AN APPROACH TO COMPUTER SECURITY

by

James Richard Woolsey
B.Sc., University of Victoria, 1968
B.F.A., University of Victoria, 1971

A THESIS SUBMITTED IN PARTIAL FULLFILMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computer Science

ACCEPTED
FACULTY OF GRADUATE STUDIES
[redacted]
DEAN
DATE Oct 3, 85

We accept this thesis as conforming
to the required standard

[redacted]
Dr. David L. Parnas

[redacted]
Dr. Michael R. Levy

[redacted]
Dr. William E. Pfaffenberger

[redacted]
Dr. Zdenek Vintr

UNIVERSITY OF VICTORIA

April 1985

Supervisor: Professor David L. Parnas

ABSTRACT

No general purpose computer operating system can completely prevent illegal or illicit access of data that it controls. An illegal system access can be an accidental or deliberate attack on the system, and is usually due to carelessness of users or to flaws in the design of the system. Systems are flawed because they, and the assumptions on which system security is based, are not well understood. Our ideas of system security are based on the finite state machine model, which can be defined in a precise, formal manner. To enable large systems with many states to be described, we use the mode class model. Using the concept of a finite state machine, we define what a system is, what entities are with respect to a system, and what it means for an entity to be independent in a system. We are then able to describe what it means for a system to be secure. As well, we describe how the parts of systems, such as programmes and processes, can be defined as entities. These concepts permit us to describe how secure systems can be designed. As a practical example of the theory, we show how a UNIX-like system that

does not have many of the same security and allocation problems as current UNIX and UNIX-like systems can be produced.

Examiners: [redacted]

Dr. David L. Parnas

[redacted]

Dr. Michael R. Levy

[redacted]

Dr. William E. Pfaffenberger

[redacted]

Dr. Zdenek Vintr

TABLE OF CONTENTS

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank David Parnas for his patience during the evolution of this thesis, and to the Federal Systems Division of IBM (U. S.) for their financial support. I would also like to thank all those who read all or part of this thesis, for their critical comments. These include Cynthia Lane, Laura Proctor, Mrs. W. R. Woolsey, and Margery Fee, whose criticism of the style was invaluable.

I What is the basic outline of this study?

I.1 What is the subject of this study?

Computer security encompasses a broad range of topics. Besides the legal considerations and concerns for the physical security of the computer, computer security can be studied in terms of development and verification of computer systems, and in terms of ways of enhancing the protection of the information controlled by a computer system. The development of systems must be concerned at some level with the architecture of the underlying machines. If the architecture does not offer facilities that can be used to secure the system, then the system can not be made secure. System verification can be done by applying models that describe the system, provided the description is accurate. To further strengthen the theoretical verification, penetration analysis can be done in which systematic attempts to simulate violation of a system are made. There are areas of study that are concerned with the protection and integrity of information systems such as data bases. Some of these problems can be dealt with through encryption and signature analysis. No one area alone can guarantee the complete security of a system. All are open for further study.

In this report we take a fresh look at the problem of designing a secure computer operating system. We consider how the parts of an operating system and the interaction of those parts can be designed and specified to prevent unauthorised persons from using or altering the operating system or the computer on which the operating system is implemented. More specifically we consider how the gaining or changing of information in the system can be restricted, with the view of developing a design for a family of secure systems. Other areas of study concerned with computer security are authentication, inference, and encryption. Although they are important to the general security of a system, we will not deal with them. Also the physical security of a computer will not be discussed since, a physical attack is difficult, if not impossible, for the operating systems to prevent.

I.2 What is the motivation of this study?

With the introduction of multiuser and distributed computer systems, the danger of illicit accesses to a computer system from within the system has become greater. Computer crime is quickly becoming the most prevalent kind of white-collar crime. Much is because of carelessness by installation managers who do not properly apply the available protection mechanisms. But some illicit or unintended accesses occur because of weaknesses or flaws in

the operating systems. These flaws can create conditions that allow deliberate and accidental incursions into the systems.

Included as part of a system is the information stored in the system. When large amounts of potentially valuable information are entrusted to computer systems, and more people have legitimate access to it, there is greater chance for the system to be violated. No general purpose computer system, civilian or military [LAND81b, STRYKER74] has been proven to be immune from all possible attacks by people who have access to the systems and who want access to the information. Most of these attacks on systems are made using programmes that are available in the systems. An intruder who uses a system's programmes to violate it may leave no detectable trace of an unauthorised access.

I.3 How do we plan to solve some of the problems that concern computer security?

In this study we try to isolate some of the fundamental flaws in the design of current systems and then suggest a way of designing a system that is free of these flaws. We start in chapter two by looking at the current approaches to operating system security. First we outline the current methods of describing operating systems and operating system security. We then present approaches that are used to develop models of operating systems that are supposed

to be secure. Some of the basic principles that are regarded as important to the development of secure systems are included in this discussion. The subject-object concept of security is discussed and some of the models that are based on this concept are described.

In the third chapter we outline some of the assumptions that are made in the current models. We give reasons why these assumptions are not valid. For these reasons, models of operating systems must be based on other assumptions.

In the fourth chapter, we start to introduce an approach that can be used to develop more secure operating systems. We review the formal definition of finite state machines and outline methods of describing them. This includes describing the mode class model. With this model, machines with large numbers of states and large numbers of transitions can be described practically. We then introduce our definition of process and programme. To conclude this chapter we present a more formal description of an operating system.

In the fifth chapter of this report we define what it means for a system to be secure and what an access right is. These definitions are based on the concepts of independent entity and authorised state which are also defined in this chapter.

An outline of the methods that can be used to specify a secure system using the concepts and definitions that have been presented is given in chapter six. Ways are suggested in which system independence could be specified using the concepts of independent entity and data types using the mode class model.

In chapter seven we consider the theory that we have been developing in a practical application. We show how our ideas can be used to design a UNIX-like operating system that does not have the security and allocation problems that current UNIX and UNIX-like systems have. In a final chapter we state why we feel the ideas that are suggested will lead to families of better, more secure operating systems.

II What is the current approach for developing and studying operating systems that are supposed to be secure?

II.1 What duties are operating systems expected to perform?

Between the users of a computer and the physical machine are several layers or levels of software. The operating system is usually in the top level and acts as an interface between the users and the computer. It is expected to handle most tedious housekeeping duties and to allocate and control system resources such as information storage devices and processors for the system and users. In multiuser systems, which are systems that can be used by more than one user at the same time, an operating system must be able to give to each authorised user some reasonable share of the resources. An operating system must also be able to control or restrict the flow of information within the computer so that the information does not become lost; does not become available to other users without permission; and does not destroy or disrupt the work of other users. This means that an operating system must be able to supply some quantity of protection to the computer and the information controlled by the computer.

II.2 How is operating system security currently described?

II.2.1 How is an access currently described?

Often, in the design or study of operating systems, a theoretical description of the system is produced. This description is considered to be a model of the system. Current models generally divide the system into a set of interacting units, some of which "contain" information. The ways of manipulating or using each unit or the contained information are referred to as accesses [JONES75, SALTZER75]. If a programme or a user, who can be viewed as a high level programme, is able to manipulate the unit or the contained information, the access is known as a write access. If a programme is able to make use of the contained information, the access is known as a read access. The models then state that a programme that has been given permission to use an access has an *access right* and that the access has been *authorised*.

II.2.2 When is an operating system considered secure?

Operating systems are subject to three general kinds of threats [LAND81a, SALTZER75]:

1. the ability of programmes to read information from the system without authorisation;

2. the ability of programmes to write into the system in an unauthorised manner;

3. the ability of programmes to cause, without authorisation, events that inhibit or disrupt the services of the system.

These threats may be deliberate or malicious attacks on a system or they may be accidental because of errors in programmes. To be considered secure, an operating system must be able to protect itself and its parts from these threats.

II.3 What is the current approach for developing models for secure operating systems?

II.3.1 What are some of the principles for developing secure operating systems?

Protection and security of any system can not rely on it being absolutely secure, since systems never are. Several principles have been suggested that are supposed to help make illicit accesses more difficult. Some of the principles listed below are given in [SALTZER75].

1. Operating systems should be small. A large system can supply many avenues of attack while there are fewer ways that can be used to "break into" a small system. A small system is also easier to specify, implement, and check.

2. Security should be considered throughout the

design of the system. It should not be an add-on feature [LAND81b]. If the assumptions in the underlying system are not consistent with the requirements for file protection, a file protection mechanism that is added without changing the underlying system will be relatively easy to circumvent. To get it to "work right" the entire system would need to be redesigned.

3. Operating systems should be implemented from formal specifications which have precise and unambiguous interpretations. When there is a vague idea of what rights are and how they are used, unintended accesses become possible.

4. It can not be assumed that the right to make an access can not and will not change. The mechanisms for accessing the parts should automatically prevent an access when the permissions have been removed.

5. A system should be based on permission instead of exclusion. If a user is not permitted to access a file, not even the name of the file should be available. This control makes it much more difficult to bypass the exclusion mechanism.

6. Systems should follow the concept of least privilege. Users should not have more privileges than they require. In this way the chance of illicit accesses to information is reduced, and if information is changed it is easier to find out the source

of the change.

7. The security of a system should not depend on keeping the design of the system secret. Otherwise, once the secret is out the system is no longer secure.

II.3.2 What is the subject-object model for secure operating systems?

Most current models of operating systems refer to the set of interacting units as "entities". These models have been described by Shankar [SHANKAR77] as follows.

> *Objects* are defined to be these information-holding entities....To develop the concept of an active entity against which objects are to be protected, an abstraction of accountability is introduced...It will be referred to as a *subject* and is the entity in the computer system to which authorizations are granted....After defining objects and subjects, the ways in which they interact are specified. This is done by formalizing the notion of access. For each type of object there exists a finite set of distinct ways in which objects that define that type can be manipulated; each type of manipulation is called an *access*. Each access to an object is made on behalf of a subject.[1]

The models then specify the conditions under which a subject may have an access to an object. If systems based on the models reflect the assumptions in the models accurately, the systems can be shown to be secure. Under

---

[1] K. S. Shankar: "The Total Computer Security Problem". *Computer*, June 1977, page 55.

assumptions in the models, the systems are logically complete (all possible access requests are considered); and they are logically consistent (given the same conditions, the effect of an access does not change) [MCLEAN82]. We will show in a later section of this paper that there are basic flaws in the use of these theoretical models that leave the systems open to security problems.

II.3.3 What models have been developed using the subject-object approach and what current systems try to use them?

II.3.3.1 What is the access matrix model?

In the access matrix model [LAMPSON71], a matrix defines the permitted accesses of each subject to each object. This model has been variously extended [GRAHAM72, HARRISON76], and two basic forms have emerged that use lists to represent the matrix [LAND81a]. One form is the access list. In this form, a list is maintained for each object. It defines what accesses each subject can have to that object. The UNIX system has been modeled by this approach [HARRISON76]. Each file or device is an object and each process, which represents a user, is a subject. Accompanying each file or device is a protection vector that indicates the accesses permitted to the different classes of users that are represented in the system.

The other form of the access matrix is the capability list. In this form, a list is maintained for each subject. It defines what accesses the subjects can have to each object. HYDRA [JONES75] is a system whose protection system was developed with this model in mind. HYDRA is composed of objects of various classes or types such as files, pages, and semaphores. Each user has the capability to make zero or more accesses to each object. These capabilities or rights are recorded in capability lists known as C-lists.

II.3.3.2 What is the information flow model?

The information flow model is more concerned with the information that can be associated with the entities [DENNING76 DENNING77]. All information in the system is given a security rating. Each object and each subject is given a security rating which relates to the security rating of the information that can be associated with that object or subject. The model then defines rules used to specify where in the system information can be transmitted with respect to the security ratings of the information and the entities.

The ADEPT-50 system is such a system [LAND81a]. The security ratings of objects, such as terminals and files, are described by sets of properties that designate sensitivity levels, compartment sets, and permitted users.

Information is associated with sensitivity levels and compartment classifications. A subject or user can only cause information to pass from one object to another if the rules that govern the passage of information are not violated.

II.3.3.3 What are some finite state machine models?

In a finite state machine model the system can assume a finite number of states. The accesses are defined in terms of transitions from one state to another. Two finite state machine models, one by Goguen and Mesequer [GOGUEN82] and one by Millen and Furtek [MILLEN78, FURTEK78] define the system security in terms of the current state of the system. The model discussed by Goguen specifies, given the current state, what the permitted state transitions are for the system. The authors claim that PSOS (Provably Secure Operating System) fits the framework of this model. The state of the system is characterised by "users' programs, data, messages, etc.", which are the objects in the system. The capability component in the system keeps track of what state changes the subjects (users) are permitted to cause.

The model by Millen and Furtek applies more strictly to the security within programmes, and specifies what transitions are not permitted by the system when it is in a

particular state. In this way a programme is prevented from being able to make certain accesses unless specified variables in the programme are in specified states. These restrictions are referred to as constraints.

II.3.4 How successful have these models been?

Each system design that is supposed to fit one of the models proposes the existence of enforcement mechanisms that allow only named subjects to have access to named objects in the ways allowed by the model. Since they fit a security model, these systems are then shown to be secure, but empirically, they remain vulnerable to security threats.

III What are some of the flaws in the models that are now used?

III.1 What is a more formal definition of a model and when is a model useful?

A *model* is an abstract description of a system or of some aspect of a system. With respect to a computer system, a model can be used to describe and to predict the behaviour of the system. Behaviour, in this context, refers to the changes in the system and the output that occurs when a specified input is received by the system. Since a model is an abstraction, it can be used to specify more than one realisation of a system. For the model to be useful, the behaviour and assumptions that are true for the model must be true about the realisations based on the model.

III.2 What is independence and what assumptions do the models make about independence?

The models on which security systems are based, assume, often implicitly, that entities will not interact with each other unless specified to do so. It is assumed that if one entity causes a change in another entity, it will not unknowingly change a third entity. It is also assumed that an entity will not unknowingly restrict changes in

another entity. This means that entities are assumed to be *independent* of one another except for the defined interactions. (For a more formal definition of independence see Glossary.) If these assumptions about the models hold, the operating systems represented can be shown to be as secure as can be proven in the model.

III.3 What is wrong with the assumptions?

III.3.1 When are subjects and objects not independent in implemented systems?

The association between the current abstract models and the current systems is not accurate. Although system designers think that systems adhere to the assumptions in the models, they do not. In some systems the subjects are not independent of one another. This means that one subject may inadvertently affect the ability of another subject to make an access. For example, the designers of the model for the EXEC III Operating System had a "fuzzy" concept of environment [STRYKER74]. Under certain conditions when using a non-system reentrant programme from inside another programme, a user had to prepare the data area with the data before calling the reentrant programme, which then established error handling procedures. If an error occurred in the reentrant programme before the error procedures were established, the reentrant programme gave the ability to access the data area to the user of the

programme who last established error handling procedures. A previous user of the reentrant programme then could have full access to information in the data area that would otherwise be inaccessible. To demonstrate this security flaw, Stryker broke into a military system by establishing faulty reentrant programmes in the system.

In other situations the objects are not completely independent. An access to one entity might, in effect, be an access to others. When this occurs using temporary or permanent shared storage, it is known as a security leak through a "storage channel" [LAND81a]. In the early IBM/360 systems it was possible for users to read primary memory directly [SALTZER75]. If this memory is considered a common part of objects of a particular kind, then an access to one object of that kind was, in effect, an access to parts of other objects of the same kind. The models of systems, therefore, do not correctly specify all subject-object interactions in systems.

III.3.2 Why and when are dependencies necessary?

There are many situations in operating systems when it has been found to be necessary or convenient to define dependencies between some of the entities of the system. A change in the ability of one entity in the system to make an access can be used to change the ability of another entity to make an access. In the UNIX system,

access privileges can be given to "groups" or sets of users, as well as to individual users. The members of these groups can be altered as the needs of the system change. In other cases, an access to one entity can be an access to several. If characters in a file are considered entities, then often the ability to access a character in the file implies the ability to access all characters in the file.

These dependencies help make systems less complex by considering entities that have qualities in common as classes, and therefore recognise inherent dependencies that entities must have for being in the same system. When designing a system, a system designer should clearly define the entities in the system and precisely specify their interdependencies. If, as in current systems, entities are considered independent when they aren't, systems will continue to have security flaws. We therefore think that security should be considered by using the concept of entity interdependence.

III.4 Why is it difficult to correct existing systems?

It is not usually practical to correct flaws in existing systems. Because models of systems are not well understood and are not specified precisely, fundamental assumptions in the models are violated by the systems. This leads to security flaws. Often a flaw is caused by

the presence of a system "feature" that is flawed. Flaws are usually corrected by trying to use a "patch". This means that some part of the system is changed in some way, or that more software is added to the system to cover the fault. Patches are normally flawed themselves [NEUMANN78]. They can violate some other fundamental assumption in the model that can lead to other unexpected security problems. Until the interaction of the parts of a system is understood, it will be impossible to show that corrections to system security are effective. Even if the system is well understood corrections can remain difficult to implement because changes to the system would mean changing the entire system.

IV What is a more formal and precise definition of an operating system?

IV.1 What is a finite state machine?

IV.1.1 What is a formal definition of a finite state machine?

To give a precise definition of an operating system and of the contained entities, the concept of finite state machine must be understood. The finite state machine model was described by Turing [TURING50] as the discrete state machine. Many forms of the model have since been developed by others. It is characterised by a set of states, a set of transitions, an input alphabet, and an output alphabet.[2] An *input* is a value from the input alphabet and an *output* is a value from the output alphabet. Two relations can be used to describe a machine. The first relation describes *events* or state changes. For each input/current-state pair the next state relation describes the set of next possible states. The output relation describes the set of possible outputs associated with each machine state. These relations are encompassed

---

[2] Many definitions of finite state machines include a set of start states and a set of final states which are subsets of the set of states. These sets of states are not relevant to the finite state machine model we discuss because any state of the machines described by our model can act as a start state and the machines are not necessarily expected to reach a final state.

in the transition relation for the machine.

IV.1.2 What are the traditional approaches for describing finite state machines?

There are many methods that have been used to describe the finite state machine model [PARNAS82]. They fall into two basic classes. The first uses tables to enumerate the inputs, states, and outputs. The events can be determined from the state transitions defined by the tables. The state of the machine can be described by enumerating the sequence of events since the machine was in a known state, or by giving a set of predicates that characterise the state of the machine. The predicates in the set are known as a *conditions*.

The other way to describe a finite state machine is to use an axiomatic approach. With this approach, and with care, a concise description of a finite state machine can be generated without listing any states. The machine is specified by specifying functions that describe events and axioms that apply to these functions. The axioms are specified by describing the permitted sequences of functions and therefore the permitted sequence of events. The current state is derived from the history of the events that have occurred. By specifying equivalent sequences of events, one can show states to be equivalent and can substitute one event sequence for another. For example, con-

sider the integers, modulo four with addition. The functions and axioms are given in Table 1. As can be seen, since the

Table 1.

Functions and Axioms for Addition Modulo Four.

```
Functions:
     O-functions:
          init   -->  <integer>
          add1  <integer>  -->  <integer>
          add2  <integer>  -->  <integer>
          add3  <integer>  -->  <integer>
     V-functions:
          val  <integer>  -->  <value>
Axioms:
     legalities:
          L(init)  =>  L(init.add1)
          L(T.add1) => L(T.add1.add1)
     equivalences:
          (T.val) = (T)
          (T.add1.add1) = (T.add2)
          (T.add1.add2) = (T.add2.add1) = (T.add3)
          (T.add1.add3) = (T.add3.add1) = (T)
    values:
          V(init.val) = 0
          V(init.add1.val) = 1
          V(init.add2.val) = 2
          V(init.add3.val) = 3
```

sequence (init.add0.add2.add3) is equivalent to the sequence (init.add1), and the sequence (init.add1.add2.add2) is equivalent to the sequence (init.add1), they are equivalent to one another. Other examples of this approach are given in [BARTUSSEK77].

IV.1.3 What are the limitations with these approaches?

These methods of describing finite state machines can have some serious limitations. When the number of states or the number of possible input or output values becomes very large, or when the events cannot be described by a few well defined, well understood rules, these may not be practical ways of describing a finite state machine. An operating system is such a machine. It usually has a very large number of possible states and a large number of possible transitions that are not usually well defined or well understood.

IV.1.4 What is the mode class model?

An alternate way of describing a finite state machine is to describe the machine with a number of partial descriptions [HENINGER80]. To do this the set of states of a machine are partitioned to form a *mode class*. Each partition within a partitioning is a *mode* within that mode class. Although the modes of a mode class can sometimes be described by enumerating the conditions that characterise the machine states for the modes, a transition table has been found to provide a more useful description [HENINGER78]. The events that lead from one mode to another are enumerated. In this way changes that are not immediately visible outside the system can be specified.

To determine the current state, the history of events or trace is used.

Each mode class is a finite state machine for which each mode of the mode class is a state. Using the mode class model, a finite state machine can be described as fully as desired by describing a mode class for each aspect of interest, and determining the events that cause changes to the modes. If a enough mode classes are defined, individual states of the system can be described by listing the current modes of all the mode classes. Such a detailed description is not usually required when working with large systems. An operating system can be described in this way, using the mode class model. To do this it is necessary to determine the aspects of the system that are relevant and to describe these as mode classes.

IV.1.5 What is an access to a finite state machine?

To have *access* to a finite state machine is to be able to change the state, or to return information about the state of that machine. If the access changes the machine state, it is known as a write access, and if the access returns information about the state, it is known as a read access. The information that is encoded in the state of a machine is referred to as *data*. Different accesses can be specified by restricting the kinds of state changes that

are permitted, or by restricting the kinds of information that can be returned.

IV.1.6 What is an operation in terms of finite state machines?

An operation can be described by an LD-relation [PARNAS83c]. An LD-relation defines a set of events on a set of states of a finite state machine. (For a more extensive definition of LD-relation see Glossary.) If the competence set of the LD-relation is not empty and the operation is started in one of those states, the operation is guaranteed to terminate. The relational part of the LD-relation is a set of ordered pairs of states of the form (A, B) where

A is a possible state of the machine before the operation and

B is a possible state of the machine after an operation started in A.

The set of possible initial states is the domain of the operation and the set of possible final states is the range of the operation. If the relation defined by the relational part of the LD-relation is a function then the operation defined is deterministic.

Classes of operations can be described by specifying a class of relations for a class of machines. If the machines are integer variables then an example of a class

of operations is described by 'x := x + 2' where 'x' identifies some integer variable. The operation described increases the value of 'x' by two. By replacing the '2' with a variable 'b', the operation becomes 'x := x + b' and describes a class of operations where the final value of 'x' is increased by the value represented by 'b'. In this way 'x := x + b' can be used to express many possible relations that describe addition operations.

IV.2 How can an entity be described?

A finite set of interrelating finite state machines can be combined into a another finite state machine, or *finite state system*. We define an *entity* as each component machine, or submachine, of a system.[3] An entity can also be considered a *variable* that has values that are characterised by its state. In a computer system, entities are such things as bits, files, and devices. A system can also be an entity of a larger system.

IV.3 How is a system described?

The current state of a system is described by describing the current state of all the component entities. The possible set of states of the system can be restricted by

---

[3] Although not strictly correct, a finite system will simply be referred to as a system in the remainder of this paper.

not permitting particular combinations of entity states. The transition relation of a system is the combination of the transition relations on the system entities, since, if an entity changes state, the system changes state. It can be specified by specifying all entities and all operations within the system, or can be specified by specifying classes of operations on classes of entities. A system can also be specified using the mode class model where each mode is associated with an entity within the system.

IV.4 What is a process and a processor?

A *process* is a set of events or set of state changes in a finite state machine or system. A single system event may change the state of more than one entity in the system. Events in processes can be caused in two ways. If the state of an entity is changed by the change of state of an input device, then the event is caused from outside the system by input. If an event is caused from within the system, then it is caused by an entity from a class of entities known as *processors*. The processors in a system define the transition relation for the system. Events caused by a processor can be associated with particular system states or with the states of particular system entities. If the events that make up the processes in a system are to be well defined, then it is necessary for the transition relations of the processors to be well

defined.

IV.5 What is a programme?

An entity whose state can define the events caused by a processor is a *programme*. The state of a programme is the input to the processor. In some situations a programme can consist of the state of one entity or can be composite and contain the states of several entities. If a composite entity in a programme is an input device, the programme may define a class of programmes. The members of the class are determined by the state of the input device. In these cases, for the processor, there is no distinction between the input device and any other entity that makes up the programme. They all constitute data to the processor. To *execute* a programme is to use the programme to determine a sequence of events. There can be more than one processor in a system, and the events defined in a programme need not all apply to a single processor.

A programme can be specified using an LD-relation. The results of each composite entity are described by the LD-relation. These are composed to define the result for the composite programme. If the programme defined by the LD-relation is not deterministic, the programme describes more than one possible process.

IV.6 What is an operating system?

An operating system is a programme that defines the permitted events, and thus the permitted states, of a computer system. Since an operating system need not terminate, the competence set of the LD-relation describing an operating system may be empty. An operating system is made up of many programmes that define accesses to many entities of several different kinds. If an operating system is to be secure, the operations on the entities must be understood and must only affect the states of the entities on which the operations are performed.

V What is a more formal definition of operating system security?

V.1 When are entities independent?

V.1.1 What is a more formal definition of independence?

For one entity to be *independent* of another entity, two conditions must hold. First, it must be able to enter any state regardless of the state of the other entity (See Table 2.a).

Table 2.

Examples of
Conditions for Entity Dependence.

Two entities exist, "A" and "B"; entity "A" has four states labeled "1", "2", "3", and "4"; and "B" has at least one state labeled "5"; then "A" is not independent of entity "B" if:
a. "A" can only enter states "1" and "2" when "B" is in its state "5"; or
b. for some state change in "A" there is also a state change in "B", or for some state change in "B" there is a state change in "A".

Second, it must not change the state of the other entity when it changes state (See Table 2.b). If two entities are not independent then a dependency exists between them. An entity is *completely independent* if it is independent

of all other entities in the system, and *partially independent* if it is independent of only some entities in the system.

These conditions imply two classes of dependence that need not be disjoint. In the first, the dependence of an entity is detectable if and only if the states of the entity are restricted by a state of another entity. In the second, the dependence of an entity is detectable if and only if it is known that a state of another entity changes for some state change of the entity. The detection of entity dependence is not always immediate. It can become apparent at a later time when the system does not act as was expected.

V.1.2 What is a structural description of entity independence?

Practically, for entities to be dependent, they must have a common component. In extreme cases one entity is a component of the other. Dependency, therefore, implies that the physical structure of the entities overlap, that the two entities are connected through the common component. A state change of the common component causes a state change in both entities, or a state of the common component restricts a state change of both entities.

A special case of dependency occurs when the common

component is a processor. Since a processor is the only entity that can cause a state change from within the system, for another entity to define or reflect the state changes of components that are not held in common, it must define state changes caused by a processor. This implies there is a dependency between the entities and the processor. If an entity is to have any effect on the definition of events in a system, it must be able to form a composite entity in which there is a processor. Examples of these kinds of dependencies are when one entity is a programme that is associated with the state changes of another entity, or when the state changes of two entities are defined by the same events associated with a programme. If the state of an entity can cause or restrict the state changes of another entity through a processor, then there is a *contingent dependency* between the two entities.

For an entity to be independent of other entities, the states and the state transitions of the components must not be affected by the states of other entities whether through the change of state of common components or through changes caused by a processor. It must be possible for a model to specify all interdependencies in the system if it is to specify an entity as independent.

V.2 How can an access be defined in terms of entity independence?

One finite state machine is said to have an access to another finite state machine if it is able to change the state, or to return information about the state of that machine. One entity can only have access to another entity if the two entities are not independent of one another. For one entity to cause a state change of another entity, a state transition of the first entity must also change the state of the second entity by changing the state of the common component, or an event in the first entity must cause a state transition in a processor that causes an event in the second entity. This is a *write access*.

To obtain information about the state of another entity, the state of the common component must affect at least one transition for the first entity, or the state of the second entity must define a state transition in a processor that causes an event in the first entity. This is a *read access*.

Networks of entities can also be set up. The output or new state of one entity defines the new state of the common component that becomes the input to another entity, or events caused by a processor cause states that define further events for other entities. Read and write accesses

should not be confused. A read access does not imply a write access and a write access does not imply a read access. In a write access an event in one entity causes or defines an event in another entity, and in a read access the state of one entity restricts or defines an event in another entity.

V.3 How can access rights be defined?

V.3.1 What is an authorised state of a finite state machine?

The set of states of a system can be partitioned into two sets that can be used to specify security. This security mode class consists of the permitted mode, which is a permitted set, and the non-permitted mode, which is a non-permitted set. The states that are in the permitted mode are said to be the *authorised states*. Inclusion in the permitted mode can be an arbitrary decision from outside the system. Specification of the permitted states is done by specifying conditions or sequences of events that characterise the states in that mode.

V.3.2 What is a security policy?

The specification of the authorised states is the *security policy* for a system. The security policy can be used to specify different requirements for different systems. Since the state of a system is determined by the

states of its entities, the security policy can specify authorised states of individual entities. If an installation requires tight security, the policy can be such that relatively few system states are authorised, whereas if security is not to be so restricted, more states can be authorised. Different security policies can apply to individual entities in a system by authorising more entity states for some entities than for others.

V.3.3 What is an access right?

An *access right* is the ability of an entity to change the state of another entity when writing to the entity or to use the state of another entity to define a state change within itself when reading another entity. This is usually done with an operation that specifies the state changes. If the system is in a state that is part of the domain of an operation of an entity, or the entity can cause the system to assume a state that is in the domain of an operation, then the entity has a access right for accesses defined by the operation.

V.4 When is a system secure?

A system is said to be *secure* if there is no way to get to an unauthorised state from an authorised state.[4] If an

---

[4] In private conversation with D. L. Parnas the concept of a *super secure* system was suggested. This is a system

access right exists that permits an operation that has an unauthorised final state, then the system is not secure. If secure systems are to be developed, the specification must be precise enough so that no unauthorised state can be reached through accesses defined by the operations for the systems. To do this the model that specifies the system must be complete: the operations for all entities must be specified; and the model must be consistent: the effect of an operation does not change if the conditions do not change [MACLEAN82]. This does not violate the definition of operation that has been presented since in a relation each value in the domain can be associated with a set of values in the range, only one of which might be used.

---

in which the range of all operations started in an authorised or unauthorised state is a set of authorised states. This assumes that there can be unforeseen events beyond the control of the operating system, such as a power surge, that can cause the system to enter an unauthorised state.

VI How can a secure operating system be specified?

VI.1 What is an entity type?

Two finite state machines are equivalent if when they are started in equivalent states and are given the same input, they produce the same output [MEALY55]. Two states are considered equivalent when they are indistinguishable. Two machines are of the same *entity type*[5] when they can be shown to be equivalent. For instance if two machines are defined with operations that are specified by integer arithmetic, both machines can be considered integer machines or integer variables. An entity type can be described by specifying the possible states for the entity and the operations on the entity. Each entity in a system is of at least one entity type. An entity can also be a subtype of another type when its specified states are a subset of the states of the other type. A system is composed of one or more entity types.

VI.2 How can entities be specified to be independent?

VI.2.1 How can entity dependence be specified?

Associated with each entity in a system is a mode class that is defined for each entity type in the system. The

---

[5] An entity type is more commonly referred to as a data type. An abstract entity (data) type is a set of entity types having a set of properties in common. The set of properties is the abstraction of the type.

system states are partitioned by considering sets of states of the entity type. When the system is in a particular mode of the mode class, the entity is in a particular state or set of states. The events that cause a change of mode from one mode in a mode class to another define the state transitions, and therefore the operations on the entity. Dependency between entities can be established by specifying exclusion or inclusion relations between sets of modes of different mode classes. Exclusion relations are relations that specify that while the system is in a particular mode of a mode class, the system is not permitted to enter specific modes of specific other mode classes. Inclusion relations are relations that specify that if the system is to enter a particular mode of a mode class, then the system must either be in a specified mode of another mode class, or a specified mode of another mode class must be assumed at the same time.

VI.2.2 How can an entity be made to be independent?

Entities can be made to be independent in a system in two different ways. First, a change to the system can be made by removing or deleting entities, or the common parts of entities. For example a device can be disconnected. If there are no common components between two entities then there is no way that one entity can have an access to the other. Another way to change a system is to redefine an

operation. With this kind of change, states would be added to or deleted from the domain of the operation. This method reflects a redefinition of the security policy. These are changes to the actual system, not just to the states of the system. Reestablishing the system requires reconnecting the entity or redefining the operation.

The second way to make an entity independent or to restrict the dependency of an entity is to have the system enter a mode where there are no dependencies between that entity and others. This is a mode where the entity can enter any state that is within that mode regardless of other modes in the system, and where operations on that entity do not cause a state change to other entities, and thus possibly other modes in the system. The mode that determines the dependency can be changed by changing the state of the entity. For example, removing or erasing files from the system. When this happens operations to the entity cease to have any effect on the entity. The mode that determines the dependency of an entity in the system can be changed by changing the state of a second entity. For example, establishing a contingent dependency such as protections bits or capability lists. By changing the state of a second entity, the state of the protected entity can remain constant. This kind of dependency is important for protecting entities that must be

maintained in a given state. To be effective, the set of entities that define the mode must be sufficiently independent to ensure that the mode can only be changed by specified entities of the system.

VI.3 Why do we think that this approach will produce a more secure operating system?

When the security mechanisms are used properly, the violations to the system security occur mainly through channels that were not expected by the designers of the system. These channels are present because the overwhelming complexity of computers and of their accompanying operating systems are not well understood. Systematic checks to these systems are not practical or even possible for most systems. Often proofs of correctness for a system depends on an intuitive understanding of the system. All possible results of all possible operations are not known or understood. If a system is to be truly secure, the dependencies and independencies that are in the system must be defined and understood. If an entity is specified to be independent then it must not interact with other entities, and if it is specified to be dependent then it must be dependent in the ways that have been specified. With a better understanding of what entity independence is and by using the mode class model to specify a system, we can design a system in which the required systematic

checks can be made, and which will be able to avoid the security problems that are present in current computing systems.

VII How can this method be used to produce a secure operating system based on the UNIX operating system?

VII.1 How can the UNIX operating system be described?

Within a relatively short period of time the UNIX system has become an important and influential computer operating system. In creating their operating system, the builders of UNIX implemented many new ideas, while they took other ideas from other systems and improved them. Although UNIX has proven to be a worthwhile system, there are problems with allocation of resources and maintaining security.

Before considering changes that would be made to the existing UNIX system to design a more secure and efficient system, some of the motivation and philosophy behind the original system design should be understood. It was originally developed in the period 1969-1970 to run on the DEC PDP-7 and PDP-9 for a group of researchers. As such it established itself and has since been more fully developed and adapted to other computers such as the VAX 780/11 and the Interdata 8/32. As well, many designers of operating systems have tried to copy the UNIX system.

The heart of the UNIX system consists of the UNIX ker-

nel.[6] The UNIX kernel defines the UNIX system and is the only part of the system that can not be replaced by user programmes [THOMPSON78]. The rest of an individual system can be changed or modified to suit the needs of a particular installation. The UNIX kernel is responsible for only three areas that the designers thought an operating system should control. These are processes, file system, and I/O. The UNIX kernel is therefore totally responsible for creation, deletion and scheduling of both user and system processes, and for the allocation of primary and secondary memory for processes and files. The I/O and file systems are interrelated. Each I/O device interface is treated as a file. "Ordinary" files are simply one dimensional character arrays stored on a storage device. The name of the file is stored in a directory, which is an ordinary file that supplies the mapping from the file name to its physical location. A "device" file, whose name is also stored in a directory, maps the device name to the appropriate device driver and the physical address of the device that is being used. The design of the file system enables a file to be accessed from more than one directory entry. That is, there can be more than one directory

---

[6] This should not be confused with the concept of kernel that defines a kernel as that minimal amount of software that supplies the interface between the hardware computer and the rest of the operating system. The UNIX kernel is more general than this.

entry that is able to supply a mapping to the same physical location or device. A different name can be associated with each entry point.

Each file is associated with a protection vector. This specifies the permitted accesses for the different kinds of users to that file or device. The system partitions the set of users into "groups" or subsets of users. For each user, there are three kinds of users and three kinds of access. For each user, the set of users is partitioned into the individual user, the other users in the user's "group", and all other users. The defined classes of access are "read", which gets information from a file; "write", which sends information to a file; and "execute", which uses the information in a file to define a process. There is also a special user, the system supervisor or "root", who is always permitted to make these accesses to all files in the system.

To the user, the kernel is a set of primitive programmes that defines the interface between a process and the system. UNIX, unlike many systems, allows each user to make direct use of the primitive programmes in the kernel [RITCHIE78a]. To execute a programme, a new process must be initialised using programmes defined in the kernel. This is done when the user makes a call to the "fork" programme that makes a copy of the calling, or

"parent" process. The copy, or "child" process, that is spawned executes the code from a named file by replacing the code in the child with the code from the named file by calling the "exec" programme. By repeated calls to "fork" and "exec", users are able to execute several parallel processes. To create, read from, and write to files the system primitive programmes "creat", "open", "close", "read", and "write" are used. In most situations the user need not distinguish between I/O devices and files since the same primitive programmes are used for I/O and for accessing files. These, plus other primitive programmes that return information about the state of files and processes and that do "housekeeping" duties, are the programmes that make up the UNIX kernel.

VII.2 What are some of the desirable features in the UNIX operating system?

The UNIX system has proven to be a versatile system. This is because the UNIX kernel supplies the necessary programmes that allow the system to be expanded and modified as needed. New services and utilities as well as different devices can be added to the system with relative ease since all are treated as files [THOMPSON78]. To implement a UNIX system, it is only necessary to write a system that supplies the set of programmes supplied by the UNIX kernel. As a result the UNIX system has been adapted

to several different large computers as well as to network systems and microcomputers [WALKER83, LYCKLAMA78a, LYCKLAMA78b].

Because the interface to the UNIX system consists of calls to primitive programmes, to use the system interactively, each user requires a programme known as a *shell* that interprets user commands to the system. It is the standard command shells [BOURNE78, JOY80] that most users associate with the UNIX system. When a user initialises a session with the system there is a prescribed sequence of steps. A new process is forked by the system, a shell programme is invoked by an exec, the process is associated with a file (usually a terminal or an ordinary file that contains a set of commands), and the process is assigned to the user. The shell need not be a special shell programme. For a given user, the shell might simply be the use of a programme such as an editor. In that case the user should have limited access to the system. At any given time the shell need not be the same for all users using the system, and at any given time an authorised user may have more than one shell of the same or different kinds, as when a user executes a programme from the login shell.

For the users all major system components are characterised as files. Files are also used for specialised

purposes. Sending messages, or "mailing" to other users in the system is, in effect, writing to a mail file. Control of various services such as the line printer or computer network programmes is accomplished through the presence or absence of "lock" files. By using only files, the system is able to control existing features and to add new features using the existing primitive programmes.

The UNIX operating system does not impose any format on "ordinary" files. Any structure for a file is dictated by the programmes that use the information in the file. For instance, the format of an executable file is set by the programmes that must interpret the information in the file to the underlying system and the arrangement of a text file is dictated by a text formatting programme.

The UNIX file system is an n-ary tree structure. The nodes of the tree contain the file identifications. Directories contain the names and pointers to nodes at lower levels in the tree. The root of the tree represents the supervisor directory, while nodes within it represent the system files and directories. Within the system directories, in the lower levels of the tree, are the user directories. The user directory for each user is normally the root directory for that user's system of files and directories, but it is possible to permit a user to create files, and therefore directories, anywhere in the file

system.

VII.3 What are the main problems with the UNIX system?

The UNIX system is not without problems. These fall into two broad classes, which are not necessarily disjoint. The first is concerned with the allocation of space and time within the system. It is relatively easy for a user process to completely exhaust primary or secondary memory. This can occur because of a "runaway process" in which a "fork" creates too many processes, or in which a "loop" writes to a file. It is possible for the system to even run out of swap space, at which time the system will "voluntarily" crash [RITCHIE78b]. One undesirable side-effect of this is that information that is in primary storage, and that the system is unable to write back to secondary storage, can be lost. A user can slow the system unacceptably through accident or intent by starting many processes. During these periods of "heavy" use, the system can become very slow in responding to user requests. To terminate the processes sometimes requires a momentary system shut down. The second problem area is that UNIX has not proved to be a secure system. Many of the security problems are discussed in [BISHOP82]. Although successive versions of the system try to "patch" problems from previous versions, some kinds of security problems seem to persist.

VII.4 Why do the problems exist?

To properly understand the UNIX system it must be realised that UNIX was originally designed and built for sophisticated, cooperative users on relatively small computers. When it was introduced for public use, the assumptions about user behaviour and the relative size of the hardware system had already become part of the UNIX kernel and utility programmes. This is an example of a programme family in which initial assumptions become difficult to change when the system is expanded or modified [PARNAS76a]. UNIX systems for larger computers and more hostile user groups still assume the original programming environment. It seems amazing that UNIX has proven to be as reliable and secure as it has when used in environments of potentially hostile users.

The problems with the allocation of time and space exist, in part, because all accounting on the system is done above the level of the system kernel, and therefore is not totally effective. Since the allocation programmes are primitive, and therefore unable to use higher level accounting programmes, they do not fully control the resources. They are able to limit allocation of resources in only rudimentary ways since it is only programmes outside the kernel that consider quotas on time and space for users. These limits apply to all processes equally and not

to single users. The lower level programmes are not responsible for accounting charges, they are only able to record the actual time that a process existed, and this is done after the process ends. Therefore, if the system crashes during a process there is no system record of the process having existed. This implies that a user can violate the system or system services without leaving a record of doing so by causing the system to crash.

The problems with allocation are also, in part, because of inefficient allocation algorithms [RITCHIE78b]. As pointed out by Thompson [THOMPSON78], it is ironic that the space allocation mechanism is most efficient when it is least needed. The problem is aggravated by the fact that the system does not implement semaphores for process synchronisation, but uses the system primitive programmes, "wait" and "signal". These programmes are not based on a queue and if processes are waiting for space, they will all be signaled when new space is available regardless of how much is needed by each. There can be a "race" for it by all waiting processes of the same priority [THOMPSON78]. The system processes are always given highest priority. In some situations these circumstances can lead to resource "starvation" for some processes.

Another reason for the slow response time is because the UNIX system assumes a system interrupt for every key

stroke at a user terminal. This assumption may be acceptable for small systems. If the system is large and if there are many users generating input at terminals, the system can become very slow.

The problem with security is as basic as the others and falls into two areas. The first concerns the implementation of processes. It is assumed that, unless otherwise specified, the components of the system will not interact--that is, the components are independent. When a programme is executed, first the fork command creates an *image* or computer environment. This is the child and it is a duplicate of the image of the calling process. An image is the state of a pseudo or virtual computer [RITCHIE78a], and includes values of general registers, status of open files, a pointer to the current directory, as well as the memory for the current process. The "exec" command overlays the memory with code from another programme, making appropriate adjustments for size differences. All other aspects of the image are left unchanged. In UNIX, a process is the execution of an image. The image can either inherit the access rights that are associated with the image from which it is forked, or if a particular bit in the protection vector of the file being executed is set, it can inherit the rights that are associated with the owner of the file containing the programme. The rights of a process are those of the image

that is being executed. The problem occurs when it is possible to fork another image or to exec still another programme for an image that has the access rights of another user's programme file. Techniques for doing this are available in the electronic mail system or through the careless use of the "set user id" programmes [BISHOP82]. Patching these problems seems to be only partly effective. The problem is compounded if the image has been given the access right of the system supervisor. In this event, the system can be totally violated.

The second security problem concerns the specification of access rights. It was decided in the early development of the UNIX system that the kernel was not to be responsible for determining the levels of security in the system and that the default access rights for user files would be decided outside the kernel in higher level programmes. When a file, "ordinary" or "device", is created for, or is assigned to a user by a system utility programme, a duty of the programme is to determine the value for the protection bits associated with the file. These are recorded in each user's "protection mask". A call to a primitive programme sets the bits. As has been mentioned, UNIX is a system developed for a cooperative community of users and when new user accounts are started, the protection mask is usually set for general read and write permissions. Many system programmes, such as those that operate the line

printer or send "secret mail", assume the more general read and write permissions. For a user to have protected files under these assumptions requires some effort. The protections have to be changed at appropriate times, thus leaving the files readable for short periods of time. This can be done by either changing them "manually" or providing personal utilities for this purpose. For the system to work around these assumptions requires new utilities or a rethinking of the basic assumptions of the system. None of these solutions seem practical for the existing systems or satisfactory for a secure system.

VII.5 What changes would be made?

VII.5.1 How would we design a secure and efficient UNIX-like operating system?

For any UNIX-like system to be compatible with, and to accept UNIX style software, it must supply programmes that are similar to the programmes supplied by the UNIX kernel. These programmes need not be in the kernel of the new system as long as the same services can be supplied by the system. Since the major problems of the UNIX system are because of assumptions that were made in the kernel, it is the kernel of the system that needs to be redesigned using a more realistic set of assumptions about the expected users. By designing a system using modern software engineering techniques, a family of operating systems

can be designed. The members of the proposed family of systems, or the Secure-UNIX System, can have varying similarity to current UNIX systems depending on the requirements of the users of the specific system. Systems can be specified for implementation on a range of computer systems from microcomputers to multiprocessor mainframes and networks. Secure-UNIX can be designed by defining the entity types required by a UNIX-like system, then designing a system based on the interaction of these types. Each entity type can be associated with a mode class of the system. If systems are to be well understood, the operations on the entities must be such that the interdependencies between the entities are specified precisely. The different family members can be defined by considering different subsets of the entities and redefining the dependencies between them.

The design will need to consider various system structures [PARNAS83b]. One structure is based on the "uses" relation [PARNAS76b, PARNAS76d]. Secure-UNIX is designed as a hierarchy of virtual computers. This consists of levels of programmes in which programmes "require a correct version of", or "use" only other programmes that have been defined in lower levels of the hierarchy. At the lowest level, programmes only use programmes that are defined by the physical computer. At higher levels, programmes can be defined that supply, in a more convenient

way, the other services the system must supply. (For a discussion on the use of the word "convenient" in this context, see [PARNAS76b].) At the top level will be a set of programmes similar to, or that can be used to construct programmes that supply the same services as the UNIX primitive programmes. This set of programmes will define the operations for each class of entity that is found at the top level of the hierarchy. It can vary with the needs of the installation and family member that is represented.

To simplify the design of a large system, the design can be divided into a set of work assignments, or modules [PARNAS72]. The specification of each module gives a clear, precise definition of what the module is supposed to do. Often the module defines an entity type, or the interaction between entity types, and it is therefore in the modules that the assumptions about the system are implemented. If a module is too large for one work assignment it can be decomposed by defining submodules that can also be further decomposed, thus forming a hierarchy of modules.

The top level modules of Secure-UNIX will be similar to those found in the *A-7E Software Module Guide* [BRITTON81]. There are two modules from this system that are relevant to a non-real-time operating system. These are the HARDWARE-HIDING MODULE, in which is implemented the vir-

tual hardware that is used by the rest of the software and the SOFTWARE-DECISION MODULE, in which is implemented the software structures in the system and the software design decisions and considerations for the system.

VII.5.2 What entity types would be specified?

The entities that are to be supplied by Secure-UNIX are those entities that are needed to handle the same areas as the current UNIX system. They are entities to handle processes, I/O, and a file system and will be defined in two high level submodules, the HARDWARE-HIDING MODULE and the SOFTWARE-DECISION MODULE. The HARDWARE-HIDING MODULE consists of the EXTENDED-COMPUTER MODULE and the DEVICE-INTERFACE MODULE. These define the entity types that would be expected from a hardware computer and the virtual devices that are defined in the system and define the basis for the files and the images. The SOFTWARE-DECISION MODULE contains the APPLICATION DATA TYPE MODULE that defines the file system and higher level concepts of a process. The entities that are needed and that are supplied by these modules are similar to those in [PARNAS83a] and are specified as follows:

Level one

data: those primitive entity types with the following characteristics.

classes of data

character: an ordered set of n + 1 values such that there is a functional relation between the values in the set and the integers numbered from 0 to n.
numeric: characterised by range and resolution.
pointer: takes on value of the data type to which it refers.

Level two

array: an ordered list of a named type. Each element is treated as the individual type.
semaphore: a numeric type used for system synchronisation.
timeint: a numeric type of specified range and resolution to record elapsed time.
type_pointer: a type that takes on the value of system or user defined types.

Level three

character_string: an array of a specified length of characters. (If the characters in the string are binary valued, the character_string is a bitstring.)

devices:

classes of devices

clock: a device that can cause timeint variables to change state.
i/o device: a device that makes character information about the state of the system available to outside the system, or that returns information about states external to the system by changing a character variable in the system.
storage: a device that records character information about the state of the system for future use by the system. The format of the stored information is a secret of the storage device.

Level four

file: a type that contains two components: a state description component and an array component. The state description component is used to supply information about various

aspects of the file such as:

1. the identification of the file,

2. the identification of the owner of the file,

3. the identification of the group to which the file is assigned,

4. the accesses that the owner, group and other users can make on the file,

5. the time and date the file was created,

6. the time and date that the file was last changed,

7. the size of array component,

8. the number of links to the file,

9. the physical location of the array component.

file descriptor: supplies a reference to the array component of a file and a reference to inside the array component.

classes of files

directory: the array component is an array of references to files.
ordinary: the array component is an array of characters.
special: the array component is a reference to a device.

Level five

image: a type that supplies the input for the processor and is classed as a programme. The state of the image is used to supply information about various aspects of the image such as:

1. the identification of the image (process identification);

2. the identification of the owner of the image;

3. the identification of the group to which the image belongs;

4. the active status of the image, either the image is currently associated with a processor, not associated with a processor and not waiting for one, or not associated with a processor and waiting for a specified event. (For a more complete list of the status an image can have, see the UNIX Users Manual.);

5. the priority of the image. Priority is used to order images with respect to the use of processors.

6. the identification of the parent of the image;

7. the length of time that an image has been associated with a processor;

8. the length of time that an image has existed;

9. the length of time that the children of the image have been associated with a processor.

10. the length of time that the children of the image have existed;

11. the amount of physical space allocated to the image;

file system: an arrangement of files that use the directory class files to form a graph.

dir_ptr: a pointer to the current working directory. Each login image is associated with a dir_ptr that is set to a specified directory when it is created at login.

VII.5.3 What are the operations on the entities in the UNIX system?

The operations are defined by programmes in the system,

some of which are described below. In version 4.1 and 4.2 of the Berkeley UNIX operating system, as well as in other versions, several programmes are supplied in the kernel that change the state or return information about more than one entity type. They are not included in the following list since the duties of these programmes can be done by programmes that define operations on only one entity type. Creat is a programme that creates a file, links it to specified directory, and returns a file descriptor for the file. Pipe creates a file and returns a set of two file descriptors. It does not link the file to a directory and the file can be used for interprocess communications. Programmes like these programmes could be included in higher levels of the system. They would use the lower level programmes.

Processes defined by some programmes can be defined to be periodic. These are processes that are intended to reoccur at specified periodic time intervals. If a programme defines a process that is intended to be periodic, it will be specified as such.

Level one

Operations on data items.

create: returns a data entity of the specified class and with the specified attributes,

free: causes a specified data entity to be unavailable for further accesses.

set: if two data entities are of the same class and have the same attributes, one is made equal to the other.

get: returns information about the state of a

specified data entity.

Operations on specific types of data items.

character

ord: returns a numeric value for a given character value.
char: returns a character value for a given numeric value.

(These are in effect conversion operations between the numeric and character values.)

numeric

comparisons

equal: returns a numeric value of 0 if a specified numeric value is not equal to another specified numeric value, and a numeric value of 1 if a specified numeric value is equal to another numeric value.
greater_than: returns a numeric value of 0 if a specified numeric value is not greater than another specified numeric value, and a numeric value of 1 if a specified numeric value is greater than another numeric value.
less_than: returns a numeric value of 0 if a specified numeric value is not less than another specified numeric value, and a numeric value of 1 if a specified numeric value is less than another numeric value.
not_equal: returns a numeric value of 1 if a specified numeric value is not equal to another specified numeric value, and a numeric value of 0 if a specified numeric value is equal to another numeric value.

calculations

absolute: returns a numeric value equal to the absolute value of a specified numeric value.
complement: returns a numeric value equal to the complement of a specified numeric value.
add: returns a numeric value equal to the sum of two specified numeric values.
subtract: returns a numeric value equal to the value when one specified numeric values is subtracted from another.

divide: returns a numeric value equal to the division of one specified numeric value by the other.
modulo: returns a numeric value equal to the remainder when one specified numeric value is divided by another.
multiply: returns a numeric value equal to the product of two specified numeric values.

pointer
set_ptr: causes a existing pointer to reference a specified entity of a specified type.

rel_ptr: causes a ptr to take on a null value. This value can be associated with the numeric value of 0.

The operations on the referenced types are the operations for the types. These are in effect only after a set or set_ptr operation.

Level two

array
create_array: returns an array of a specified type and of specified length.
free_array: causes specified array, and therefore the elements of the array, to be unavailable for further operations.

The operations on array elements, except the create and free, are the operations for the individual types.

semaphore
up: increases a specified semaphore value by one.
down: decreases a specified semaphore value by one.

timeint
set_timeint: sets a specified timeint to a specified value.

incr: increase the value of a specified timeint by one.
decr: decreases the value of a specified timeint by one.

type_pointer
cre_typ_ptr: returns a type_pointer for a specified entity type. The initial value will be null which can be associated with a numeric value of 0.

rel_typ_ptr: sets a specified type_pointer to null, which will be associated with a numeric value of 0.

Level three

character_string[7]

comparisons

equal: returns a numeric value of 0 if a specified character_string value is not equal to another specified character_string value, and a numeric value of 1 if a specified character_string value is equal to another character_string value.

not_equal: returns a numeric value of 1 if a specified character_string value is not equal to another specified character_string value, and a numeric value of 0 if a specified character_string value is equal to another character_string value.

calculations

and: returns a character_string value equal to the logical 'and' of two specified character_string values with the same attributes.

nand: returns a character_string value equal to the logical 'nand' of two specified character_string values with the same attributes.

or: returns a character_string value equal to the logical 'or' of two specified character_string values with the same attributes.

xor: returns a character_string value equal to the logical 'xor' of two specified character_string values with the same attributes.

concat: returns a character_string value equal to the concatenation of two character_strings values.

shift: returns a character_string value equivalent to shifting a specified character_string value in a specified direction, padding in

---

[7] For a more formal definition of the operations on characters and character_strings see Appendix II.

an appropriate manner.
not: returns a character_string value equal to the logical 'not' of a specified character_string value.

devices

Operations to classes of devices.

clock
get_t: returns a timeint value from the system clock.
set_t: sets the system clock to a specified value.

i/o device
get: returns character information about the state of an i/o device.
put: sends specified character information to a specified i/o device.
storage
getch: returns character information about the state of a specified storage device from a specified location.
putch: sends specified character information to a storage device.

Level four

file
mknod: returns a file of a specified class with specified user, group, and accessibility states. The states that record the create and accessed times are also set at this time to the current time. The states that record the size of the array component are set to minimum values and the inode is set to the current location of the array component.
del_nod: causes a file to be unavailable for further operations.

Operations to the state description component:

set_inode: causes the state description component of a specified file to enter a state that can be used to supply information about the physical location of the array component of the file.
get_inode: returns character information about the physical location of the array component of a specified file.

chmod: causes the state description component of a specified file to change to, or enter a state that can be used to supply information about the permitted accesses to the file.

access: returns a numeric value of 1 if the specified accesses are permitted to the specified file, 0 if the specified accesses to the specified accesses can not be made, and -1 if the specified file can not be found.

chown: causes the state description component of a specified file to change to, or enter a state that can be used to supply information about the current owner of the file.

gtown: returns character information about the owner of a specified file.

chgrp: causes the state description component of a specified file to change to, or enter a state that can be used to supply information about the current group to which that the file belongs.

gtgrp: returns character information about the group to which a specified file belongs.

set_cdate: causes the state description component of a specified file to enter a state that can be used to supply information about the time and date that the file was created.

get_cdate: returns character information about the time and date that the file was created.

set_udate: causes the state description component of a specified file to enter a state that can be used to supply information about the time and date the array component of a file was last changed.

get_udate: returns character information about the time and date that the state of the array component of a specified file was changed.

set_fspace: causes the state description component of a specified file to enter a state that can be used to supply information about the allocated space for the array component of the file.

get_fspace: returns character information about the allocated space for the array component of a specified file.

Operations to the array component of a file:

alloc_fspace: allocates or deallocates storage space for the array component a specified file.

read_f: returns character information from a specified point in an array component of a file or from a referenced device.

write_f: sends character information to a specified location in an array component of a file or to a specified device.

file descriptor

crea_fd: returns a file descriptor a null value. The null value is associated with the numeric value of 0.

dup: returns a duplicate of a specified file descriptor.

free_fd: causes a specified file descriptor to be unavailable for further operations.

open: sets a file descriptor to reference a specified file.

close: sets a specified file descriptor to a null value. The null value is associated with the numeric value of 0.

lseek: changes the internal file pointer in a specified file descriptor in a specified way.

Level five

image

reboot: causes all images in the system to be unavailable for further accesses by returning a single system image with specified qualities. (For further information about this system image see the UNIX Users Manual.)

fork: returns another image known as a child image of the calling process. The state of the child is identical to the state of the parent except the state of the child can be used to identify the parent.

get_parent: returns identification of the parent of a specified image.

exec: causes the state of that part of the image that defines the events caused by a processor to be changed to the state defined

in a specified file.

exit: causes the current image to be unavailable for accesses.

setpid: causes the state of a specified image to enter a state that can be used to supply specified information about the "process identification" of the image.

getpid: returns information about the "process identification" of current image.

setuid: causes the state of a specified image to enter a state that can be used to supply specified information about the user of the image.

getuid: returns information about the user of current image.

setgid: causes the state of a specified image to enter a state that can be used to supply specified information about the group that is associated with the current image.

getgid: returns information about the group that is associated with the current image.

set_status: causes the state of a specified image to enter a state that can be used to supply information about the active status of the image. (This does the same duties as the "wait" in the UNIX system, but is more general.)

get_status: returns character information about the current status of a specified image.

nice: causes the state of a specified image to enter a state that can be used to supply information about the priority level of the image.

get_pri: returns character information about the priority level of a specified image.

set_s_time: defines a periodic process that causes the state of a specified image to enter a state that can be used to supply information about the amount of time that the image has had the use of a processor.

get_s_time: returns information about the amount of time that a specified image has had the use of a processor.

acc_time: defines a periodic process that returns information about the amount of time that a specified image has had the use of a

processor.

set_u_time: defines a periodic process that causes the state of a specified image to enter a state that can be used to supply information about the amount of time the image has existed.
get_u_time: returns information about the amount of time that a specified image has existed.

set_cs_time: defines a periodic process that causes the state of the parent image to enter a state that can be used to supply information about the amount of time that the children of the parent image have had the use of a processor.
get_cs_time: returns information about the amount of time that the children of a specified image have had the use of a processor.

set_cu_time: defines a periodic process that causes the state of the parent image to enter a state that can be used to supply information about the amount of time that the children of the parent image have existed.
get_cu_time: returns information about the amount of time that the children of a specified image have existed.

brk: allocates or frees space for a specified image.
acc_space: defines a periodic process that returns information about space allocation for a specified image at specified intervals of time.

file system
link: sets a file pointer in the array component of a directory to reference a specified file.
unlink: causes a specified file pointer in a specified directory to take on a null value. This value can be associated with the numeric value 0.

dir_ptr
cre_dir_ptr: returns a dir_ptr with a specified value.
free_dir_ptr: causes a dir_ptr to be unavailable for further operations.

get_wd: returns the identification of the current working directory.
chdir: causes a dir_ptr to reference a specified working directory.

Each level in the hierarchy represents the uses level in the proposed system. These levels do not correspond to modules and different modules can be represented at any level in the hierarchy.

VII.6 How would these changes help overcome the problems with the UNIX system?

VII.6.1 How would these changes help overcome the problems with allocation?

The control of processes in Secure-UNIX will be done with semaphores that can be used to define modes when allocations are permitted. The operations on the semaphores will be similar to Dijkstra's P and V operations [DIJKSTRA68] and will be used to simulate UNIX process control operations such as "wait" and "kill". These will be used in resource allocation algorithms that use a queue.

To overcome problems with running out of swapping space, the new system will allow the use of algorithms such as Dijkstra's "Bankers Algorithm" [DIJKSTRA65]. This means that information will have to be included with the image identification in the queue to indicate how much

space is needed. In this algorithm, before a consumer process is allowed to make a claim to more of a resource, there must be sufficient quantities of the resource that are available or that will become available after a "pay-back", so that the system will not become deadlocked. For our algorithm, the system will, when possible, calculate during periods of low processor use, which processes can claim more swapping space and thus speed up allocation on a request. In Dijkstra's algorithm this check is done at the time of the request.

Secure-UNIX will permit close system monitoring of system resources. Since the control of resources will be contained in the kernel, the system will be able to take more decisive action when a user trys to overspend an allocation. One major change is to make a per user resource allocation policy. At present, allocation is per process. Since a single user is able to initiate several processes, a user can take more than a "fair share" of the resources while using the system. The new system will allow an allocation limit to be placed on individual users. The duties of the accounting primitives will be extended and will be able to do periodic updates of the accounting files. Different users can then be given different quotas of time and space. Different user quotas have been tried in different UNIX installations, but have never been considered totally successful since the high

level mechanisms have been found to be easily circumvented by more sophisticated users.

In current UNIX systems, mechanisms that apply limits on secondary memory are not generally implemented since it is thought this would cause underutilisation of file space [RITCHIE78b]. These mechanisms have been shown to be necessary for a less trusted user population. By including monitoring mechanisms in the kernel the controls can operate in conjunction with the allocation mechanisms, thus making control of memory more practical.

VII.6.2 How would these changes help overcome the problems with security?

In any system designed to be secure, it should be possible to give to an entity only those rights required to perform the needed operations. This means that users in a system might not even know about the existence of other users through information available in the system. For a system to be secure the specification of entities must ensure that entities are independent of other entities unless access rights have been explicitly permitted. The enforcement of the rights of entities in Secure-UNIX is done with programmes found in the RESOURCE-MONITOR MODULE, which is a submodule of the SOFTWARE-DECISION MODULE. The security of Secure-UNIX, and any system, can be assured by precisely specifying the entity types in the system and

the conditions that permit accesses to them.

Images are the entities in the UNIX system that describe the sequence of events for a processor. Since the events may describe accesses to other entities in the system, an image requires access rights. The changes to the UNIX image that are proposed to create Secure-UNIX are given in Table 3.

Table 3.

Changes to access rights.

UNIX	Secure-UNIX
The access rights are those of the parent image or the owner of the file that defines the image (not both).	The access rights of an image can be a subset of rights of the owner of the image plus extensions to these rights specified by the files used to define the image. Specific accesses to specific entities can be defined in this way.
The rights of the image are constant throughout the execution of the image.	An image can be divided into parts. Rights of the image may change as different parts of the image are executed.
Rights of images are passed to subsequent images when they are created (forked) unless explicitly stopped.	The user programme that defines part of an image and the rights associated with that part, has control over the passing of those rights when a new image is created.

These changes will help overcome some of the problems found in the UNIX system. One such problem is the Trojan

Horse. This is a programme that also performs an illicit access when called to perform a specific duty. In Secure-UNIX, the rights of a called programme can be limited. These changes also help to overcome the problem with the line printer programme that was discussed earlier in this paper. It will only be necessary to give that programme rights to read files to be printed, and thus the file contents will not be made available to other users. By associating access rights with specific programmes in the system, it will be possible to define more detailed security policies.

VIII How do we think that this method will help produce better and more secure operating systems?

Computer operating systems have evolved to hide the complexity of computers, and to make the use of computers more convenient for users. To make the development of systems easier, many assumptions are made about the underlying computers as well as the behaviour of the users and the programmes being executed. Because computers are not usually understood precisely and the populations of users and programmes have changed, many of these assumptions are wrong or do not now apply. They have lead to operating systems that are flawed. Some of the assumptions are fundamental to the systems to which they apply, thus making the systems impractical and difficult to correct.

Many current systems assume that the components of the systems do not interact, that they are independent unless otherwise specified. Under this assumption systems can be proven to be secure. But the components of the systems and the interaction between the components are not precisely defined. The result is operating systems that have been shown to be secure are not.

Our answer to the development of secure operating systems is first to define the operating system and its components in terms of the finite state machine model which can be precisely defined and is well understood. An oper-

ating system is defined by defining the interaction of its component machines or entities. This allows precise definitions of entity independence and security. To reduce the complexity of a model of an operating system, the system is described as mode classes which can be associated with classes of equivalent entities. In this way a system and the security of the system can be precisely defined and systematically checked in a practical manner.

Using this model a system can be designed that does not violate the principles of secure operating systems given in Chapter II of this paper.

1. Systems developed using this model can be small. Only those parts that are required by the system need be implemented. Depending on the requirements, each level of the system can evolve in several directions to produce different systems.

2. Security is a primary concern of the system design that is suggested. Entity independence, on which the security is based, is a primary concern of the system. It is not a concept that is laid on top of the system after the fact.

3. The system is formally specified. It is based on the finite state machine model that can be formally defined.

4. The rights of the entities in a system can vary. Since rights can be specified to depend on various

conditions within the system, the rights can change as the conditions change. These can be implemented according to the requirements of the specific systems and can be specified such that changes in access rights are reflected immediately to the entities in the system.

5. The rights in the system can be based on permission rather than exclusion. The specification of exclusion and inclusion relations allow specification of a system that is based on permission. In this case an access right is only permitted when a specified set of conditions is true.

6. The model allows the concept of a least privilege system to be followed. It does not assume more privileges are needed than are required by the various entities of the system.

7. The system does not rely on secrecy of the code to maintain security. It relies on an understanding and precise specification of the components and relations within the system.

The description of a secure UNIX-like system is used as a practical example to show how the model can be used to develop secure operating system. At each level of the system the entities can be specified and the security of the system can be systematically checked and verified. The higher levels of the system can then be specified and

verified without having to reconsider the lower levels. The emulation of the UNIX system is complete when all programmes that are supplied by the UNIX kernel are supplied by the new system. Since all security and monitoring programmes have been moved to the kernel and can be verified at that level, Secure-UNIX can be made to be more secure than the UNIX system without the allocation problems.

BIBLIOGRAPHY

[BARTUSSEK77] Bartussek, W. & Parnas, D. L.: *Using Traces to Write Abstract Specifications for Software Modules.* Technical Report, University of North Carolina, N. C. (December 1977).

[BISHOP82] Bishop, M.: *Security Problems with the UNIX Operating System,* Technical Report, Purdue University, (April 19, 1982).

[BOURNE78] Bourne, S. R.:"The UNIX Time-Sharing System: The UNIX Shell". *Bell System Technical Journal 55,* 6 (July - August 1978), pp. 1971 - 1990.

[BRITTON81] Britton, K. H. & Parnas, D. L.: *A-7E Software Module Guide;* NRL Memorandum Report 4702, (December 8, 1981).

[DENNING76] Denning, D. E.:"Lattice Model of Secure Information Flow". *Communications ACM 19,* 5, (May 1976), pp. 236 - 243.

[DENNING77] Denning, D. E. & Denning, P. J.:"Certification of Programs for Secure Information Flow". *Communications ACM 20,* 7, (July 1977), pp. 504 - 513.

[DIJKSTRA65] Dijkstra, E. W.:"Cooperating Sequential Processes". Technological University, Einhoven, The Netherlands, 1965. (reprinted in *Programming Languages,* F. Genuys, ed., Academic Press, N. Y. 1968).

[DIJKSTRA68] Dijkstra, E. W. "The Structure of the 'T.H.E.' Multiprogramming System". *Communications ACM 11,* 5, (May 1968), pp. 341 - 346.

[FURTEK78] Furtek, C. F.:"Constraints and Compromise" in *Foundations of Secure Computation.* R. A. DeMillo, D. P. Dopkin, A. K. Jones, R. J. Lipton (Eds.), Academic Press, N.Y. (1978), pp. 189 - 204.

[GOGUEN82] Goguen, J. A. & Mesequer, J.:"Security Policies and Security Models". *Proceedings, 1982 Berkeley Conference on Computer Security,* IEEE Computer Society Press (1982), pp. 11 - 22.

[GRAHAM72] Graham, G. S. & Denning, J. P.:"Protection--Principles and Practice". *AFIPS Conference Proceedings, 1972 SJCC, 40,* AFIPS Press, Montvale, N.J., (1972), pp. 417 - 429.

[HARRISON76] Harrison, M. A., Ruzzo, W. L. & Ullman, J.

D.: "Protection in Operating Systems". *Communications ACM 19*, 8, (August 1976), pp. 461 - 471.

[HENINGER78] Heninger, K. L., Kallander, J. W., Shore, J. E., & Parnas, D. L.: *Software Requirements for the A-7E Aircraft*. NRL Memorandum Report 3876, (November 27, 1978).

[HENINGER80] Heninger, K. L.:"Specifying Software Requirements for Complex Systems: New Techniques and their Application". *IEEE Transactions on Software Engineering, SE-6*, 1, (January 1980), pp. 2 - 13.

[JONES75] Jones, A. K. & Wulf, W. A.:"Towards the Design of Secure Systems". *Software--Practice and Experience 5*, 4, (October - December 1975), pp. 321 - 336.

[JOY80] Joy, W.:"An Introduction to the C Shell". *UNIX Programmer's Manual Vol. 2*, (November 1980).

[LAMPSON71] Lampson, B. W.:"Protection". *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*. Princeton University, (March 1971), pp. 437 - 443, (reprinted in *Operating Systems Review, 8*, 1, (January 1974), pp. 18 - 24.)

[LAND81a] Landwehr, C. E.: "Formal Models of Computing Security". *ACM Computing Surveys 13*, 3 (September 1981), pp. 247 - 278.

[LAND81b] Landwehr, C. E.: *Best Available Technologies (BATs) for Computer Security*. NRL Memorandum Report 8554, (December 21, 1981).

[LYCKLAMA78a] Lycklama, H. & Bayer, D. L.: "The MERT Operating System". *Bell System Technical Journal 55*, 6 (July - August 1978), pp. 2049 - 2086.

[LYCKLAMA78b] Lycklama, H.:"UNIX on a Microprocessor". *Bell System Technical Journal 55*, 6 (July - August 1978), pp. 2087 - 2101.

[MCLEAN82] McLean, J. *A Formal Foundation for the Trace Method of Software Specification*, NRL Memorandum Report 4874, (September 1, 1982).

[MEALY55] Mealy, G. H.: "A Method for Synthesizing Sequential Circuits". *Bell System Technical Journal 34*, 5 (May 1955), 1045 - 1079.

[MILLEN78] Millen, J. K.:"Constraints and Multilevel Security" in *Foundations of Secure Computation*. R.

A. DeMillo, D. P. Dopkin, A. K. Jones, R. J. Lipton (Eds.), Academic Press, N.Y. (1978), pp. 205 - 222.

[NEUMANN78] Neumann, P. G.:"Computer Security Evaluation". *AFIPS Conference Proceedings 47*, 1978 National Computer Conference, pp. 1087 - 1095.

[PARNAS72] Parnas, D. L.: "A Technique for Software Module Specification with Examples". *Communications ACM 15*, 5 (May 1972), pp. 330-336.

[PARNAS76a] Parnas, D. L.:"On the Design and Development of Program Families". *IEEE Transactions in Software Engineering SE-2 (1)*, (March 1976), pp. 1 - 9.

[PARNAS76b] Parnas, D. L.: *Some Hypotheses about the "Uses" Hierarchy for Operating Systems*. Technical Report, Technishe Hochschule Darmstadt, Darmstadt, West Germany, (March 1976).

[PARNAS76c] Parnas, D. L., Shore, J. E., & Weiss, D.: Abstract Types Defined as Classes of Variables. *SIGPLAN Notices, 8*, 21, (1976).

[PARNAS76d] Parnas, D. L., Hanzdel. G., & Wurges: "Design and Specification of the Minimal Subset of an Operating System Family". Presented at the 2nd Int. Conf. Software Engineering, October 13-15, 1976; also *IEEE Transactions in Software Engineering, (Special Issue), SE-2*, (December 1976), pp. 301-307.

[PARNAS82] Parnas, D. L.: *Some Perspectives on the State Machine Model*. Internal Memorandum, Federal Systems Division, IBM, Bethesda, Md., (1981)

[PARNAS83a] Parnas, D. L., Weiss, D. M., Clements, P. C., & Britton, K. H.: *Interface Specifications for the SCR (A-7E) Extended Computer Module*. NRL Memorandum Report 4843, (January, 1983).

[PARNAS83b] Parnas, D. L.: *Software Engineering Principles*, Technical Report, University of Victoria, (February 1983).

[PARNAS83c] Parnas, D. L.:"A Generalized Control Structure and Its Formal Definition". *Communications ACM 26*, 8, (August 1983), pp. 572 - 581.

[RITCHIE78a] Ritchie, D. M. & Thompson, K.:"The UNIX Time-Sharing System". *Bell System Technical Journal 57*, 6, (July - August 1978), pp 1905-1929.

[RITCHIE78b] Ritchie, D. M.:"The UNIX Time-Sharing System: A Retrospective". *Bell System Technical Journal 57*, 6, (July - August 1978), pp. 1947-1969.

[SALTZER75] Saltzer J. H. & Schroeder, M. D.:"The Protection in Information Computer Systems". *Proceedings of the IEEE 63*, 9, (March 1975), pp. 1278 - 1308.

[SHANKAR77] Shankar, K. S."The Total Computer Security Problem". *Computer*, (June 1977), pp. 50 - 73.

[STRYKER74] Stryker, D.: *Subversion of a "Secure" Operating System*. NRL Memorandum Report 2821, (June 1974).

[THOMPSON78] Thompson, K.:"UNIX Time-Sharing System: UNIX Implementation". *Bell System Technical Journal 57*, 6, (July - August 1978), pp. 1931 - 1946.

[TURING50] Turing, A. M.:"Computing Machinery and Intelligence". *Mind*, LIX, 236 (October 1950), 433 - 460.

[WALKER83] Walker, B. J. and al.:"The LOCUS Distributed Operating System". *Proceedings of the Ninth Symposium on Operating Systems Principles*, Breton Woods, N.H. (October 10 - 13, 1983), pp. 1 - 21.

APPENDIX I

Glossary

It is assumed that the reader has an intuitive understanding of the following words: information, state.

FINITE STATE MACHINE: A machine that can be described by an n-tuple (Q, I, O, R), where:

Q is a set of states,

I is an input alphabet,

O is an output alphabet,

R is a transition relation which can be defined by a subset of the relation defined by Q x I --> Q x O.

Many other definitions of finite state machine include a set of start states and a set of final states. These subsets of states are not relevant to us.

TRANSITION RELATION: The relation between a state-input alphabet pair and a state-output alphabet pair of a finite state machine (see FINITE STATE MACHINE).

INPUT: A value from the input alphabet.

OUTPUT: A value from the output alphabet. For a each finite state machine, the transition relation defines the outputs for the state-input pairs.

EVENT: A change in the state of a finite state machine. We name classes of events and describe them by a relation on the set of states, giving the set of transitions that we include in that event class.

PROCESS: A set of events.

LD-RELATION: An ordered pair (R, C), where R is a relation on a set of elements from a universe U and C is a set of elements from U. An LD-relation determines two relations of interest:

R, the first component, and
D, a set of ordered pairs (x, y), such that x is in C and y is in U.

We call C the competence set of the LD-relation [PARNAS83b].

MODE CLASS: A partitioning of the states within a finite state machine.

MODE: A partition within a mode class.

DATA: Information stored by encoding it in the states of a

finite state machine.

ACCESS: To have access to a finite state machine is to be able to

-change its state or,

-return information about its state.

To have an access means that the kinds of state changes that can be made have been restricted, or the kinds of information that can be returned have been restricted.

OPERATION: An LD-relation describes a relation on the states of a finite state machine. The relational part of the LD-relation is of the form (A, B) where:

A is a possible state before the operation;

B is a possible state after the operation if it was started in A.

If the relation is a function then the operation is deterministic.

SYSTEM: A finite set of interrelating finite state machines. A system is also a finite state machine.

ENTITY: A component finite state machine of a system.

PROGRAMME: An entity that describes the possible event sequence in a process. To execute a programme is to use the programme to determine such a sequence. The net effect of a programme can be described by an LD-relation.

PROCESSOR: An entity within a system that can cause a sequence of events within the system. A processor can use a programme plus input into the system to determine the events.

STATE OF A SYSTEM: The states of all the entities of a system.

INDEPENDENT ENTITY: Two entities are said to be independent of one another if:

- one entity can enter any state regardless of the state of the other entity and
- any change of state of one of the entities can occur without a change of state of the other entity.

We say an entity is independent in a system if it is independent of all other entities in the system.

ENTITY TYPE: When two entities are equivalent, which means that they are indistinguishable, we say that they are of the same type. Entity types are commonly referred to as data types. An entity type can be described by specifying the possible states for the entity and the

operations on the entities.

AUTHORISED: The set of system states can be partitioned into the permitted states and the non-permitted states. This is the security mode class. Those states that are in the permitted mode are said to be authorised. The inclusion of a state in the permitted mode can be arbitrary decision from outside the system.

SECURITY POLICY: The specification of the authorised states for a system.

SECURE: A system is secure if it cannot enter an unauthorised state.

RIGHT: The ability for an entity to make a state transition defined by an operation. Each authorised state is the initial state for a set of operations. These define the rights of the entities for that system state.

CONDITION: A predicate that characterises a state of an finite state machine or a state of a set of entities within a system.

CONTINGENT DEPENDENCY: There is a contingent dependency between two entities in a system when the ability of one entity to access the other is contingent upon some condition within the system.

APPENDIX II

Characters and Character Strings

CHARACTER: A primitive data type characterised by a finite bijection between character values and integer values. The range of the integer values are between 0 and r, and the order of the character set is R, where R = r + 1.

Character Operations

INDEX	<character> --> <integer>
CHAR	<integer> --> <character>

Effects

INDEX returns an integer whose value is defined by the bijection used to define the character set with respect to the character parameter.

CHAR returns a character whose value is defined by the bijection used to define the character set with respect to the integer parameter.

CHAR_STRING: A sequence of specified length of characters from a specified character set.

Char_String Operations

NOT	<char_string> --> <char_string>
AND	<char_string> <char_string> --> <char_string>

```
OR      <char_string> <char_string> --> <char_string>
NAND    <char_string> <char_string> --> <char_string>
NOR     <char_string> <char_string> --> <char_string>
XOR     <char_string> <char_string> --> <char_string>
MINUS   <char_string> <char_string> --> <char_string>
CAT     <char_string> <char_string> --> <char_string>
EQ      <char_string> <char_string> --> <boolean>
NEQ     <char_string> <char_string> --> <boolean>
```

Effects

To define operations on char_strings, p1, p2, p3 will be used to designate the char_strings, nk where k is on of 1, 2, or 3 will be used to designate the maximum character position for the appropriate string, and i and j will be used to designate individual character positions. The maximum integer value of the characters in the char_string is designated by r.

```
NOT   returns a value equal to
      CHAR(r - INDEX(p1[i])) 0 <= i <= n1

AND   returns a value equal to
      CHAR(minimum (INDEX(p1[i]), INDEX(p2[i])))
      0 <= i <= n1

OR    returns a value equal to
      CHAR(maximum (INDEX(p1[i]), INDEX(p2[i])))
      0 <= i <= n1

NAND  returns a value equal to NOT(AND(p1, p2))

NOR   returns a value equal to NOT(OR(p1, p2))

XOR   returns a value equal to
      OR(AND(p1, NOT(p2)), AND(p2, NOT(p1)))
```

```
MINUS returns a value equal to AND(p1, NOT(p2))

CAT   returns a value equal to p3
        p3[i] = p1[i]
          0 <= i <= n1
        p3[n1 + j + 1] = p2[j]
          0 <= j <= n2

EQ    returns true if
         n1 = n2
         and
         p1[i] = p2[i]
           0 <= i <= n1
      otherwise false.

NEQ   returns true if
         n1 not= n2
         or
         there exists p1[i] not= p2[i]
           0 <= i <= n1
      otherwise false
```

APPENDIX III

Readings in various areas of computer security.

Architectures.

Lampson, B. W. & Sturgis, H:"Reflections on an Operating System". *Communications ACM 19*, 5, (May 1975), pp. 251 - 265.

The authors briefly describe the CAL Operating System. They outline those areas that they considered to be successes. But more important, they try to isolate those areas that they thought failed and try to describe the reasons for the failures.

Parnas, D. L. & Price, W. E.:"Using Memory Access Control as the Only Protection Mechanism". *Proceedings of the International Workshop on Protection* in *Operating Systems*, (1974), pp. 13 - 14.

This paper briefly describes the concept of the virtual machine mechanism and how it can be applied as the only protection mechanism in a system. The discussion includes a description of the required architectural features to support the virtual memory mechanism and a description of system control through the mechanism.

Saltzer, J. H.:"Protection and Control of Information Sharing in Multics". *Communication ACM, 17*, 7, (July 1974), pp. 388 - 402.

Multics is an operating system whose primary aim is to ensure system security. Useful operating system tend to grow and change continually. In this paper the author presents a description of the system as it was at the time of the writing, and comments on its protection mechanisms.

Saltzer, J. H. & Schroeder, M. D.:"The Protection of Information in Computer Systems". *Proceedings of the IEEE 63*, 9, (March 1975), pp. 1278 - 1308.

In this paper the authors review the development of machine architectures. As system needs change, so change the designs of the underlying machines. These design changes are outlined for various security needs, and are then discussed with respect to current machines. The paper concludes with a review of research directions.

Coding and Encryption.

Kline, C. S. & Popek, J. G."Public Key Vs Conventional Key Encryption". *AFIPS Conference Proceedings*, 48, 1979 NCC.

In this paper the authors discuss various techniques of protecting information through both conventional and public key encryption. They conclude that what is really needed is research into better, more trusted encryption algorithms.

Rabin, M. O.:"Digitalized Signatures" in *Foundations of Secure Computation*. R. A. DeMillo, D. P. Dobkin, A. K. Jones, & R. J. Lipton (Eds.) Academic Press, N.Y. (1978), pp. 155 - 168.

The author discusses the concept of digital signatures, which can accompany computer based messages, and which is used to authenticate the identification of the sender. This method uses the text of the message and a set of predetermined keys to produce a message signature that can be used to verify that a particular sender sent a particular message.

Data Bases.

Denning, D. E.:"Secure Statistical Databases with Random Sample Queries." *ACM Trans on Database Systems 5*, 1, (September 1980), pp 291 - 315.

In this paper the author discusses methods of compromising data bases using inference and aggrega-

tion. A method is then presented that can be used, in some situations, to reduce these dangers.

Fernandez, E. B., Summers, R. C., & Wood, C.: *Database Security and Integrity*, Addison-Wesley, Mass., (1981).

The authors of this book review the various aspects of computer security with respect to the effect that these methods have on data bases. They also discuss security techniques that apply particularly to data bases such as the data base machine.

Mathematical Models.

Lampson, B. W."Protection". *Proceedings Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971, pp. 437 - 443, reprinted in *Operating System Review*, *8*, (January 1974), pp 18 - 24.

This is a classic paper that outlines the matrix model for computer protection. This is the basic model for many other models that have followed.

Landwehr, C. E.:"Formal Models of Computer Security". *ACM Computing Surveys 13*, 3 (1981), pp. 247 - 278.

The author, in this paper, presents a good

overview of the major security models that have been used and that have been proposed. These are discussed with respect to their underlying theory as well their usefulness in given situations. The author concludes with a discussion on the direction and needs of research in computer security.

Physical Protection.

Hsiao, D. K., Kerr, D. S., & Madnick, S. E.: *Computer Security*, Academic Press, N. Y., 1979.

In this review of computer security is a chapter devoted to the physical protection of computers and computer installations. This includes commentary on guarding an installation against natural disaster such as fire. As well there is discussion on protection against intruders who use either physical entry or wire taps.

VITA

Surname: WOOLSEY

Given Names: JAMES RICHARD

Place of Birth: VICTORIA, B. C.

Date of Birth: JANUARY 14, 1946

Educational Institutions Attended, with Dates of Entering and Leaving:

University of Victoria, B. C. 1964 - 1968

University of Victoria, B. C. 1971 - 1974

University of Victoria, B. C. 1980 - 1985

Degrees, with Dates and Names of Institutions:

B. Sc. 1968 University of Victoria, Victoria, B. C.

B.F.A. (with distinction) 1974 University of Victoria, Victoria, B. C.

Publications:

Introduction to the Extended Computer, (with David L. Parnas) SEE No. 2, University of Victoria.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis or dissertation (the title of which is shown below) to users of the University of Victoria Library, and to make single copies only for the such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without written permission

Title of Thesis:

AN APPROACH TO COMPUTER SECURITY

Author [redacted]

J. R. WOOLSEY

17/4/85