bar{E}) \subseteq SET(M_1)\,, \quad SET(\bar{G}) \subseteq SET(M_1). \tag{2.11}$$

Relation 2.9 is equivalent to $SET(\bar{E}) \subseteq SET(V_1)$ and relation 2.10 is equivalent to $SET(V_2) \cap SET(\bar{G}) = \emptyset$.

To say that $\bar{T}_{\mathrm{rd}}$ indelibly dominates T_{rd} means that $SET(E) \subseteq SET(\bar{E})$ and $SET(G) \subseteq SET(\bar{G})$. Hence the previous paragraph implies $SET(E) \subseteq SET(V_1)$ and $SET(V_2) \cap SET(G) = \emptyset$. Also, $SET(E) \subseteq SET(M_1)$ and $SET(G) \subseteq SET(M_2)$. Thus $V_{\mathrm{rd}} \Rightarrow T_{\mathrm{rd}}$ and $\mathcal{P} \in READS(\mathcal{F})$. □

Enforcing confinement means that if some process $\mathcal{P}$ reads $\mathcal{F}$ and can write into $\bar{\mathcal{F}}$, then $READS(\bar{\mathcal{F}}) \subseteq READS(\mathcal{F})$ should hold. This is insured by the following specification:

Definition 2.16 *Let a process $\mathcal{P}$ have the Read and Write privileges V as in (2.6) and V_{wr}; and a file $\bar{\mathcal{F}}$ have the Read and Write protections $\bar{T}_{\mathrm{rd}} = \langle \bar{T}_1 \wedge !\bar{E}, \bar{T}_2 \vee !\bar{G} \rangle$ and $\bar{T}_{\mathrm{wr}}$. The process $\mathcal{P}$ will be permitted to* write *in F if and only if*

1. $V_{\mathrm{wr}} \Rightarrow \bar{V}_{\mathrm{wr}}$,
2. $SET(M_1) \subseteq SET(\bar{E})$
3. $SET(M_2) \subseteq SET(\bar{G})$

Theorem 2.2 *If for files $\mathcal{F}, \bar{\mathcal{F}}$ some process $\mathcal{P}$ can read $\mathcal{F}$ and write into $\bar{\mathcal{F}}$ then $READS(\bar{\mathcal{F}}) \subseteq READS(\mathcal{F})$. Thus confinement is enforced.*

Proof: Let $\mathcal{P}$ have privileges $V_{\rm rd}$ and $V_{\rm wr}$ as in (2.6), and let $\mathcal{F}$ and $\bar{\mathcal{F}}$ have protections $T_{\rm rd} = \langle T_1 \wedge !E, T_2 \vee !G\rangle$, $\bar{T}_{\rm rd} = \langle \bar{T}_1 \wedge !\bar{E}, \bar{T}_2 \vee !\bar{G}\rangle$, respectively. The conditions of the theorem and definitions 2.14 and 2.16 imply

$$SET(E) \subseteq SET(M_1) \subseteq SET(\bar{E}),$$

$$SET(G) \subseteq SET(M_2) \subseteq SET(\bar{G}).$$

Hence $\bar{\mathcal{F}}$ indelibly dominates $\mathcal{F}$, which by lemma 2.1 entails $READS(\bar{\mathcal{F}}) \subseteq READS(\mathcal{F})$. □

2.7 Bypass Privileges

The introduction of indelible protections serves to enhance security by prohibiting inadvertent change of protections and by enforcing confinement. The intention in having indelible expressions is that they are unmodifiable even by a process does have powerful Change Protection privileges $V_{\rm cp}$. This strict interpretation of indelible protection creates practical difficulties. Clearly some carefully controlled processes must have the ability to change even indelible protections if such protections turn out to be too strict or if, later on, there arises the possibility of "declassifying" a file. Also, strict enforcement of confinement causes information to flow only upward in terms of security. Pragmatic needs require exceptions to this rule. The possibilities are realized through augmentation of the Change Protection Privilege.

Definition 2.17 *The* Change Protection privilege *has the structure*

$$V_{\rm cp} = \langle V^+, V^-, B\rangle$$

where B is called the bypass *component. Let $\mathcal{F}$ be a file with Change Protection protection $T_{\rm cp} = \langle T^+, T^-\rangle$ and $T_X = \langle T_1 \wedge !E, T_2 \vee !G\rangle$ be any of $\mathcal{F}$'s protections ($X =$ cp included). If $V \Rightarrow T^+$ and $V \not\Rightarrow T^-$ then the privilege $V_{\rm cp}$ is sufficient for*

changing the mutable components T_1 *and* T_2 *in* T_X *to any other mutable expressions. If, in addition,*

$$SET(E) \subseteq SET(B) \text{ and } SET(G) \subset SET(B) \tag{2.12}$$

then the indelible components $!E$ *and* $!G$ *can be changed as well.*

Assume that a process $\mathcal{P}$ has Change Protection privilege V_{cp}, and a file $\mathcal{F}$ has Change Protection protection T_{cp} and another protection, say T_{rd}, with notations as above. If V_{cp} satisfies T_{cp} and (2.12) holds for T_{rd}, then $\mathcal{P}$ could read $\mathcal{F}$ by first changing T_{rd} to $\langle\{\emptyset\}, \emptyset\rangle$. Later $\mathcal{P}$ can restore T_{rd} to its original value $\langle T^+, T^-\rangle$. This motivates the following rule: *Under the above conditions,* $\mathcal{P}$ *can read* $\mathcal{F}$*; similarly, any other access* X *to* $\mathcal{F}$ *is permitted.*

2.8 Gradations of Actions on Files

Until now we made no assumption about the relative strengths of protections on a file $\mathcal{F}$. If we consider the effect of various modes of access to a file, we see that it does not make sense to have a weak Change Protection protection T_{cp} coupled with a strong Read protection T_{rd}. For a file protected in this way may be read by first changing the Read protection T_{rd} and then reading the file. Thus it makes to partially order the modes of access to a file and require that protections be correspondingly ordered.

Similar considerations apply to privileges associated with processes. Here the privilege for the more powerful action should be weaker. A reasonable, but by no means unique, gradation of modes of access to files is as follows. Changing protection of a file is the farthest reaching action, for by doing this all other modes of access can become available. Detecting the existence of a file $\mathcal{F}$ in a directory $\mathcal{D}$ is the minimal mode of access. For if a process, for example, writes into $\mathcal{F}$ it presumable knows of the existence of $\mathcal{F}$. Reading and executing a file are about

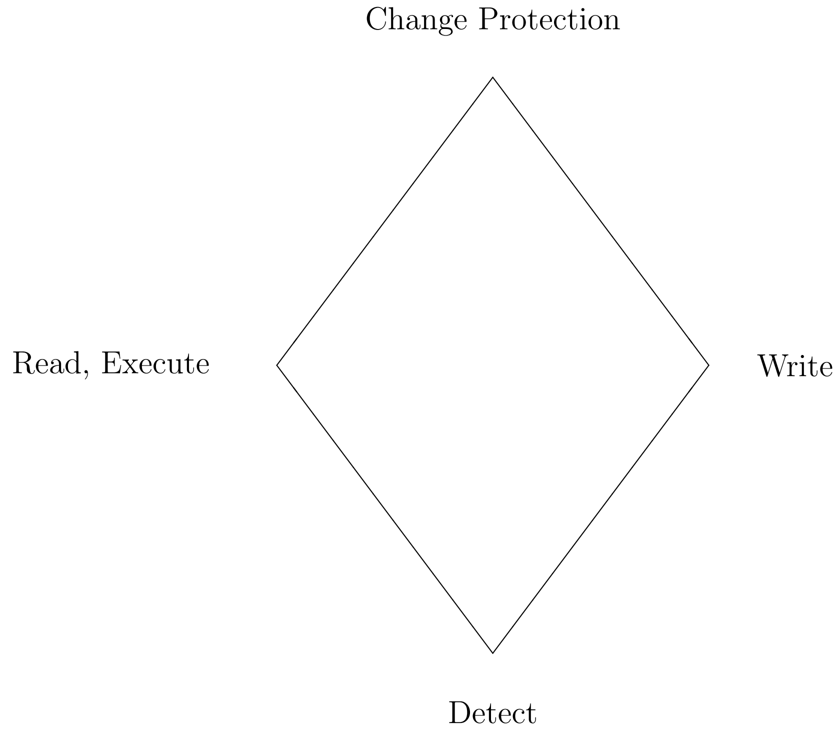

Figure 2.1: The hierarchy of access modes

equally powerful actions. For if a process $\mathcal{P}$ can read an executable file $\mathcal{F}$ then it can copy it to an executable file and subsequently execute it. Conversely, if $\mathcal{P}$ can execute $\mathcal{F}$ then by appropriate tracing the content of $\mathcal{F}$ can be fairly accurately reconstructed. Finally, there is no clear dominance between reading or writing into a file. These considerations lead to the semi-ordering of strengths of actions depicted in Figure 2.8.

As explained before, protections on a file $\mathcal{F}$ should reflect this ordering. This does *not* mean, of course, that across different files $\mathcal{F}$ and $\bar{\mathcal{F}}$, the Change Protection protection $\bar{T}_{\text{cp}}$ for $\bar{\mathcal{F}}$ should be stronger than, say, the Read protection T_{rd} for $\mathcal{F}$.

A convenient way for implementation order among protections is by syntactic means. We shall say that $\bar{T} = \langle \bar{T}_1 \wedge !\bar{E}, \bar{T}_2 \vee !\bar{G} \rangle$ *dominates* $T = \langle T_1 \wedge !E, T_2 \vee !G \rangle$ if

for some expressions H_1, H_2,

$$\bar{T}_1 = T_1 \wedge H_1, \quad \bar{T}_2 = T_2 \vee H_2,$$

and, furthermore, $\bar{T}$ indelibly dominates T, i.e.

$$SET(E) \subseteq SET(\bar{E}), \quad SET(G) \subseteq SET(\bar{G}).$$

Chapter 3

Sentinels

3.1 Overview

The mechanisms described in the previous chapter form a *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. For example, if we wanted to audit $\mathcal{F}$, that is record the names of all users who accessed a secure 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, it could only be rarely used and with great difficulty. In this chapter we will extend the pure access scheme with *sentinels.* Sentinels allow us to add features such as auditing conveniently. These features can be customized to reflect any special needs an organization may have.

Here is how sentinel for a file $\mathcal{F}$ works: A sentinel S is a program listed in a $\mathcal{F}$'s protection header. When any process $\mathcal{P}$ opens $\mathcal{F}$, the operating system schedules S for execution as a process. S is passed some variables that allow it to perform various operations. The sentinel can perform arbitrary actions. For example, S could record the names of all users accessing $\mathcal{F}$.

Suppose we only wished to record when 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 our protection header to contain a *trigger* condition R. S is only scheduled for execution by the operating system kernel when $\mathcal{F}$ is accessed *and* the process meets the trigger condition R.

Even further efficiency can be derived by separating sentinels into different classes. For example, auditing simply requires that S make the necessary records in the audit file when $\mathcal{P}$ reads $\mathcal{F}$. A more sophisticated sentinel might decide to take action only when certain records are read. For example, certain records in a $\mathcal{F}$ might have a keyword `secret` attached to them. The sentinel S guarding $\mathcal{F}$ might 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 monitor and approve individual I/O operations between $\mathcal{P}$ and $\mathcal{F}$. Accordingly, we allow two types of sentinels: *asynchronous* sentinels which run separately from $\mathcal{P}$ and *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 $\mathcal{F}$. (Funnels are an extension of UNIX *named pipes.* [Ritchie-Thompson 74], [Kernighan-Plauger 76], [Leffler-Fabry-Joy 83])

A very sophisticated attack might try to read $\mathcal{F}$ protected by 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 can 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. But 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 an assigned privilege V. When the sentinels is scheduled by the operating system, it will run with privilege V.

3.2 Semantics of Sentinels

Definition 3.1 *A* sentinel S *is an ordered tuple* $\langle \mathcal{S}, t, V_S \rangle$ *where* $\mathcal{S}$ *is the name of a file,* V_S *is a privilege called the* assigned privilege *and* t *is the type,* 1*,* 2*, or* 3 *indicating that the* $\mathcal{S}$ *is*

1. Asynchronous*,*

2. Synchronous*,*

3. Lazarus*.*

Suppose a process $\mathcal{P}$ with privileges V_P attempts to access a file $\mathcal{F}$ with protections T and 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 $\mathcal{S}$ exists and is executable. If it is, $\mathcal{S}$ is created by the operating system with privileges V and executes according to type t. (See section 3.3 for a discussion of types of sentinels.)

To reduce unnecessary executions of 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 how we could express 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.

Recall in equation (2.3) we saw the most general form of a protection,

$$T = \langle T_1 \wedge !E, T_2 \vee !G \rangle. \tag{3.1}$$

Definition 3.2 *A* sentinel clause C *is an ordered pair of a* trigger R *and sentinel* S*. R is a protection of the form in (3.1) and S is a sentinel as defined in Definition 3.1. We write C as $R \rightarrow S$.*

Suppose a process $\mathcal{P}$ with privileges V_P tries to access a file $\mathcal{F}$ protected by a sentinel clause $R \rightarrow S$, where $S = \langle \mathcal{S}, t, V_S \rangle$. The operating system checks whether $V_P \Rightarrow R$. If it does, the operating system will run S.

We allow arbitrarily many sentinel clauses $C_1 = R_1 \rightarrow S_1$, $C_2 = R_2 \rightarrow S_2, \ldots$ to be attached to a protection. The operating system kernel will check each C_i, seeing whether $V_P \Rightarrow R_i$, and running all the S_i which match the trigger conditions.

Using this notation, we have a unified way of writing not just sentinel clauses, but also the pure access scheme. For example, if $\mathcal{F}$ has protection T and clauses $C_1, C_2, \ldots$ for access X we can write an *extended protection for access* X,

$$(T \rightarrow ACCESS_X) \wedge (R_1 \rightarrow S_1) \wedge \ldots \wedge (R_k \rightarrow S_k) \tag{3.2}$$

where $ACCESS_X$ means that access type X is permitted to the file.

There are three important exceptions to the above rules:

1. If $S = \langle \mathcal{S}, t, V \rangle$ is a sentinel, then $\mathcal{S}$ can be an arbitrary executable file. In particular, $\mathcal{S}$ itself may be protected for Execute access by a sentinel S'. If S is triggered, it could lead to a chain of sentinels triggering sentinels — or even an infinite loop of sentinel scheduling. To prohibit this, we specify that a sentinel can only be triggered by a process $\mathcal{P}$ which is *not* a sentinel.

2. If a funnel sentinel S guards a file $\mathcal{F}$ in its Change Protection protection, it can approve or disapprove each attempted action on $\mathcal{F}$. In particular, it could prevent *any* process from changing $\mathcal{F}$'s protection. Since the sentinel S is part of $\mathcal{F}$'s extended protection, S could never be removed. To prevent this, sentinels are not allowed to be attached to Change Protection protections.

3. We allow individual users to attach sentinels to files. A user might try to violate confinement by attaching sentinel $S = \langle \mathcal{S}, t, V_S \rangle$ to a confined file $\mathcal{F}$. $\mathcal{S}$ runs with privileges V_S as a sentinel. In particular, if V_S does not have an indelible (mediating) component $\mathcal{S}$ can copy data from the confined file $\mathcal{F}$ to an unconfined process $\mathcal{F}'$. In the pure access system, a process could not "leak" information because a process that was able to read $\mathcal{F}$ would not be able execute $\mathcal{S}$. To protect confinement, we require that process P accessing $\mathcal{F}$ protected by S must have privileges sufficient to execute $\mathcal{S}$.[1]

3.3 Types of Sentinels

As indicated above, there are three types of sentinels: asynchronous, funnels (also called synchronous), and Lazarus. This section discusses technical issues related to the different types of sentinels. When process $\mathcal{P}$ triggers sentinel $S = \langle \mathcal{S}, t, F \rangle$ by accessing $\mathcal{F}$, it is passed certain information through environment variables:

1. the name of $\mathcal{P}$,
2. $\mathcal{P}$'s privileges,
3. the name of $\mathcal{F}$,
4. $\mathcal{F}$'s protections,
5. parameters to the system call accessing the file,[2] and
6. whether access was permitted by the pure access scheme.

[1] Notice that a sentinel knows whether it is executed as a sentinel or as an ordinary program by checking whether its privileges are P_V or those of an ordinary user.

[2] ITOSS I/O calls include `read, write, seek, chmod, chprot, exec, unlink, close, open` or `stat`. However, the call starting a sentinel can not be `read, write,` or `seek` because these are only meaningful to open files.

If the S is synchronous its input stream is triggered whenever $\mathcal{P}$ attempts to access $\mathcal{F}$. The data on the input stream to $\mathcal{S}$ contains the parameters of the I/O call to $\mathcal{F}$. While $\mathcal{S}$ executes, the process $\mathcal{P}$ sleeps. $\mathcal{S}$ can perform arbitrary actions on the input parameters. $\mathcal{S}$ writes data to its output stream. That data forms the actual parameters used in the I/O called. If $\mathcal{S}$ writes a null field, the I/O call is refused. When the I/O call returns, the value is passed to $\mathcal{S}$. $\mathcal{S}$ can then alter the return value. $\mathcal{S}$ then sleeps while $\mathcal{P}$ continues. If several synchronous sentinels are triggered by one access, the I/O calls are passed to the sentinels in the same order they were triggered.

If S is Lazarus, a simple reliability mechanism is provided which protects sentinels from attacks which rely on crashing the system to avoid auditing. When S is triggered, a file system entry $\mathcal{E}$ is entered in non-volatile memory before access to $\mathcal{F}$ is permitted. When $\mathcal{S}$ terminates normally, $\mathcal{E}$ is removed. However, if the operating system should crash or $\mathcal{S}$ is manually terminated, the entry $\mathcal{E}$ remains. When the system reboots, the boot procedure can detect $\mathcal{E}$ and retrigger S.[3]

3.4 Examples

We envision that a library of sentinels would provide basic security for most uses. This library could be supplemented by a small number of specially written sentinels for more complicated situations.

Here are just a few examples of how sentinels can be used. Some computer terminals have an *identify sequence.* When the terminal is polled with a particular sequence of ASCII characters, it transmits the identify sequence. A *letter-bomb* is a piece of electronic mail containing character strings which first load a new identify

[3]While the current version of ITOSS has not explicitly implemented Lazarus sentinels, they are available in the current implementation by using a two-phase locking protocol [Eswaran *et al* 76] in a synchronous sentinel and a special directory `/lock`.

sequence into the terminal, and then requests the identify sequence back. Hence the sender of a letter-bomb can force any sequence of characters to be entered from a terminal. There are only two ways to protect against letter bombs in existing operating systems: either remove all terminals supporting identify sequences or put special code in the kernel (device driver) that prevents certain sequences of characters from being transmitted. (Every time a new type of terminal is added, new code must also be added to the system.) But sentinels yield a simple solution to the above dilemma: simply attach a funnel which checks for the dangers sequence to the device files corresponding to the terminals which are vulnerable.

In fact, we can see that all classical operating system access protection schemes can be implemented by using sentinels. For example, standard UNIX allows a program $\mathcal{F}$ to assume the rights of its owner including a set-UID (set User ID bit). ITOSS can easily emulate this by adding a dummy entry $\mathcal{F}'$ to the file system. $\mathcal{F}'$ has ordinary protection data in it and a sentinel clause $(\Lambda \rightarrow \langle \mathcal{F}, 2, V \rangle)$, where Λ denotes the empty string, V is the block of necessary privileges to run the set-UID program, and 2 denotes that the sentinel is synchronous.[4]

Because sentinels are a powerful and general mechanism, we believe that they can be used in other software engineering applications outside of security.

3.5 Previous Work

The nearest previous concept to sentinels are *daemons*. (Introduced in the MULTICS operating system [Daley-Dennis 68] and the THE operating system [Dijkstra 68].) Daemons are continually running processes which maintain operating system functions such as, printing subsystems which require management of queues of files

[4]In fact, in ITOSS, a clever implementation of this can be done with just one file system entry. Simply add the sentinel clause to $\mathcal{F}$, setting $\mathcal{F}$ to trigger itself as a sentinel! The restriction against sentinels triggering subsequent sentinels prevents an infinite loop.

to be printed. Sentinels provide a much finer degree of control that daemons.

Several operating systems, such as MULTICS ([Organick 72], [Schroeder-Saltzer 72],), Hydra ([Wulf-Levin-Harbison 81), and UNIX ([Ritchie-Thompson 74]) attempted to address the issue of allowing a special process to actively intervene in I/O operations. The approach these systems adopted was to require that the intermediate processes — which correspond to our sentinels — be directly executed rather than passively triggered. Files were guarded by a restricted protection, and the intermediate process was granted powerful rights when executed. In UNIX, for example, In UNIX, each program $\mathcal{X}$ can have a set-UID bit set to be true. If it is, then $\mathcal{X}$ will assume the rights of the owner of $\mathcal{X}$ when it is executed. For example, a protected database would have a set-UID front-end; when data was changed, it could only be written by the front-end since no other ordinary user could write any bits of the database. Unfortunately, the only appropriate choice for most set-UID identities is the special user `root`, which can read or write any file in the system. Whenever `root` owns a set-UID program, that program becomes a potential source of security errors; since the set-UID program has arbitrary power in the operating system, the entire security structure of UNIX must be reproduced in every set-UID program. (Berkeley UNIX 4.2 has dozens of set-UID programs and there several widely known ways to violate the security of UNIX by exploiting weaknesses in those programs. Several methods that are also common to AT&T UNIX system V are described in [Grampp-Morris 84].)

Chapter 4

Incarnations and Secure Committees

4.1 Overview

A delicate point of maintaining a secure operating system is the management of security. Security management has two aspects: users must correct protections and privileges assigned to their files and processes; and the system administrators must manage the integrity of system files protection, installation of sentinels, consistent interpretation of values in the securon tree, and creation and deletion of users. ITOSS provides two structures for conveniently managing information. For individual users, user identities are divided into *incarnations*, each of which provides a default set of privileges for the user's processes and automatic assignment of protections to the user's files. For system administration, ITOSS provides *secure committees*, a special incarnation allowing a group of users to form a special incarnation which can perform operations that can not be trusted to a single user. Hence each user may have several incarnation or, through committees, a single incarnation may reflect the coordinated actions of several users.

4.2 Incarnations

In real computer installations, users do many things. They read and write electronic mail, they share information with different sets of users, they run applications, they write programs and documents, they play games and read electronic bulletin boards, and they do personal work. For each class of separate tasks, users require differently configured security features.

Definition 4.1 *An* incarnation *is a list of privileges* $(V_{\text{rd}}, V_{\text{wr}}, V_{\text{ex}}, V_{\text{dt}}, V_{\text{cp}})$ *and a* automatic protection schema π*. (The structure of* π *is given in definition 4.2.)*

We allow each user $\mathcal{U}$ to have a number of incarnations $\mathcal{I}_1, \mathcal{I}_2, \ldots$. When $\mathcal{U}$ logs onto the computer he must pick an incarnation. He can change his incarnation at any time. (On display terminals supporting windows, the incarnation can be displayed at all times to remind the user of which mode he is in.) All processes created by the user will have the default privileges specified by the incarnation.

Using minimum privileges is an effective safeguard against some *Trojan Horse* attacks. A Trojan Horse is a seemingly innocuous program which has a hidden feature subverting security. When the program is executed by a user with powerful privileges, the Trojan Horse uses those powerful privileges to gain access to otherwise protected data. By assigning users an incarnation with few privileges which can be used with untrusted software, a system administrator can eliminate much of the risk associated with Trojan Horses.

Definition 4.2 *Let* **C** *be a set containing names of (common) programs and the keyword* **other***,* **Z** *be the set of integers, and* **H** *be the set protection headers. An automatic protection schema is a function* $\pi : (\mathbf{C}, \mathbf{Z}) \rightarrow \mathbf{H}$.

Suppose $\mathcal{U}$ is in incarnation $\mathcal{I}$ with automatic protection schema π. When $\mathcal{U}$ creates a new file $\mathcal{F}$ with application program $\mathcal{A}$, the protection assigner checks whether $\mathcal{A} \in \mathbf{C}$. If it is not, the keyword **other** is used as the first argument to π.

$\mathcal{A}$ can pass a suggested protection value as an integer through the second argument in $\mathcal{Z}$. (In ITOSS, this is done by using the standard UNIX "chmod values.") $\mathcal{F}$ is given the protection header calculated from π.

Similar mechanism provide an easy to use high level "change protection" facility; the users chooses a suggested incarnation (determining π) and picks a protection from a menu (determining the second argument to π).

The specification and implementation of powerful automatic protection schemas is an area of active research.

4.3 Secure Committees

In many organizations actions with major consequences are made by groups of people. For example, payments of large amounts of money may require co-signers. This eliminates risk arising from the corruptibility of a single person.

We wish to emulate this feature for system management. A define a *committee* to be an incarnation requiring q people out of a set of users $\{\mathcal{U}_1, \ldots, \mathcal{U}_m\}$ to approve all actions. A *committee session* is an active meeting of q committee members, who are called the *participants*.

During a committee session, a proposed command will be presented to all of the participants. The operation will be performed only if each of the participants explicitly approves it. Hence each committee member has veto power when he participates in a session.

To implement secure committees, we need to use a secret sharing algorithm due to Adi Shamir. [Shamir 79] The algorithm give a protocol for dividing a text t into m encrypted pieces such that q pieces determine the value of t but $q-1$ pieces give *no* information about the value of t.

We do all computations with integer residues modulo a prime p. Let

$$h(x) = r_1 x^{q-1} + r_2 x^{q-2} + \cdots + r_{q-1} x + t$$

where the r_i are independent random integer residues. The pieces of the secret are the values $h(1), \ldots, h(m)$. The secret is $t = h(0)$. Since q values of h determine a nonsingular linear system of q equations in q variables, Lagrangian interpolation will allow us to recover the values of the coefficients of h and hence t. On the other hand, since our operation is over a finite field, $q - 1$ values of h do not determine any information about the value of t.

For implementing secure committees, we let t be the password to the secure committee incarnation. A sentinel called the *dealer box* picks a random h function and securely distributes the pieces $h(1), \ldots, h(m)$ of the secret to the individual committee members $\mathcal{U}_1, \ldots, \mathcal{U}_m$. (Secure distribution may use file system protection primitives or may depend on private key encryption.)

When a quorum of q users wish to form a secure committee, they pass their secrets to the dealer box which assembles the pieces of the secret and passes the value t to the password checking program. If the password checker accepts t, a secure committee is formed. Meanwhile, the dealer box picks a new random password t' and random function h'. After registering the password t' with the password checker, the dealer box securely distributes the new pieces of the secret $h'(1), \ldots, h'(m)$ to the committee members. (If the system should crash prior to completing distribution, the password checker will still accept t.) This keeps an opponent from slowly gather pieces of t and bing able to form a secure committee by himself.

The secure committee concept can be extended in several straightforward ways. The protocol can support "weighted quorums" where different number of members are required to form a secure committee depending on rank. For example, we can give junior members one secret each and senior members two secrets. Our protocol is also very appropriate for distributed implementations of secure operating systems.

If there are operations which do not require the full strength of the committee but are still sensitive, the committee can delegate these tasks to *subcommittees* by using sentinels. In the degenerate case, some simple sensitive tasks may be

entrusted to a single user — a machine operator, for example.

Chapter 5

Fences

5.1 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 in that many cases we are trying to prohibit some event from occurring. In this chapter, 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.

5.2 Fingerprints

In the course of his research on string matching, Michael Rabin 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 funnel 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 funnel sentinel guarding $\mathcal{F}$ keep an independent fingerprint 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.

5.3 System Call Fingerprints

A wide class of existing bugs in Berkeley UNIX 4.2 are based on race conditions. In UNIX, system calls such as "read file", "write file", and "change protection" are not atomic but concurrent. Because of the way the UNIX kernel is structured, it is very difficult to detect all possible race conditions.

One race condition exists between the `link` system call and the `chmod` system call. `chmod` changes the protection on a file $\mathcal{F}$. In UNIX 4.2, security is enforced by only allowing the owner of $\mathcal{F}$ or the system-user `root` to access $\mathcal{F}$, so it first checks ownership information and then changes the protection. `link` makes a new file system entry $\mathcal{F}'$ and then sets it to point to $\mathcal{F}$. Since a `link` call merely establishes a link, no special security rights are required to execute it.

By running the two system calls simultaneously, it is possible for an opponent to gain access to a file he does not own. Let $\mathcal{F}$ be owned by $\mathcal{U}$. An opponent $\mathcal{U}'$ might gain access to $\mathcal{F}$ by executing these two system calls simultaneously:

1. `link(`$\mathcal{F}'$`,`$\mathcal{F}$`)`. Link file $\mathcal{F}$ to $\mathcal{F}'$.

2. `chmod(0666,`$\mathcal{F}'$`)`. Make $\mathcal{F}'$ publicly available for reading and writing.

If these system calls are executed, the following chain of events sometimes will occur:

1. `link` creates a dummy entry $\mathcal{F}'$ owned (temporarily) by $\mathcal{U}'$.

2. `chmod` checks $\mathcal{F}'$ to see whether $\mathcal{U}'$ is allowed to change its protection. Since $\mathcal{U}'$ temporarily owns $\mathcal{F}'$, `chmod` approves the action.

3. `link` completes the pointer from $\mathcal{F}'$ to $\mathcal{F}$. At this point $\mathcal{U}'$ can not access $\mathcal{F}$ or $\mathcal{F}'$ since those files are owned by $\mathcal{U}$.

4. `chmod`, having already approved rights to change the protections to $\mathcal{F}'$, goes ahead and makes that file publicly readable and writeable. Since $\mathcal{F}'$ is now a pointer to $\mathcal{F}$, this makes $\mathcal{F}$ publicly readable and writeable, and thus $\mathcal{U}'$ can access that file.

We found this bug and several other bugs in UNIX 4.2 by inserting fences in the operating system which fingerprint system call requests at the top level of the kernel and at the driver level. The fingerprints contain the text of the request and the security data (process privilege information and file protection information) associated with the processes and files. If the fingerprints do not match, the fence determines that race conditions must exist in the kernel and halts the processor. In the above example, the protection information associated with $\mathcal{F}'$ changes from the top level of the kernel and the driver level and hence the fingerprints are different.

While our current library of fences is not yet sufficient to validate a secure system by itself, we believe that the technique can be used in conjunction with more traditional validation methods to provide a very high degree of confidence in security software.

Chapter 6

A Prototype Implementation

Approximately one man-year was spent converting 4.2 UNIX running on a SUN-2 processor into a prototype version of ITOSS. This chapter describes the results of some performance experiments, some hints for the potential users of the system, and some fine points of the implementation.

This section discuss details drawn from UNIX internals and presumes a familiarity with UNIX at the level of [Berkeley 84].

6.1 Performance

Performance was measured on a fragmented file system, that is one which had blocks distributed as would be typical after several hours of intensive use since the last reformatting operation. 88% of the blocks on a 4.2 BSD disk were used. Normally, disks are only formatted once, so fragmented file systems are the normal state of affairs. [McKusick *et al* 83] indicate that this file system configuration with approximately 39% overhead above that of an empty file system.

We tested the performance of five system calls: `open`, `close`, `read`, `write`, and `chdir`. Only operations that succeeded were counted for performance purposes. [Ousterhout *et al* 85] shows that these five system calls account for 78% of all

system calls under UNIX. Moreover, the I/O structure of these system calls reflect typical uses of ITOSS system.

Using the UNIX emulation mode described below, ITOSS files were assigned naturally occurring privileges. The ITOSS process taking measurements was given a fixed privilege involving two positive terms and one negative term. Parameters to `open` attempted to open a file for both reading and writing. Parameters to `write` attempted to write 1K bytes into a file. Parameters to `read` attempted to read 1K bytes from a file.

Files were chosen from randomly chosen top level user directories. All files were ordinary block files. The experiment was repeated three times, once under standard UNIX, once under ITOSS without fingerprinting, and once under ITOSS with fingerprinting. Each experiment examined 1,000 files.

Here are the experimental results. Percentages compare relative time compared with standard UNIX.

	Standard UNIX	ITOSS w/o fences	ITOSS with fences
`open`	100%	104%	104%
`close`	100%	99%	100%
`read`	100%	100%	108%
`write`	100%	101%	109%
`chdir`	100%	104%	104%

To summarize, ITOSS introduces less than 10% performance degradation compared with standard UNIX. (5% performance degradation if fingerprinting is not used.)

6.2 How to Initially Configure ITOSS

We envision ITOSS being shipped with an initial default security configuration and an initial secure committee. The ITOSS tape would come with a full set of initial

passwords which could then be used to immediately use ITOSS. A basic set of general purpose sentinels would also be provided for the purposes of bringing up the system. The secure committee, with the security engineers, could then configure ITOSS to the exact structure demanded by the organization using it.

6.3 Some Implementation Details

6.3.1 Peculiarities

Some UNIX peculiarities forced us to alter the current model described in Chapters 2 through 5.

Detection access: There is no general access right for file detection under UNIX. By convention, all files are stored in a directory. Knowledge of the file's existence, by convention, is contingent on read access to the directory containing the file. However, it is also possible to attempt to discover the existence of a file by overwriting it, so all access rights to a directory must be turned off to make detection impossible. For this reason we adopted the following rule: *Restriction of detection to a file is only meaningful when all rights to the directory containing the file are similarly restricted.*

Process structure: In UNIX, a process can fork off arbitrary other subprocesses. Typically, the correct operation to perform in this context is to give the subprocesses identical privileges to the original process. Unfortunately, it is frequently necessary to further restrict the privileges of the subprocess. For this reason, a subprocess may have positive privilege set equal or strictly less strong than that of the original process. Similarly, the negative privileges of the process must be equal to or strictly stronger than that of the parent process. Privilege change is controlled via the new `chpriv` system call. The process table in memory is not accessible except through special programs having read rights on `/etc/kmem`. Signaling follows standard UNIX conventions.

Resource control: No attempt is made to protect covert resource channels such as free blocks or free inodes. To run under a fully secure environment, either substantial excess inodes and free blocks should be provided, or the file system should be partitioned finely to correspond to confined regions.

6.3.2 File System Alterations

Additional space is needed to store the protection information associated with files. This information is stored in a 1K block header to each header occurring directly after the indirection block header. Note that this relieves the necessity to inspect the inode table in most cases, and in any case allows the inode table to shrink in size, causing non-trivial performance improvements over standard UNIX. However, since this information may be needed for the compatibility library described below, the inode table has been kept intact.

6.3.3 Internet

All Internet support has been turned off. All new IPC mechanisms, that is, all interprocess communication except pipes, has been disabled.

6.3.4 Hooks for Fingerprints

At the system call level, prior to updating the active file table, all I/O calls make a kernel call is made to the procedure `marktop` which can provide arbitrary fingerprinting. A similar call to `markbot` is provided at the virtual file system block allocation level. Performance could be improved, at the loss of easy control, by making the mark procedures into macros with explicit parameters.

6.3.5 System Calls

The following system calls have been added to ITOSS: `chprot` allows a process to change the protection of a file to which the process has change protection access. `protstat` retrieves the header of a file. `chpriv` changes the privilege of the active process.

6.3.6 Compatibility Mode

The following system calls are implemented via a compatibility mode call which emulates the behavior of standard UNIX sytem calls: `access`, `chmod`, `chown`, `creat`, `getgid`, `getegid`, `getgroups`, `getuid`, `geteuid`, `mkdir`, `mknod`, `open`, `setgroups`, `setpgrp`, `setregid`, `setreuid`, `stat`.

Chapter 7

Open Problems

This thesis is a snapshot of the current state of the ITOSS implementation and security toolkit project. This work leaves many issues open for future exploration. Because this is a list of potential future work, many of these items are quite speculative.

The modification of UNIX utility programs to match ITOSS data structures: Currently ITOSS supports UNIX utilities via a compatibility library. While this provides a quick way of including UNIX utilities in ITOSS, it also means that those UNIX utilities are limited to security features provides in standard UNIX. In the near future it is important to port over a central core of UNIX utilities to use full ITOSS features.

Extension of ITOSS to a distributed system: ITOSS currently runs on a uniprocessor. Because of the decentralized nature of its security data, the conversion of ITOSS to a distributed system is not an unreasonable project to consider. Distributed ITOSS could shed considerable light on the nature of the hierarchical division of security. Moreover, by using redundancy available in the system, it may be possible to preserve file system integrity even if physical security is not complete. It may be possible to preserve the file system even if an adversary tampers with one processor.

Integration of ITOSS with existing network secure systems: Private-key cryptography provides a powerful but coarse method for protecting network security. What aspects of the finer security given in ITOSS can be transferred under network security systems?

Simulation of heterogeneous systems: ITOSS has features allowing it to simulate all conventional security structures. Moreover the committee/subcommittee structure allows us to run separate security regimes on a single file system. This allows us to simulate heterogeneous secure operating systems connected by a simple file transfer protocol or common file system. The implied connection between the heterogeneous elements will be determined by the form selected for embedding the separate security structures in ITOSS. Eventually, such an approach may yield a general method for representing translation functions of security parameters in arbitrary heterogeneous systems.

Graphical human interfaces: The securon tree we use is a graphical object, and many of the logical operators we use can be conveniently described pictorially. This suggests that we can help the security engineer by providing graphical descriptions of particular ITOSS security structures.

More fences: The fence method presented in this thesis can undoubtably be used to yield further tests for insuring the security of systems. A particularly intriguing direction to explore is to search for high-level fences which check the configuration of a running system against security constraints expressed in an alternative notation.

Strong automatic protection schemas: The operating system has additional environmental information available to it when it assigns automatic protections to files. For example, it frequently knows the file type, the current applications being run, the time of day, etc. This information may be usable to make the automatic protection scheme give a finer degree of control.

Sentinels as a software engineering tool: Sentinels are a very general mechanism and their use is not limited to security applications. What are the applications of

sentinels as software engineering tools? Here are two examples which give a flavor of what might be possible with sentinels: we can attach sentinels to window devices to provide very intelligent windowing systems; or we can use sentinels to provide version control and configuration management.

Hardware assists for fences: Much of the specialized processing used for fences can be performed in parallel with standard operating system operations. Could specialized hardware running in parallel be cost effective for increased performance?

User Authentication: This thesis did not address issues of making a better password control mechanism. Given the variety of cryptographic tools available to computer science researchers, the time is ripe to reconsider the important area of user authentication for secure operating systems.

ITOSS subsets: Is there a small subset of the ITOSS model which can be brought up very quickly for special purpose applications?

Chapter 8

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.

[Berkeley 84] *UNIX Programmer's Manual: 4.2 Berkeley Software Distribution.* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California. March 1984.

[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.

[Eswaran *et al* 76] Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger. "The Notions of Consistency and Predicate Locks in a Database System." *Communications of the ACM,* **19:**11, pp. 624–633 (November 1976).

[Fisher-Kung 84] Fisher, A. L., H. T. Kung, L. M. Monier, and Y. Dohi. "Architecture of the PSC: a Programmable Systolic Chip." *Journal of VLSI and Computer Systems*, **1:**2, pp. 153-169 (1984)

[Gold-Linde-Cudney 84] "KVM/370 in Retrospect." *Proceedings of the 1984 Symposium on Security and Privacy*, Oakland, California, May 1984, pp. 13–23.

[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).

[Kernighan-Plauger 76] Kernighan, B. W., and P. J. Plauger. *Software Tools.* Addison-Wesley, Reading, Massachusetts, 1976.

[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).]

[Lampson-Sturgis 76] Lampson, B. W., and H. E. Sturgis. "Reflections on an Operating System Design." *Communications of the ACM,* **19:**5, pp. 251–265 (May 1976).

[Leffler-Fabry-Joy 83] Leffler, S. J., R. S. Fabry, and W. J. Joy. *A 4.2 BSD Interprocess Communication Primer.* Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California. March 1983.

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

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

[McKusick *et al* 83] McKusick, M. K., W. J. Joy, S. J. Leffler, and R. S. Fabry. *A Fast File System for UNIX.* Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California. March 1983.

[Ousterhout *et al* 85] Ousterhout, J. K., H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System." *Proceedings of the 10th ACM Symposium on Operating Systems Principles.* December 1985, Orcas Island, Washington. [In *ACM Operating Systems Review*, **19**:5, pp. 13–24 (December 1985).]

[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.