

AD-A056 901

MITRE CORP BEDFORD MASS

F/G 9/2

MULTICS SECURITY KERNEL VALIDATION: PROOF DESCRIPTION. VOLUME I--ETC(U)

JUL 78 S R AMES & D K KALLMAN

F19628-77-C-0001

UNCLASSIFIED

MTR-3384-VOL-1

ESD-TR-78-148-VOL-1

NL

1 OF 1
AD
A056 901



END
DATE
FILMED
9-78
DDC

AD A 056901

AD No. ~~1~~
DDC FILE COPY

LEVEL II

11

2

ESD-TR-78-148

MTR-3384, Vol. I

MULTICS SECURITY KERNEL VALIDATION:
PROOF DESCRIPTION
VOLUME I

BY S.R. AMES AND D.K. KALLMAN

JULY 1978

235 050

Prepared for

DEPUTY FOR TECHNICAL OPERATIONS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts

DDC
RECEIVED
AUG 2 1978
F



Approved for public release;
distribution unlimited.

Project No. 522N
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-77-C-0001

78 07 27 03 2

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

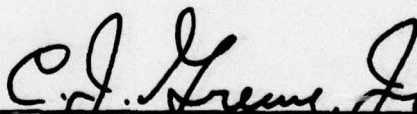
Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.



WILLIAM R. PRICE, Captain, USAF
Technology Applications Division



CHARLES J. GREWE, Jr., Lt. Col, USAF
Chief, Technology Applications Division

FOR THE COMMANDER



STANLEY F. DERESKA, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Technical Operations

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

18 19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD TR-78-148-VOL-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MULTICS SECURITY KERNEL VALIDATION: PROOF DESCRIPTION, VOLUME I	5. TYPE OF REPORT & PERIOD COVERED	
7. AUTHOR(s) S.R./Ames D.K./Kallman	6. PERFORMING ORG. REPORT NUMBER MTR-3384, VOL-1-VOL-1	
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation P.O. Box 208 Bedford, MA 01730	8. CONTRACT OR GRANT NUMBER(s) F19628-77-C-0001	
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Technical Operations Electronic Systems Division, AFSC Hanscom Air Force Base, MA 01731	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 522N	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	12. REPORT DATE JUL 1978	
	13. NUMBER OF PAGES 12 38 P.	
	15. SECURITY CLASS. (of this report) UNCLASSIFIED	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) MULTICS SECURITY KERNELS VERIFICATION		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A major step in the design of a system intended to support multilevel operation is the demonstration that the design is secure. One methodology for demonstrating security is to define (model) a security policy and mathematically verify that the design adheres to that policy. The first step in the design process, after the model, is a rigorous but abstract specification of the system. This paper was written to demonstrate the		

(over)

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

235 454

78 07

27 032

mc

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

20. Abstract (continued)

✓ security of the top level specification for Multics security kernel. Security is demonstrated by a rigorous proof that the abstract specifications correspond to the model. ↗

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ACKNOWLEDGMENT

This report has been prepared by The MITRE Corporation under Project No. 522N. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

ACCESSION for	
NTIS	White Section ☒
DDC	Buff Section ☐
UNANNOUNCED	☐
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
Dist	SPECIAL
A	

PREFACE

This is Volume I of a two volume report that proves that a top-level specification for a Multics security kernel is secure. In this volume an introduction to the proof methodology is given. Volume II contains the actual proof.

All references to sections refer to Volume I. All references to appendices refer to Volume II.

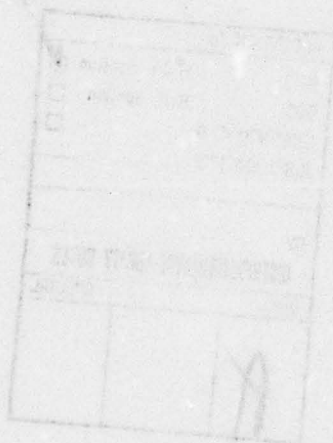


TABLE OF CONTENTS

	<u>Page</u>
LIST OF ILLUSTRATIONS	5
LIST OF TABLES	5
SECTION I INTRODUCTION	6
BACKGROUND	6
Model	7
Specification	8
Implementation Validation	8
FACTORS	9
SCOPE	10
SECTION II DESCRIPTION OF THE PROOF	11
SIMPLE SECURITY CONDITION	11
THE *-PROPERTY	12
Identify Objects and Assign Security Levels	14
Identify the Subjects	15
Identify Accesses	16
Generate the Reference and Case Tables	17
Lemma Generation	17
Prove the Lemmas	19
ACTIVITY, TRANQUILITY, AND ERASURE	19
TRUSTED SUBJECTS	20
SECTION III A COMPARISON OF THREE TOP LEVEL VALIDATION EFFORTS	22
CAVEATS	22
11/45 VALIDATION	22
FIRST ATTEMPT AT MULTICS STORAGE CONTROL	23
VERIFICATION	23
MULTICS TOP LEVEL VALIDATION	24
Syntax Changes	24
Functional Behavior Changes	24
Changes for Validation	25
Errors Found by Validation	25
Cost of the Validation	27

TABLE OF CONTENTS (Concluded)

SECTION IV	REMAINING PROBLEMS	28
	THE IMPLEMENTATION V-FUNCTION-MACROS	28
	THE CLOCK	29
	PROCESS SCHEDULING	29
	DISCRETIONARY ACCESS CONTROL	29
	PROOF ERRORS	30
SECTION V	CONCLUSION	32
REFERENCES		33

LIST OF ILLUSTRATIONS

<u>Figure Number</u>	<u>Page</u>
1 The Labelled Specification of Release_device	13

LIST OF TABLES

<u>Table Number</u>	<u>Page</u>
I Security Level Assignment Table	14
II Reference Table for Release_device	18
III Case Table for Release_device	18
IV Lemmas For Release_device	18
V 11/45 Top Level Validation	23
VI Multics Storage Control Validation	23
VII Specification Changes	25
VIII Design Errors Found by Validation	26
IX Multics Top Level Specification Validation	27

SECTION I

INTRODUCTION

A major step in the technical certification of a system intended to support multilevel operation is the demonstration that the design is secure. One methodology for demonstrating security is to define (model) a security policy and mathematically validate that the design adheres to that policy.

For the past several years, MITRE has addressed the problem of secure multilevel computer operation. The outgrowth of these efforts is a computer security design methodology based on a formal, mathematical definition of a security policy. The first step in construction of an operational system is the specification of an abstract design. The next step is a validation that the design is secure. We demonstrate herein the security of the Multics design specifications, described in a companion paper [1], by formally checking whether the design specifications satisfy the conditions of the model. This complete mathematical proof is a key step in the design methodology for secure systems. Without validation, there is no guarantee of security.

A summary of the validation methodology is presented in Section II of this report. Section III compares several top level validation exercises that MITRE has performed. Section IV addresses open issues in the proof methodology. The actual proof is contained in the appendices in Volume II. The remainder of Section I is devoted to background on the mathematical model and specification language.

BACKGROUND

In response to the need to process multiple levels of classified data in computer systems, the Air Force Electronic Systems Division sponsored efforts to develop an operating system that satisfies the DoD security requirements [2] in a technically verifiable way. The first of these efforts was a panel composed of recognized experts from industry, universities, and government organizations [3]. The panel recommended the use of a reference monitor - an abstract mechanism that controls the flow of information within a computer system. To be the basis for a multilevel secure computer system, this mechanism must meet three requirements:

Completeness -- the mechanism must mediate every access by a subject (active system element) to an object (information container).

Isolation -- the mechanism and its data bases must be protected from unauthorized modification.

Verifiability -- the mechanism must be small, simple, and understandable so that it can be verified to correctly perform its functions, i.e., be consistent with the DoD security policy.

The hardware-software implementation of a reference monitor is called a security kernel.

A kernel satisfies requirement (1) by creating an environment within which all non-kernel software is constrained to operate, and by maintaining control over this environment. The kernel can be thought of as creating an abstract machine or virtual environment on a per process basis. This machine performs instructions from the base hardware (i.e., the hardware on which the kernel software runs) and invokes functions implemented by kernel software. The accesses allowed a user are only those permitted by the hardware-kernel pair.

The requirement for protection against unauthorized modification is satisfied by isolating the security kernel software in one or more protection domains. As an example, a ring mechanism [4] can be used to provide a domain protected from unauthorized modification.

The requirement that the kernel verifiably perform its function stimulated the development of a formal methodology to demonstrate its correctness. A suitable methodology was introduced by Bell and Burke [5]. It Includes:

1. a formal description of the DoD security policy in terms of a mathematical model;
2. a complete description (specification) of the kernel behavior that is proved consistent with the model; and
3. a proof that the kernel is correctly implemented with respect to the description of its behavior used in the second step.

Model

The mathematical model [6] [7] establishes an "inductive nature" of security by demonstrating that security is preserved from one state to another. The security policy is defined by a set of axioms that

limit the allowable access. These axioms were defined to prohibit a program operating on behalf of a user from obtaining data that the user is not entitled to see and from reducing the classification of any information. The enforcement of these axioms by the kernel means that user code need not be validated.

Specification

A proof that the kernel behavior is consistent with the model necessitates a description of that behavior. We describe kernel behavior with a non-procedural specification [8] that we call a "top level specification." The top level specification verified herein is a formal description of an abstract machine intended to implement the reference monitor for the Multics system [9].

The specification language is derived from a technique developed by Parnas [8] and extended by Price [10]. A "Transition specification" is a method of describing an abstract machine. The state of the machine is embodied in a set of primitive "value" functions (V-functions), so called because when invoked, they yield the last value assigned to them. The set of possible machine transitions are defined by a set of "operate" functions (O-functions), so called because they define the operations that can be performed on V-functions. To specify indivisible operations, a third set of functions (OV-functions) are defined that both effect an operation and return a value.

The specification technique allows the use of various simplifications: O-function-macros, V-function-macros, global abbreviations, and local abbreviations. The impact of these are discussed in the Factors section.

Implementation Validation

Once the kernel's behavior is shown to be consistent with the model it is necessary to show that the kernel is correctly implemented with respect to that behavior.

An extensive methodology for showing behavioral identity between a high level user interface specification and a software level specification has been developed by SRI [11]. The pragmatic side of this methodology is the use of a hierarchical approach to design the software in such a way that the proofs required will be divided naturally into simple steps. The mathematical formalism for the proofs is related to work by Milner [12], and by Hoare [13]. We shall not attempt to cover this subject further here.

The validation that the primitives in the code level specification and the corresponding code subroutines are behaviorally identical

would be an application of the generally accepted methodology introduced by Floyd [14]. The code level specification for a function plus the code relations provide the input and output assertions required by the Floyd technique. Whereas a top level specification to model proof can be done by hand there are serious questions about the credibility of a manually produced proof of the code with respect to the code level specification, due to the magnitude of the problem. Automatic program verification tools will most likely need to be employed at this level to produce a credible proof.

FACTORS

The validation methodology that we are using for Multics was first applied by Millen to the specification of a prototype kernel operating on a PDP-11/45 [15]. This initial validation exercise involved a small and simple kernel. The Multics kernel is much larger and more complex. To ease the difficulty in both reading and writing such a large and complex specification, the specifiers extended the specification language to allow for a more concise and compact specification. The basic extensions are the use of strong typing, O-functions macros, and global abbreviations. The specification language was made into a strongly typed language to make implicit many of the argument validation checks. This addition simplifies both the specification and the validation.¹

O-functions macros and global abbreviations support global macro expansion facilities. The use of the macro facilities greatly reduces the complexity that is local to a particular function, thus making that function more readable. However, the validation methodology requires that tables be constructed, for each function, that describe all objects that are read and/or written. To construct these tables, these macros, as well as any abbreviations, must be expanded.² Thus, although the new specification technique increases readability, it also requires that the validators spend more time doing bookkeeping functions on the specification.

¹The derivation of our specification language was influenced by early versions of the SPECIAL specification language [16]. SPECIAL has since been expanded to include automated aids not found in our language.

²V-function-macros are not expanded. They are handled in a special way described in Appendix III.

To ease the bookkeeping burden we employed automated tools to construct the validation tables. The impact of the use of the tools is discussed in Sections III and IV.

SCOPE

This validation covers only the proof that the top level Multics specification obeys the non-discretionary security properties imposed by the mathematical model of security. Discretionary access controls (need-to-know), guaranteed services and priority treatment, properties that must be addressed in an operational environment, were not addressed by this validation. Lower level proofs were also not addressed because lower level specifications or code do not yet exist for the Multics kernel.

SECTION II

DESCRIPTION OF THE PROOF

In this section we describe how the top level specification is validated. As described in the introduction, the validation of non-discretionary security requires proving that certain axioms are preserved by each O-function in the specification. These axioms are:

1. the simple security condition;
2. the *-property; and
3. the activity, tranquility, and erasure principles.

Below, we will discuss the proof of each of these axioms. First each axiom will be expressed in terms of the abstract model entities (subjects, objects). Next, we will give a concrete interpretation of the axiom in terms of the specification entities (O-functions, V-functions, and OV-functions). Finally, we show how the concrete interpretation is proven. Our initial discussion will be for untrusted subjects only.³ Following this initial discussion, we will describe the treatment of trusted subjects.

SIMPLE SECURITY CONDITION

The simple security condition is the direct analog of the DoD requirement that individuals may not be given information that they are not cleared to read. The simple security condition states that "subjects" (active system entities) may only observe less than or equally classified "objects" (information repositories).

When an individual is given a clearance he is charged with the responsibility for maintaining the level of classified information. However, tools that a computer utility may provide cannot necessarily be given the same trust. This is due to: the amount of information that may be compromised; the speed with which the compromise may occur; and the difficulty in detecting or apprehending the violating program.

³A subject is considered trusted if it has been given the ability to violate one of the axioms of the Model.

In order to ensure that a program operating on behalf of a user does not have the capability to reduce the classification of any information, we control the modification of all information.

The *-property, tranquility principle, and erasure principle of the model (discussed in subsequent sections) are the axioms of the model that control modification.

The proof of the modification axioms of the model forms a sufficient condition for proving the simple security condition. This is because information can only be obtained on a computer system by observing one object while modifying a second object. Therefore, the simple security condition need only be independently proven for subjects that have been given the ability to violate one of the models modification axioms.

THE *-PROPERTY

The *-property axiom of the model restricts possible access by stipulating that a subject may only modify an object if that object's security level⁴ is greater than or equal to the subject's security level. The *-property was designed to prohibit a program operating on behalf of a user from reducing the classification of any information.

To simplify the validation process, we use an equivalent version of the *-property that states that: If a subject has read access to an object, O1, and write access to another object, O2, then the security level of O1 is less than or equal to the security level of O2.

To prove the *-property, we must decide what the objects and subjects are in the computer system, what the security levels of all the objects are, and what the read and write accesses in every function are. For each subject, we must list all its read and write accesses and prove that the security levels of all the read accesses are less than or equal to the security level of all the write accesses. The steps involved in the *-property proof are:

1. Identify and assign security levels to all objects.
2. Identify the subjects.

⁴We define a security level to be an element of a lattice formed by a clearance/classification component and a compartmented (formal need-to-know) component.

3. List all the accesses by each subject.
4. For each subject, write down the inequalities needed to prove the *-property. We call these the lemmas.
5. Prove that the lemmas are preserved by all O-functions.

Below, we present a brief explanation of these steps. Since much of the work involved in the *-property proof is strictly mechanical and has been automated, we will comment on the degree of automation used for each step. A further exposition of the techniques is presented in [15].

As a means of illustrating the proof technique, we have chosen to use the O-function `Release_device` (see Figure 1) from the Multics specification as an example.

```

O_function Release_device(device_id)
-----
A  exception
-----
B    ^Secure_read(Cur.access_level, Device(device_id).access_level);
-----
C    ^Device(device_id).owner = Cur_process;

-----
D  effect
-----
E    Device(device_id).owner = "undefined";

```

Figure 1. The Labelled Specification of `Release_device`

The effect of this function is to return ownership rights of a particular device to the system. In order to be successfully invoked two conditions must be met. First the device must have a security level less than or equal to the security level of the process making the call (this allows the process to read the current ownership of the device.) Second the process making the call must be the current owner of the device.

Identify Objects and Assign Security Levels

In our specification, objects, the repositories of information, are the V-function references.⁵ Since the *-property requires a security level assignment to every object, we must therefore assign a security level for every V-function reference.

The security levels of V-functions visible to a kernel user are fixed at the externally specified security level. For instance, contents of a segment must be at the level given when the segment was created. The security levels of internal (hidden) V-functions, however, are chosen by the validators (with the aid of the designers). If we can find a security level assignment that enables us to prove the *-property, we are done. If we can not, then we must change either the security level assignments or the specification, and make another iteration through the proof.

The security level assignments used in this validation are given in Appendix I. These assignments are kept in a security level data base for use in later steps. The security levels of the V-functions used in the Release_device example are provided in Table I.

<u>V-function References</u>	<u>Security Level Assignment</u>
Cur_process	PROC(Cur_process)
Device(device_id).access_level	LOW
Device(device_id).owner	DEVICE(device_id)
Cur.access_level	PROC(Cur_process)
Secure_read(subject_level,object_level)	LOW
exception_return	PROC(Cur_process)
instruction_pointer	PROC(Cur_process)

Table I. Security Level Assignment Table

These security levels were chosen as follows:

PROC(Cur_process) -- The level of the current process is used as a reference point to indicate the security level of all V-function references that are equal to the calling process.

⁵A V-function reference is a V-function plus an assignment of values to all its parameters. In the example statements labeled B, C, and E are V-function references as well as the parameters Cur.access_level, Cur_process and Device(device_id.access_level).

LOW -- The security level of all devices is kept at the lowest security level to facilitate reading the current security level of all devices.

DEVICE(device_id) -- The ownership of a device is obtained and released at the security level of the device.

Identify the Subjects

In the simplest interpretation, all the accesses in a function may be performed by one subject dedicated to that task. This assignment of subjects will work for a function operating at only one level. In a more complicated function, which performs actions at various security levels, the *-property will not be satisfied for this simple subject assignment. For these functions, the accesses must be divided up among various subjects, each of which operates at a single security level [19].

The division is accomplished by first dividing up the function into sections, each of which is given a label. (Note the labels A-E in Figure 1.) The accesses in the function are then decomposed into various cases. A case (see Table II) is composed of two parts - a condition and an effect. For example, the case with condition part $\neg B \wedge C \wedge D$ and effect part E, means that if D is true and B and C are not, then the effect labelled E is considered. By proving the *-property in each possible case we prove the *-property for the entire function [17].

For most functions, we use a standard set of cases determined by a standard set of labels (one for each exception, one for the effect section, one for the derivation section, and one for the exception section). These standard labels are automatically provided and are illustrated in Figure 1. Extra labelling, needed for more complicated functions, is provided by inserting label control characters into the specification. We must ensure when generating the list of cases for an O-function (especially one with a non-standard set of labels) that all possible conditions are considered and that all effects implied by a given condition are used.

As a result of our decision to use a standard set of cases whenever possible, we have had to make individual case divisions for only a few O-functions. Although more cases than were absolutely necessary have been used for some functions, we did not have to make case decisions for the large majority of the O-functions. Thus the overall amount of work has been reduced.

Identify Accesses

In each case the accesses performed in a case are the V-functions referenced in the sections making up the case. These accesses must be divided up into read's and write's. The following rules are used to determine if a V-function is read or written:

1. V-functions that occur in exception or derivation sections are read,
2. V-functions in the hypothesis of an if clause are read,
3. V-functions on the right hand side of an assignment statement⁶ are read,
4. A V-function, not a parameter of a V-function, on the left hand side of an assignment is written, and
5. V-functions that are parameters to a V-function reference are read.

In addition to V-function references, three other references are made due to the semantics of the specification language. Each function of the specification may contain an exception, an effect, and/or a derivation section, depending on the type of function. Each type of section has its own semantics. These semantics must be reflected in the proof in order that any interpretations made by the validators correspond to any made by the designers of the specification and also to any to be made by the implementers.

Our interpretation of the exception semantics is: if a function is called, and one of its exceptions is true, then the caller of the function is notified (by some unspecified method) that the exception occurred. Conversely, if an exception does not occur, then the caller is not notified. In either case the caller of the function can tell whether or not any of the exceptions was true. Therefore, information is always passed from the V-functions referenced in the exceptions to the caller. We have represented this information flow by introducing a new V-function, "exception_return", that is always written in the exceptions section of a function. This V-function is represented by the label assigned to "exception" in Figure 1.

⁶An assignment statement is an equality followed by a semicolon, e.g., $A = B;$, in the effect section, which is not in the hypothesis of any if clause.

Each time a function is called information is read at the level of the caller. We represent this information by including a second V-function, "instruction_pointer". Every effect in an O-function is contingent on a read of this function. This allows us to prove that all writes in the O-function are at a security level greater than or equal to the level of the caller.

The value determined by the derivation section of V-functions or OV-functions is returned to the caller of the function upon a successful call. To account for this, we have included a third V-function, "value return", that is written on every successful call.

Generate the Reference and Case Tables

The reference table (see Table II) identifies for each function in the specification: the V-functions referenced; the type of access made to each of these V-functions; and in which label of the function that access occurred. The generation of the reference tables has been completely automated. All local and global macros are expanded so that the reference tables contain a complete listing of the V-functions referenced.

Once the security level table and reference tables are complete a case table can be generated for each function. These case tables (see Table III) list for each case the security levels of V-functions that are read and the security levels of V-functions that are written. The case tables list all of the information necessary to show the *-property related information flows. The generation of these tables has been completely automated.

Lemma Generation

Lemma generation involves writing down security level inequalities that, if true, will prove that all writes in each case are greater than or equal to the security level of all the reads in the case. If these inequalities can be shown to be true, then the *-property is satisfied. The lemmas for the Release_device function are listed in Table IV. These lemmas are written as two parts - the *-property inequality and the conditions (taken from the condition column) under which the inequality must be true.

Lemma generation has not been automated as yet. So far we have automated only completely mechanical parts of the proof process and not automated those parts closer to theorem proving or theorem checking.

Function: Release_device
Reference Table

Reference	Read in	Written in
Cur_process	B, C	
Device(device_id).access_level	B	
Device(device_id).owner	C	E
Cur.access_level	B	
Secure_read(Cur.access_level, Device(device_id).access_level)	B	
exception_return		A
instruction_pointer	D	

Table II. Reference Table for Release_device

Case Table for Release_device			
Cond.	Effect	Read Levels	Write Levels
B	A	LOW PROC(Cur_process)	PROC(Cur_process)
$\neg B C$	A	LOW PROC(Cur_process) DEVICE(device_id)	PROC(Cur_process)
$\neg B \neg C$	A	LOW PROC(Cur_process) DEVICE(device_id)	PROC(Cur_process)
$\neg B \neg C D$	E	LOW PROC(Cur_process) DEVICE(device_id)	DEVICE(device_id)

Table III. Case Table for Release_device

$DEVICE(device_id) \leq PROC(Cur_process)$	$\neg B$
$DEVICE(device_id) = PROC(Cur_process)$	$\neg B \neg C$

Table IV. Lemmas For Release_device

Prove the Lemmas

As automating the lemma proof would be complicated, the lemmas are proven without machine aid. Thus, the method we have chosen to prove the lemmas is tailored to a hand proof. Future work on theorem provers and checkers should allow much of these proofs to be done by machine. Even with theorem provers, however, the generation of the relations will, most likely, be done by hand.

The first step in proving the lemmas is to minimize the lemmas generated by all the functions into a small succinct set of invariants, called the relations, that give facts about the specification. Second, we list with each lemma the relations needed to prove it. Finally, the relations must be shown to be preserved by each O-function. This involves interpreting the effects section of each O-function and showing that: if no exceptions occur, and if the relations are true when the O-function is called, then the relations are true in the next state. The relations and their proof are given in Appendix VI.

In the Release_device example the first lemma is true because of the condition in label B of the specification. The second lemma is true because of the condition in label C and because the Multics specification obeys a relation that states that if the owner of a device is defined, then the security level of the device and its owner are the same.

ACTIVITY, TRANQUILITY, AND ERASURE

Activity is a property that ensures that whenever a V-function is read it has a well defined value. A V-function is considered active if it is accessible for reading by any subject. To prove this property, we must show that no subject reads an inactive V-function. This is done by giving every V-function reference an activity assignment, a map from a V-function reference to {"active", "inactive"}, and showing that whenever a V-function is referenced it is in the active state. Since most V-functions remain active, this proof is short. The proof is appended to each reference table, when needed, under the heading "Active When Read Because".

The mathematical model assumes a "tranquility principle" [6] that the classification of active objects will not be changed. This prevents inadvertent downgrading of an object by changing its security assignment during a state change. The tranquility principle does not apply for V-functions inactive either before or after an O-function call. The tranquility principle requires showing that whenever a primitive V-function changes its security level during an O-function call, the V-function was either inactive before the O-function call or

it is inactive upon completion of the call. This proof appears at the bottom of the case tables when needed.

In principle, activity and tranquility form a sufficient set of rules. However, in the Multics specification some objects must be reused. Erasure is the property that ensures that all of the information in an object is destroyed before that object can be reused. The erasure principle requires showing that whenever a primitive V-function is activated in an O-function, it must be totally rewritten in that O-function. The erasure proof is contained at the bottom of the case tables when needed.

The activity, tranquility, and erasure proofs are done by hand. Tools have been provided to handle some of the bookkeeping (e.g., filling in the activity status in the correct place once it has been provided). These tools have been kept to a minimum as the tranquility, activity, and erasure proofs are the smallest part of the overall proof.

TRUSTED SUBJECTS

The enforcement of the *-property prohibits a program operating on behalf of a user from reducing the classification of any information. Users, however, need not be constrained by the *-property. As an example, certain users have the authority to alter the classification of a document if the classification of the document is inappropriate to the classification of the information content. These users are trusted not to misuse this authority.

In order to facilitate operations of this type, the kernel provides a mechanism that interfaces the rules of the external environment to rules enforced on uncertified programs. This mechanism is referred to as the "Trusted Subjects" interface. Functions that reside in this interface are exempt from certain rules of the model. Only users or certified software are allowed to invoke these functions.

In the Multics top level specification the System Security Officer (SSO) is the only trusted subject. The Multics specification gives the SSO the capability to, for example, reclassify segments and update storage management (quota control). We must prove that users of the SSO interface can not compromise information that they cannot otherwise read. To do this we will show that the SSO obeys:

1. the simple security condition;
2. a modified form of the tranquility principle called the trusted tranquility principle; and

3. the activity and erasure principles.

In order to simplify the simple security condition proof we actually attempt to prove the *-property and then show that all the *-property violations are not violations of the simple security condition.

The trusted tranquility principle is defined to be: a trusted subject can change the classification of an active V-function from c_1 to c_2 if the trusted subject has a security level greater than or equal to both c_1 and c_2 . This definition allows the SSO to reclassify V-functions that the SSO has the authority to read.

The proof that the SSO obeys the simple security condition, the trusted tranquility principle, and the activity and erasure principles are given in Appendix VII. This appendix will also give a list of any *-property or tranquility violations that can be performed by the SSO, so that he will know what information he is downgrading. We note here also that the SSO O-functions must be shown to preserve the relations. The importance of this is discussed in the next section. The proof that the SSO O-functions preserve the relations is given in Appendix VI.

SECTION III

A COMPARISON OF THREE TOP LEVEL VALIDATION EFFORTS

To date, three separate top level validation exercises have been completed: an in-house PDP-11/45 kernel specification; an early version of the storage control subsection of the Multics specification;⁷ and the full Multics specification contained herein. We have used the results of these exercises to evaluate the relative efforts needed for validation and the effectiveness of top level validation.

CAVEATS

Before making this evaluation, a few caveats are in order. None of the three validation exercises was done in a production environment. This factor contributes both positively and negatively to measured productivity and expended manpower. While the very best people were often available to work on the project (people with high productivity), the research and development environment did not lend itself to the rapid attainment of goals (e.g., it took a few years to develop the kernel design and validation methodologies). Nevertheless, the relative time spent in various portions of the validation exercises does indicate a trend that is very interesting to observe. For if these validation techniques do provide the basis for a sound software engineering discipline, then the trend is one that will eventually predominate.

11/45 VALIDATION

The first validation exercise was done for an in-house 11/45 kernel (see Table V). The vast majority of the time spent on this validation was in developing the methodology. The most significant change in the methodology occurred when the validators discovered that a top level specification that only indicated user visible activity greatly simplified the validation. Upon completion of the new top level specification, the actual validation took about a month.

The 11/45 kernel specification consists of 21 O-functions and 8 mapping V-functions. Only the O-functions required kernel proof. The mapping V-functions were simple enough so that security levels could be easily determined. Although small, the validators concluded that

⁷This validation exercise was not published.

Task	Man/months
Developing Methodology	12
Rewriting Specifications	.5
Actual Proof	1
Documentation	6

Table V. 11/45 Top Level Validation

much of the validation process was mechanical and tedious, and that both the morale and credibility of the provers could benefit from machine aid.

FIRST ATTEMPT AT MULTICS STORAGE CONTROL VERIFICATION

A top level specification for Multics was started at about the same time that the 11/45 validation was being completed. The specifiers were able to use experience gained on the 11/45 work to construct a specification that could be validated. To test out the hypothesis that specifiers could construct this type of specification, a student employee was given the task of validating the storage control section of the specification (see Table VI).

Task	Man/months
Learning Methodology	.75
Developing Tables	.5
Proofs Up to Proof of Lemmas	.5
Documentation	not completed

Table VI. Multics Storage Control Validation

The storage control validation was useful in that it demonstrated that our methodology was simple enough to instruct inexperienced individuals to perform the validation up to the proof of relations. Proving relations still needs a competent mathematician. As in the 11/45 validation, a significant portion of the time was spent mechanically building the necessary tables.

MULTICS TOP LEVEL VALIDATION

For the current Multics validation we used tools to aid the validation process (see Section II). At the onset of the validation, we calculated that it would take two people about a month and a half to complete the validation. We based this calculation on the belief that the specification was virtually correct. As it turned out, the specifications contained a substantial number of errors. To document and control the changes to the specification, we instituted a configuration management procedure and required that each change be accompanied by a change request.

During the validation process we received 58 change requests affecting 70 out of the 104 functions. These changes included the addition of 21 and the removal of 5 functions. In an effort to document the types of changes made, we have divided the changes into four categories: syntax changes; changes to the functional behavior submitted by the specifiers; changes to make the specification conform with the validation methodology; and design errors discovered by the validation process. Table VII reports on the breakdown of these changes.

Syntax Changes

The first category of changes was syntax changes. Fifteen change requests were received of which 3 affected more than two functions. It is interesting to note that over half of these changes were received during the first week the tools were available. The others were scattered around major changes added by the specifiers. The most important errors found in this category were references to functions that had been removed or had been renamed during the time that the specification was being developed.

Functional Behavior Changes

Functional behavior changes refer to those changes submitted by the specification writers that changed the behavior of the specification. We received 9 change requests of this type: 7 of these affecting more than two functions. Over half of the changes received were as a direct result of the validation. We found that the validation was very useful in pointing out portions of the specification whose

	Number of Change Requests	Number of Requests that Effected More Than 2 Functions	Number of Functions Changed
Syntax Changes	15	3	15
Functional Behavior Changes	9	7	50
Changes For Validation	11	3	21
Errors Found by Validation	23	5	31

Table VII. Specification Changes

behavior did not conform to the behavior that the specifiers had intended.

Changes for Validation

The tools we used to construct the validation table required the specification to be in a particular format. In addition, the validation methodology used could be applied more readily if the specification was written in a certain style. This category of changes represents those changes made to the specification to enable us to use our methodology and our tools. In most cases, these changes were minor. Close communication between specifiers and validators can minimize this type of change.

Errors Found by Validation

The most important category of change requests were those that involved actual errors found by the validation process. Included in this category were all: *-property violations, simple security violations, relation violations, and errors that prohibit an implementation of the specification from working. We have further broken down this category of errors in Table VIII to those changes that occurred in the trusted (SSO) portions and those that occurred in the untrusted portion of the specification.

	Definitely Would be Found by Testing	#-property Violations	Simple Security Condition	Violation of Relations
Trusted (SSO) Section	3	none	3	5
Non Trusted Section	2	8	none	2

Table VIII. Design Errors Found by Validation

The most significant errors were found in the SSO portion of the specification. As an example of the type of error found, one error had the result of introducing a large bandwidth storage channel into the specification [18]. The channel occurred when the SSO removed quota from an empty directory. Subsequent calls on the system from untrusted processes could pass information by first creating a file in the directory and then modulating the length of that file. The disturbing characteristic of this path is that it was unnoticed until the validation was performed. The path was found, because one of the SSO functions violated a relation that was used to validate the rest of the specification.

The security methodology being employed guarantees that once in a secure state, the machine will stay in a secure state. However, by violating a relation, the SSO subsection allowed the SSO to place the system into an insecure state. The specification documentation did not indicate that such an action was possible.

An argument was made that this evaluation was unfair, because the path involved an action of the SSO, a trusted individual. However, even though the SSO is trusted, he can make mistakes (especially when the global affects of a function are unknown). We believe that the SSO must be made constantly aware of the global affects to his activity. The only known method to ascertain the global affects of a function is to validate it. The above situation, in which a function had a side effect of causing the system to enter an insecure state, is an example of a function that had unknown global effects. The validation of the SSO subsection precludes this possibility.

Cost of the Validation

The Multics top level validation took 14.5 man months of effort to complete (see Table IX). The actual proof, however, took only about two months, approximately the same time it took to prove the 11/45 top level specification. This figure is especially interesting in that the Multics top level specification had 47 O- and OV-functions compared to the 21 O-functions of the 11/45 validation. In addition, the number of V-functions increased from 8 to 57.

Task	Man/months
Tool Development	2
Refining Methodology	2
Correcting Specifications	2.5
Final Proof	2
Documentation	6

Table IX. Multics Top Level Specification Validation

The primary reason that the Multics specification could be validated in such a short time was the use of the tools. Following each set of change requests, the tools were employed to regenerate the validation table. The use of more sophisticated tools should decrease the validation time even further.

SECTION IV

REMAINING PROBLEMS

Several problems remain in the validation process. These are the implementation V-functions, the clock, process scheduling, discretionary access control, and the possibility that the proof may contain errors.

THE IMPLEMENTATION V-FUNCTION-MACROS

In the specification there are several V-function-macros (Acle_offset, Branch_offset, Dir_page_exists, and Message_offsets) called the implementation V-function-macros, for which no derivation is given. The derivation can be determined only when a specific implementation is chosen. Instead of giving a derivation, the designers have given a list (in the value section of each of the implementation V-function-macros) of V-functions from which the implementation V-function-macros can be derived. It is the task of the individuals performing the verification of the implementation to prove that each of the implementation V-function-macros can actually be derived from the V-functions listed in its value section.

That a specific derivation is not given is no problem from a top level validation point of view. In the validation, for V-function-macros that do not appear in any relations, we need to know only what V-functions are read in determining the value of the mapping V-function.

The problem with the implementation V-function-macros is that the corresponding values are in fact not derivable from the V-functions referenced in their value section. In Multics, the implementation V-function-macros indexed by segment are a complex function of the sequence of actions done to the segment. Since the sequence of actions done to a segment is at the level of the segment, the level we have assigned to these implementation V-function-macros, we do not foresee a security problem. Extra V-functions could, in principle, be added in the top level to record the history, in order that the implementation V-functions be derivable from primitive V-functions.

Adding an extra history V-function should affect a top level proof very little; the level of the implementation V-function-macros would still be at the level of the segment, and every O-function that modified the segment would also modify the history function. This extra write reference would not cause the *-property to be violated,

because we already have, in the O-function, a write at the level of the segment.

THE CLOCK

Between O-function calls in the top level, the clock (the V-function `Current_calendar_time`) is incremented. A security violation can be found in this arrangement if, for example, we had as an effect in the O-function the incrementing of the clock. Every process would both read and write the clock; this is a clear *-property violation.

In general, time or covert [19] channels are a well known way to violate the *-property. We know of no practical methods to eliminate the time channel totally. [18]

We have also had to assume in our proof (See relation R20 in Appendix VI.) that the clock is actually incremented between O-function calls. An implementation must make sure that both the granularity of the clock is fine enough and the number of the bits in the clock word large enough so that the clock is a different value before the next O-function call. This can be easily accomplished with available technology.

PROCESS SCHEDULING

The process scheduling algorithm was felt to be too implementation-dependent to be put in the specification, but it can be shown that a malicious scheduler can compromise information. Even an innocent scheduler can pose a security risk by basing its decisions on execution characteristics under the control of a user process, such as CPU/IO ratio. Some sort of restrictions on the scheduler are needed.

DISCRETIONARY ACCESS CONTROL

Discretionary access control (need-to-know) is not proven here. At most we could have made static assertions such as: If a user does not have read access to a segment then in the next O-function call he can not read the segment; and a user cannot obtain read access without having the necessary permission. Such static assertions could be proven rather easily because the function "Inas" is always called before any accesses are allowed.

On the other hand, no meaningful dynamic assertions (e.g., can a user anytime in the future access a segment) could be made for the Multics specifications. The O-functions `Set_principal_identifier` and `Create_proc` effectively allow a user to become anybody and thus able to access that person's files. Also, a trojan horse could give an unauthorized user access to a file by calling `Add_ACL_Element`, or by copying the file to a file that the unauthorized user could access. Dynamic assertions can be made only in the context of a supervisor program that restricts calling of the above three O-functions and in restrictive domains [20].

In view of the lack of meaningful dynamic assertions and because previous work has not dealt adequately with even the static properties, we have avoided discretionary access control altogether. To tackle the discretionary access control problem, the problem of good static assertions must be solved first. This must be followed by some meaningful dynamic assertions such as a detailed description of how a user might get access to a file. We leave discretionary access control to further investigation.

PROOF ERRORS

Errors must be expected to occur in the validation itself. The best defense against them is to make the validation procedure as standard as possible, and to use software tools to carry out the mechanical parts of it. The weaknesses of the validation then fall into four main categories:

- adequacy of the methodology
- correctness of the software tools
- use of the tools
- proofs of lemmas, relations, etc.

Justifying the methodology and validating the software tools are both nontrivial endeavors, but they need only be done once.

Proofs of lemmas, preservation of relations, and other results that apply to individual functions yield many opportunities for mistakes, but proofs seem to be self-monitoring in the following sense: easy proofs are not a problem, and hard proofs call attention to themselves by reason of their difficulty, so that many mistakes are caught during the proof process.

Software tools can be used incorrectly, especially if there are many steps involved in applying them. A number of errors in the validation were discovered which were the result of failing to use an abbreviation-related tool properly. Because of the independence of the tools from the specification itself, however, these errors were errors only in the proof - the functions involved were actually correct.

The use of abbreviations is generally a delicate matter that must be handled carefully, especially if the semantics of abbreviation expansion involves nesting or is otherwise complicated.

SECTION V

CONCLUSION

In this two volume report we have presented a mathematical validation that proves that the Multics Top-Level Specification adheres to the rules of a mathematical model for security. Validation has been shown to be a relatively inexpensive form of finding errors in, and demonstrating the correctness of, a design. The expenditure of funds in the design phase to correct design problems should significantly reduce the more costly problems found in the implementation or testing phases. We believe that the concept of defining a policy of correctness, followed by a validation that the design meets that policy, is useful in wider applications.

Software tools were used for both initial table generation, and subsequent table generation as the 58 change requests were made. Without these tools, the validation process would have been much more costly.

It is our recommendation that future validations begin earlier in the specification phase. The vast majority of these changes could have been made by the specifiers had the tools been made available as the specification was written.

REFERENCES

1. W.L. Schiller, P.T. Withington, and J.P.L. Woodward, "The Design and Abstract Specification of a Multics Security Kernel," ESD-TR-77-259, Volumes I - III, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, November 1977 - March 1978 (ADA048576, ADA053148, ADA053149).
2. Department of Defense, "Security Requirements for Automatic Data Processing (ADP) Systems," Department of Defense Manual 5200.28, December 1972.
3. J.P. Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, Volume I and II, James P. Anderson & Co., Fort Washington, Pennsylvania, October 1972.
4. R.M. Graham, "Protection in an Information Processing Utility," Communications of the ACM, Volume 11, Number 5, May 1968, pp. 365-369.
5. D.E. Bell and E.L. Burke, "A Software Validation Technique for Certification, Part 1: The Methodology," ESD-TR-75-54, Volume I, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., April 1975 (AD009849).
6. D.E. Bell and L.J. LaPadula, "Secure Computer Systems," ESD-TR-73-278, Volume I-III, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, November 1973 - April 1974 (AD770768, AD771543, AD780528).
7. K.G. Walter, W.F. Ogden, W.C. Rounds, F.T. Bradshaw, S.R. Ames, Jr., and D.G. Shumway, "Primitive Models for Computer Security," ESD-TR-74-117, Case Western Reserve University, Cleveland, Ohio, January 1974.
8. D.L. Parnas, "A Technique for Software Module Specification with Examples," Communications of the ACM, Volume 15, Number 5, May 1972, pp. 330-336.
9. E.I. Organick, The Multics System: An Examination of Its Structure, MIT Press, Cambridge, Massachusetts, 1972.
10. W.R. Price, "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1973.

11. J.M. Spitzzen, K.N. Levitt, and L. Robinson, An Example of Hierarchical Design and Proof, Technical Report 2, Stanford Research Institute, Menlo Park, California, March, 1976.
12. R. Milner, "An Algebraic Definition of Simulation between Programs," Proc. 2nd Int. Joint Conf. on Artificial Intelligence, London, 1971, pp. 481-489.
13. C.A.R. Hoare, "Proof of Correctness of Data Representations," Acta Informatica, 1,44(1972), pp. 271-281.
14. R.W. Floyd, "Assigning Meaning to Programs," Mathematical Aspects of Computer Science, Volume 19, American Mathematics Society, Providence, Rhode Island, 1967, pp. 19-32.
15. J.K. Millen, "Security Kernel Validation in Practice," Communications of the ACM, Volume 19, Number 5, May 1976, pp. 243-250.
16. O. Roubine, and L. Robinson, "SPECIAL (SPECification and Assertion Language): Reference Manual," SRI Technical Report CSG-45, Stanford Research Institute, Menlo Park, California, August 1976.
17. S.R. Ames, "File Attributes and their Relationship to Computer Security," ESD-TR-74-191, Masters' Thesis, Case Western Reserve University, June 1974 (AD A002159).
18. S.B. Lipner, "Comment on the Confinement Problem," ACM Operating Systems Review, Volume 9, Number 5, May 1975, pp. 192-196.
19. B.W. Lampson, "A Note on the Confinement Problem," Communications of the ACM, Volume 16, Number 10, October 1973, pp. 613-615.
20. M.A. Harrison, W.L. Ruzzo, and J.D. Ullman, "On Protection in Operating Systems," ACM SIGOPS Operating Systems Review, Volume 9, Number 5, Proceedings of the Fifth Symposium on Operating Systems Principles, pp. 14-25.