AD-A070 141 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
MULTILEVEL SECURITY FOR INTELLIGENCE DATA PROCESSING SYSTEMS.(U)
APR 79 J COTRELL, C SHU, G SHORT F30602-77-C-0119
UNCLASSIFIED RADC-TR-79-88 NL

1 OF 3
AD
A070141



AD A070141

LEVEL II

RADC-TR-79-88
Final Technical Report
April 1979

# MULTILEVEL SECURITY FOR INTELLIGENCE DATA PROCESSING SYSTEMS

TRW Defense & Space Systems Group

J. Cotrell
C. Shu
G. Short

DDC
RECEIVED
JUN 19 1979
C

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DDC FILE COPY

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

79 06 18 009

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-88 has been reviewed and is approved for publication.

APPROVED: Lee T. Harris

LEE T. HARRIS
Project Engineer

APPROVED: Howard Davis

HOWARD DAVIS
Technical Director
Intelligence & Reconnaissance Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (IRDA), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

# *MISSION*
# *of*
# *Rome Air Development Center*

*RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence ($C^3I$) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.*

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|
| 1. REPORT NUMBER: RADC-TR-79-88 | 2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle): MULTILEVEL SECURITY FOR INTELLIGENCE DATA PROCESSING SYSTEMS. | 5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. |
| | 6. PERFORMING ORG. REPORT NUMBER: N/A |
| 7. AUTHOR(s): J. Cotrell, C. Shu, G. Short | 8. CONTRACT OR GRANT NUMBER(s): F30602-77-C-0119 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS: TRW Defense & Space Systems Group, One Space Park, Redondo Beach CA 90278 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 64750F, 19550115 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (IRDA), Griffiss AFB NY 13441 | 12. REPORT DATE: April 1979 |
| | 13. NUMBER OF PAGES: 211 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same (12) 227 p. | 15. SECURITY CLASS. (of this report): UNCLASSIFIED |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DDC RECEIVED JUN 19 1979 C

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Lee T. Harris (IRDA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Multilevel Security
Computer Security
Security
Security Kernel
Security Policy Enforcement

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This technical report summarizes the major results of a study to determine the feasibility of integrating a security kernel (enforcing DOD security policy) with the IAS operating system installable on the PDP 11/45 and other computers. The report contains sections on the following:

a. Background and Problem Description;
b. Computer System Security Requirements;
c. Current IAS Architecture;

(Cont'd)

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

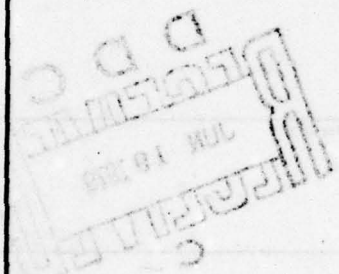
409 637

79 06 18 009

Item 20 (Cont'd)

d. Secure IAS Design Considerations;
e. Secure IAS Architecture;
f. Verification Plan;
g. Development Plan; and
h. Conclusions and Recommendations.

The latter section is tempered by the DARPA sponsored development of KSOS, a "kernalized Secure Operating System" version of UNIX. Possible options are suggested for follow-up study and development effort applicable to the AF Intelligence Community, in regard to the question of remaining with IAS/RSX or migrating to KSOS in all systems or selected ones.



## ABSTRACT

This technical report summarizes the major results achieved in performance of the USAF Rome Air Development Center contract with TRW Defense and Space Systems Group of TRW Inc. The goal of the study was to determine the feasibility of integrating a security kernel which enforces DOD security policy with the IAS operating system to achieve an operational data processing system having verifiable security, comparable performance, and compatibility with current IAS services. The technical report contains a section on the requirements for a secure data processing system for intelligence application, a section on the preliminary design of a security kernel-based IAS system, a section covering a preliminary development plan, and a section covering a verification plan.



| Accession For | |
|---|---|
| NTIS GRA&I | ☑ |
| DDC TAB | ☐ |
| Unannounced | ☐ |
| Justification | |
| By | |
| Distribution/ | |
| Availability Codes | |
| Dist | Avail and/or special |
| A | |

CONTENTS


| | | PAGE |
|---|---|---|
| 1. | INTRODUCTION. . . . . | 1 |
| 1.1 | Background. . . . . | 1 |
| 1.2 | Problem Description . . . . . | 1 |
| 1.3 | Purpose of Study. . . . . | 4 |
| 2. | COMPUTER SYSTEM SECURITY REQUIREMENTS . . . . . | 6 |
| 2.1 | Scope . . . . . | 6 |
| 2.2 | High Level Security Definition. . . . . | 6 |
| 2.3 | Second Level Computer System Security Requirements . . . . . | 7 |
| 2.3.1 | Security Policy Enforcement Requirements. . . . . | 8 |
| 2.3.1.1 | Identification Requirements . . . . . | 9 |
| 2.3.1.2 | Controlled Access Requirements. . . . . | 11 |
| 2.3.1.3 | Isolation Requirements. . . . . | 17 |
| 2.3.2 | Security Surveillance Requirements. . . . . | 19 |
| 2.3.2.1 | Threat Monitoring Requirements. . . . . | 20 |
| 2.3.2.2 | Security Audit Requirements . . . . . | 21 |
| 2.3.3 | Response Requirements . . . . . | 23 |
| 2.3.3.1 | Interception Requirements . . . . . | 23 |
| 2.3.3.2 | Recovery Requirements . . . . . | 23 |
| 2.3.4 | Integrity Requirements. . . . . | 24 |
| 2.3.4.1 | Completeness Requirements . . . . . | 24 |
| 2.3.4.2 | Correctness Requirements. . . . . | 26 |

CONTENTS
(Continued)


PAGE

2.3.4.3 Maintainability Requirements. . . . . 27

2.4 Requirements Traceability . . . . . 31

2.4.1 Security Policy Enforcement Requirements Traceability . . . . . 33

2.4.1.1 Identification Requirements Traceability. . . . . 33

2.4.1.2 Controlled Access Requirements Traceability . . . . . 34

2.4.1.3 Isolation Requirements Traceability. . . . . 38

2.4.2 Security Surveillance Requirements Traceability. . . . . 39

2.4.2.1 Threat Monitoring Requirements Traceability . . . . . 39

2.4.2.2 Security Audit Requirements Traceability. . . . . 40

2.4.3 Response Requirements Traceability. . . . . 41

2.4.3.1 Interception Requirements Traceability. . . . . 41

2.4.3.2 Recovery Requirements Traceability. . . . . 41

2.4.4 Integrity Requirements Traceability . . . . . 41

2.4.4.1 Completeness Requirements Traceability. . . . . 41

2.4.4.2 Correctness Requirements Traceability. . . . . 42

2.4.4.3 Maintainability Requirements Traceability. . . . . 43

CONTENTS
(Continued)


| | | PAGE |
|---|---|---|
| 3. | CURRENT IAS ARCHITECTURE. . . . . | 44 |
| 3.1 | Overview. . . . . | 44 |
| 3.2 | Kernel Executive. . . . . | 44 |
| | 3.2.1 Task Management . . . . . | 46 |
| | 3.2.1.1 Task Memory Space . . . . . | 46 |
| | 3.2.1.2 Task Privileges . . . . . | 47 |
| | 3.2.1.3 Task Scheduling . . . . . | 47 |
| | 3.2.1.4 Inter-Task Communication. . . . . | 48 |
| | 3.2.2 Memory Management . . . . . | 48 |
| | 3.2.3 I/O Management. . . . . | 50 |
| | 3.2.4 System Trap Management. . . . . | 51 |
| | 3.2.5 System Interface. . . . . | 51 |
| 3.3 | Time-Sharing Executive. . . . . | 52 |
| | 3.3.1 Time-Sharing Control Primitives . . . . . | 53 |
| | 3.3.2 Scheduler . . . . . | 53 |
| | 3.3.3 User Interface. . . . . | 55 |
| 3.4 | File System . . . . . | 56 |
| | 3.4.1 File Structure. . . . . | 56 |
| | 3.4.2 File Protection . . . . . | 57 |
| | 3.4.3 File Control. . . . . | 57 |
| 3.5 | System Tasks. . . . . | 58 |
| | 3.5.1 System Startup Commands . . . . . | 58 |
| | 3.5.2 Utility Programs. . . . . | 59 |

CONTENTS
(Continued)


PAGE

3.5.3 System Recovery and File Preservation Procedures. . . . . 61

3.5.4 Device, Volume, and Library Managers . . . . . 61

3.5.4.1 Devices. . . . . 61

3.5.4.2 Volumes. . . . . 62

3.5.4.3 Libraries. . . . . 62

3.5.5 User Authorization and Privilege Functions. . . . . 63

3.5.5.1 User Authorization . . . . . 63

3.5.5.2 Privileges . . . . . 64

4. SECURE IAS DESIGN CONSIDERATIONS . . . . . 66

4.1 Overview . . . . . 66

4.2 Security Considerations. . . . . 66

4.2.1 Security Kernel. . . . . 66

4.2.2 Trusted Software . . . . . 69

4.2.3 Time-Sharing Control Primitive . . . . . 70

4.3 Performance Considerations . . . . . 71

4.3.1 Real-Time Processing . . . . . 71

4.3.2 Interactive and Batch Processing . . . . . 71

4.4 Compatibility Considerations . . . . . 72

4.5 ARPANET Connection Design Considerations . . . . . 73

4.5.1 Secure ARPANET Connections . . . . . 73

4.5.2 Functional Breakdown . . . . . 74

CONTENTS
(Continued)


| | | PAGE |
|---|---|---|
| 5. | SECURE IAS ARCHITECTURE. . . . . | 76 |
| 5.1 | Overview . . . . . | 76 |
| 5.2 | Verified Software. . . . . | 76 |
| 5.2.1 | Security Kernel. . . . . | 76 |
| 5.2.1.1 | Task Management. . . . . | 78 |
| 5.2.1.1.1 | Initiate . . . . . | 78 |
| 5.2.1.1.2 | Terminate. . . . . | 78 |
| 5.2.1.1.3 | Suspend. . . . . | 79 |
| 5.2.1.1.4 | Resume . . . . . | 79 |
| 5.2.1.1.5 | Set-Task-Status. . . . . | 79 |
| 5.2.1.1.6 | Get-Task-Status. . . . . | 79 |
| 5.2.1.1.7 | Set-Event-Flag . . . . . | 79 |
| 5.2.1.1.8 | Clear-Event-Flag . . . . . | 80 |
| 5.2.1.1.9 | Read-Event-Flag. . . . . | 80 |
| 5.2.1.1.10 | Declare-Significant-Event. . . . . | 80 |
| 5.2.1.1.11 | Send . . . . . | 80 |
| 5.2.1.1.12 | Receive. . . . . | 81 |
| 5.2.1.2 | Memory Management. . . . . | 81 |
| 5.2.1.2.1 | Allocate-Segment . . . . . | 81 |
| 5.2.1.2.2 | Release-Segment. . . . . | 81 |
| 5.2.1.2.3 | Remap. . . . . | 82 |
| 5.2.1.2.4 | Set-Segment-Status . . . . | 82 |
| 5.2.1.2.5 | Get-Segment-Status . . . . | 82 |

CONTENTS
(Continued)


| | | PAGE |
|---|---|---|
| 5.2.1.2.6 | Save. . . . . | 82 |
| 5.2.1.2.7 | Get-Partitions. . . . . | 82 |
| 5.2.1.3 | File Management . . . . . | 82 |
| 5.2.1.3.1 | Create-File . . . . . | 83 |
| 5.2.1.3.2 | Delete-File . . . . . | 83 |
| 5.2.1.3.3 | Open-File . . . . . | 83 |
| 5.2.1.3.4 | Close-File. . . . . | 83 |
| 5.2.1.3.5 | Get-File-Status . . . . . | 84 |
| 5.2.1.3.6 | Set-File-Status . . . . . | 84 |
| 5.2.1.4 | I/O Management. . . . . | 84 |
| 5.2.1.4.1 | Create-Device . . . . . | 84 |
| 5.2.1.4.2 | Open-Device . . . . . | 84 |
| 5.2.1.4.3 | Close-Device. . . . . | 85 |
| 5.2.1.4.4 | Get-Device-Status . . . . | 85 |
| 5.2.1.4.5 | Set-Device-Status . . . . | 85 |
| 5.2.1.4.6 | I/O-Read. . . . . | 85 |
| 5.2.1.4.7 | I/O-Write . . . . . | 85 |
| 5.2.1.4.8 | I/O-Function. . . . . | 86 |
| 5.2.1.4.9 | Mount . . . . . | 86 |
| 5.2.1.4.10 | Dismount. . . . . | 86 |
| 5.2.1.5 | System Trap Management. . . . . | 86 |
| 5.2.1.6 | Kernel Interface. . . . . | 86 |
| 5.2.1.7 | Auditing. . . . . | 87 |

CONTENTS
(Continued)


| | | PAGE |
|---|---|---|
| 5.2.2 | Trusted Software. . . . . | 87 |
| 5.2.2.1 | System Boot . . . . . | 87 |
| 5.2.2.2 | Terminal Manager. . . . . | 88 |
| 5.2.2.3 | Discretionary Authenticator . . . . . | 88 |
| 5.2.2.4 | Secure Dialoguer. . . . . | 89 |
| 5.2.2.5 | Directory Manager . . . . . | 90 |
| 5.2.2.6 | System Maintenance. . . . . | 91 |
| 5.2.2.7 | Time-Sharing Executive. . . . . | 92 |
| 5.3 IAS Emulator. . . . . | | 92 |
| 5.3.1 | Task Execution Control Directives . . . . . | 93 |
| 5.3.1.1 | EXIT$ . . . . . | 93 |
| 5.3.1.2 | EXST$ . . . . . | 93 |
| 5.3.1.3 | RQST$ . . . . . | 93 |
| 5.3.1.4 | EXEC$ . . . . . | 94 |
| 5.3.1.5 | SCHD$ . . . . . | 94 |
| 5.3.1.6 | RUN$. . . . . | 94 |
| 5.3.1.7 | CSRQ$ . . . . . | 95 |
| 5.3.1.8 | SPND$ . . . . . | 95 |
| 5.3.1.9 | RSUM$ . . . . . | 95 |
| 5.3.1.10 | ABRT$ . . . . . | 95 |
| 5.3.1.11 | SYNC$ . . . . . | 96 |

CONTENTS
(Continued)


| | | PAGE |
|---|---|---|
| 5.3.2 | Task Status Control Directives. . . . . | 96 |
| | 5.3.2.1 FIX$. . . . . | 96 |
| | 5.3.2.2 UFIX$ . . . . . | 96 |
| | 5.3.2.3 DSBL$ . . . . . | 97 |
| | 5.3.2.4 ENBL$ . . . . . | 97 |
| | 5.3.2.5 DSCP$ . . . . . | 97 |
| | 5.3.2.6 ENCP$ . . . . . | 97 |
| | 5.3.2.7 ALTP$ . . . . . | 97 |
| 5.3.3 | Event Associated Directives . . . . . | 98 |
| | 5.3.3.1 SETF$ . . . . . | 98 |
| | 5.3.3.2 CLEF$ . . . . . | 98 |
| | 5.3.3.3 RDEF$ . . . . . | 99 |
| | 5.3.3.4 RDAF$ . . . . . | 99 |
| | 5.3.3.5 WTSE$ . . . . . | 99 |
| | 5.3.3.6 WTLO$ . . . . . | 99 |
| | 5.3.3.7 EXIT$ . . . . . | 100 |
| | 5.3.3.8 MRKT$ . . . . . | 100 |
| | 5.3.3.9 CMKT$ . . . . . | 100 |
| | 5.3.3.10 DECL$ . . . . . | 101 |
| | 5.3.3.11 WSIG$ . . . . . | 101 |
| 5.3.4 | Informational Directives. . . . . | 101 |
| | 5.3.4.1 GTIM$ . . . . . | 101 |
| | 5.3.4.2 GCOM$ . . . . . | 102 |

CONTENTS
(Continued)


| | | PAGE |
|---|---|---|
| 5.3.4.3 | GPRT$. . . . . | 102 |
| 5.3.4.4 | GSSW$. . . . . | 102 |
| 5.3.5 | Trap Associated Directives . . . . . | 102 |
| 5.3.5.1 | IHAR$. . . . . | 103 |
| 5.3.5.2 | ENAR$. . . . . | 103 |
| 5.3.5.3 | SPRA$. . . . . | 103 |
| 5.3.5.4 | SFPA$. . . . . | 103 |
| 5.3.5.5 | SRDA$. . . . . | 103 |
| 5.3.5.6 | SVDB$. . . . . | 103 |
| 5.3.5.7 | ASTX$. . . . . | 104 |
| 5.3.5.8 | SVTK$. . . . . | 104 |
| 5.3.6 | I/O Related Directives . . . . . | 104 |
| 5.3.6.1 | GCUN$. . . . . | 104 |
| 5.3.6.2 | QIO$ . . . . . | 105 |
| 5.3.6.3 | QIOW$. . . . . | 105 |
| 5.3.6.4 | ALUN$. . . . . | 106 |
| 5.3.6.5 | VSDA$. . . . . | 106 |
| 5.3.6.6 | VSDR$. . . . . | 106 |
| 5.3.6.7 | VRCD$. . . . . | 106 |
| 5.3.6.8 | VRCS$. . . . . | 107 |
| 5.3.6.9 | VRCX$. . . . . | 107 |

CONTENTS
(Continued)


| | | PAGE |
|---|---|---|
| 6. | VERIFICATION PLAN. . . . . | 108 |
| 6.1 | Scope. . . . . | 108 |
| 6.2 | Verification Objectives. . . . . | 108 |
| 6.3 | Verification Issues. . . . . | 109 |
| 6.4 | Development Components for Verifiable Software. . . . . | 112 |
| | 6.4.1 Security Model . . . . . | 112 |
| | 6.4.2 Formal Specification Technique . . . . . | 113 |
| | 6.4.3 HOL for Implementing the Verifiable Software. . . . . | 115 |
| | 6.4.4 Verification Tools and Techniques. . . . . | 118 |
| 6.5 | Verification Methodology . . . . . | 121 |
| | 6.5.1 Validation of the Security Model and Security Requirements. . . . . | 122 |
| | 6.5.2 Verification of Top Level Formal Specifications. . . . . | 124 |
| | 6.5.3 Verification of Lower Level Specifications . . . . . | 129 |
| | 6.5.4 Verification of Trusted Software . . . . . | 135 |
| | 6.5.5 Verification of the HOL Code . . . . . | 136 |
| | 6.5.6 Verification of the Executable Machine Code . . . . . | 139 |
| | 6.5.6.1 Unit (Routine) Testing . . . . . | 140 |
| | 6.5.6.2 Incremental Integration. . . . . | 142 |
| | 6.5.6.3 Security Testing . . . . . | 143 |
| | 6.5.6.4 Euclid Compiler Testing. . . . . | 143 |

CONTENTS
(Continued)


PAGE

6.5.7 Verification Trace. . . . . 144

6.6 Verification Tasks. . . . . 144

6.6.1 Validation of the Security Model and Security Requirements . . . . . 144

6.6.2 Development of Top Level Formal Specification Tools . . . . . 145

6.6.3 Verification of the Top Level Formal Specifications . . . . . 145

6.6.4 Verification of the Lower Level Formal Specifications . . . . . 145

6.6.5 Establishing the Verifiability of the HOL Implementation. . . . . 145

6.6.6 Verification of the Executable Machine Code. . . . . 145

7. DEVELOPMENT PLAN. . . . . 146

7.1 Scope . . . . . 146

7.2 Project Objectives. . . . . 146

7.3 Development Phases. . . . . 147

7.4 Technical Management of Software Development . . . . . 147

7.4.1 Project Progress Reporting and Reviews . . . . . 148

7.4.2 Software Development Problem Surfacing . . . . . 148

7.4.3 Formal Reviews. . . . . 148

7.4.3.1 Preliminary Design Review . . . . . 148

7.4.3.2 Critical Design Review. . . . . 150

CONTENTS
(Continued)


PAGE

7.4.3.3 Test Readiness Review. . . . . 150

7.4.3.4 Functional Configuration Audit . . . . 151

7.4.3.5 Physical Configuration Audit . . . . . 151

7.4.4 Security Management/Configuration Management . . . . . 151

7.4.5 Configuration Management . . . . . 153

7.4.6 Quality Assurance. . . . . 153

7.4.7 Software Standards and Procedures. . . . . 153

7.5 Development Methodology. . . . . 153

7.5.1 Plan the Total Software Project. . . . . 154

7.5.2 Establish a Firm Base. . . . . 154

7.5.3 Develop Secure IAS Incrementally . . . . . 156

7.5.4 Keep Product Documentation Current . . . . . 156

7.5.5 Perform Disciplined Test Planning and Execution . . . . . 158

7.5.5.1 Allocation of Requirements to Test Levels. . . . . 159

7.5.5.2 Test Plans . . . . . 159

7.5.5.3 Test Procedures. . . . . 160

7.5.5.4 Development Level Testing. . . . . 161

7.5.5.5 System Level Testing . . . . . 162

7.5.5.6 Test Reporting . . . . . 163

7.5.6 Formal Verification. . . . . 163

CONTENTS
(Continued)


| | | PAGE |
|---|---|---|
| 7.6 | Work Breakdown Structure. . . . . | 164 |
| 7.6.1 | Work Package 1 - Project Management Support. . . . . | 164 |
| 7.6.2 | Work Package 2 - Development. . . . . | 166 |
| 7.6.3 | Work Package 3 - Testing. . . . . | 166 |
| 8. | CONCLUSIONS AND RECOMMENDATIONS . . . . . | 168 |
| 8.1 | Conclusions . . . . . | 168 |
| 8.2 | Recommendations . . . . . | 170 |
| APPENDIX A | ABSTRACT IMPLEMENTATION MODEL. . . . . | A-1 |
| APPENDIX B | GLOSSARY OF ACRONYMS . . . . . | B-1 |
| APPENDIX C | REFERENCES . . . . . | C-1 |

## LIST OF ILLUSTRATIONS


| | | Page |
|---|---|---|
| Figure 2.4-1 | Computer System Security Requirements Traceability | 32 |
| Figure 3.1-1 | Current IAS Architecture | 45 |
| Figure 5.1-1 | Secure IAS Architecture | 77 |
| Figure 6.5-1 | Verification Methodology Relationships | 123 |
| Figure 6.5-2 | Commutativity of T | 134 |
| Figure 7.5-1 | Software Development Process | 155 |


## LIST OF TABLES

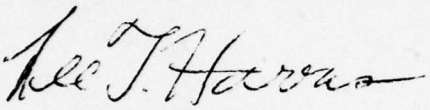

| | | |
|---|---|---|
| Table 3.2.4-1 | File Ownership Categories | 58 |
| Table 4.2-1 | Security Attributes of Kernel Supported Subjects and Objects | 68 |

## EVALUATION

The final report on subject contract delineates the major results of the study to determine the feasibility of integrating a security kernel into the IAS/RSX-11D operating system. These operating systems are in widespread use on PDP-11/45, PDP-11/70 and other PDP-11 series computers within the AF intelligence community.

Since there has been extensive software written uner IAS/RSX-11D and in view of the critical need for multilevel ADP services in many or the IDHS systems this study effort was highly significant.

The report goes one step further in assessing the relative impact of the present DARPA sponsored work in KSOS for UNIX.



LEE T. HARRIS
Project Engineer

# 1. INTRODUCTION

## 1.1 Background

The need for a multi-level security mode in computer systems processing intelligence data arises from the increasing amount of classified and sensitive compartmented data stored in such systems and the need for using data processed in these systems by an increasing community of users not cleared for all compartments and classifications. Many of these intelligence systems have been implemented on the PDP-11/45 computer, using the RSX-11D operating system or its interactive version IAS. Because of the large amount of intelligence applications software written based on the facilities provided by RSX-11D, RADC contracted for this study to determine the feasibility of integrating a security kernel with the IAS operating system to achieve a multi-level security mode, accreditable by the Designated Approving Authority.

Most contemporary resource sharing computer systems are not secure because security was not a requirement of the initial hardware and software design and because there was not a generally accepted computer security definition usable as a basis for a secure design. Since the Air Force has effective means for implementing personnel, physical, communication, emanation, and procedural security, the heart of the computer system security problem is the security certification of the access controls in the operating system and supporting software. AF/ESD sponsored the development by MITRE Corp. of a security kernel providing a basis for a solution to the multi-level security problem. Such a security kernel, the basis for this study, enforces DOD security policy in a demonstrably verifiable manner.

## 1.2 Problem Description

Intelligence systems are being required to manage an increasing amount of sensitive information, including both compartmented and collateral intelligence, which is continually accessed, processed, and

modified, for multiple users having different access authorizations, clearances, and need-to-know. In some planned applications, there is a requirement to integrate data from several security compartments and classifications and to distribute integrated information to users not cleared for all compartments and classifications.

Although it is feasible, in some intelligence environments, to clear all personnel using computers at the highest level, there remains a need to segregate data into compartments and to limit access to each compartment. Thus, in addition to the well-known security classification categories -- TOP SECRET, SECRET, CONFIDENTIAL, and unclassified -- each security compartment must be separately controlled and, within these classifications and compartments, access may be further restricted on a need-to-know basis. As the number of users increases, it becomes impractical to clear all of them to TOP SECRET; therefore, some systems are planned to have both SECRET and TOP SECRET users. Thus each data processing system must be capable of controlling, for users having different clearances and need-to-know, access to the various security compartments and classifications.

Security in current intelligence data processing systems is maintained through use of conventional security measures -- clearance of users to the highest level, physical security of the computer installation and terminal locations, administrative security procedures governing access to and use of the data processing systems, encryption of communications, emanation security -- plus processing limitations and access control software. With these security measures, processing of compartmented information is limited by security regulations -- DOD 5200.28 and DCID 1/16 -- to one or the other of two security modes -- dedicated, where all users with access to the system have both a security clearance and need-to-know for all classified material then contained in the system, and system high, where all users with access to the system have a security clearance for the highest classification and most restrictive type of material contained in the system but at least some users do not have a need-to-know for all classified material contained in the system. In either of these modes, access to the system

by users not cleared for all compartments or classifications is not allowed. Because computer systems processing compartmented data must be either dedicated completely to one or the other of these modes or scheduled for separate processing periods for different compartments, timeliness of processing is limited and operating costs are increased.

Contemporary computer systems do not contain the security controls needed to allow processing in a true multi-level security mode and they generally contain security vulnerabilities allowing evasion of any security controls. The principal vulnerabilities are in the operating system software, which must control concurrent processing and resource sharing for a large number of users, provide user services, and interface directly with the hardware. The complexity of operating systems is such that their structure and functioning has not been well understood, leading to design and implementation errors and hidden capabilities not specified in the design. A further complication is that operating systems are tested by the manufacturer on a limited number of machine configurations, but most users have a configuration differing in some way from the one on which the operating system was tested.

The security kernel, developed originally under AF/ESD sponsorship, provides the basis for solution to the multi-level security problem. A security kernel is a software module interfacing directly with the computer hardware so that all access to computer resources must take place through it. The Kernel checks each access attempt for conformance to security policy. The Security Kernel enforces DOD security policy defined in a mathematical model. The software implementing the Kernel is formally specified and proved to correspond to the mathematical model. DARPA is sponsoring development of KSOS (Kernelized Secure Operating System), an implementation of a security kernel interfaced with the UNIX operating system on the PDP-11/70 computer. Although this system, when its security is accredited, would be usable for intelligence data processing systems, its use would require substantial conversion of current intelligence applications software; accordingly, the Air Force needs to know the feasibility of integrating a security kernel with the RSX-11D operating system and its interactive version.

## 1.3 Purpose of Study

The objective of this study was to determine the feasibility of integrating a security kernel with the IAS operating system of the PDP-11/70 computers to obtain a data processing system having acceptable multi-level security for use in the AF intelligence community. The feasibility is to be determined by a preliminary design which defines the changes needed to integrate the security kernel and IAS in a secure manner.

Specifically, the following tasks were performed.

1) Formulate the security and service requirements for a secure data processing system for intelligence applications,

2) Analyze prototype security kernels and determine changes needed to be integrated with IAS operating system on the PDP-11/70 computer,

3) Analyze the IAS operating system of the PDP-11/70 computer and determine the changes needed for it to be integrated with the Security Kernel,

4) Prepare a preliminary design of a secure data processing system based on integrating a Security Kernel with the IAS operating system,

5) Prepare a security verification plan by which the defined system can be verified,

6) Investigate the security problems arising from connecting the secure data processing system to a network of computers, and

7) Prepare a development plan.

The result of this study is this final technical report. Section 2 of this report contains requirements for a secure data processing system for intelligence application. The preliminary design of the security kernel based IAS is presented via Sections 3-5. Section 3 describes

the current IAS architecture; Section 4 describes the design issues; and Section 5 describes the Secure IAS architecture. Sections 6 and 7 present the verification plan and development plan respectively. Section 8 contains the conclusions and final recommendations of obtaining a secure data processing system for the AF intelligence environment.

## 2. COMPUTER SYSTEM SECURITY REQUIREMENTS

### 2.1 Scope

This section contains the computer system security requirements for the development of a secure system. The computer system security requirements presented herein define the protection capabilities a secure computer system is required to have and are expressed in terms traceable to DOD security regulations and intuitive concepts of security. First, a simple, high level definition is presented (Section 2.2); then, the high level security definition is expanded (Section 2.3) into 29 more detailed requirements. Section 2.4 provides traceability of these requirements to DOD directives.

### 2.2 High Level Security Definition

"Computer system security" is the property a computer system must have to be acceptable (to the Designated Approving Authority) for processing classified information. A definition of this property is needed to design "secure computer systems" and to determine whether or not a given computer system has computer system security. The definition here is based on careful iterative exploration of the meaning of security in other contexts, DOD security regulations, properties of computer hardware and software, and deficiencies in the protection capabilities of current systems.

The high level security requirement is:

- A secure computer system is required to:
  - enforce the applicable security policy in all information transfers under control of the system, and
  - deter, to at least the extent specified by the Designated Approving Authority, threats of unauthorized:

- retrieval of information
- modification of information
- denial of service.

The security policy defines the rules used by the computer system to determine who is authorized access to what information, in what manner, and under what circumstances.

## 2.3 Second Level Computer System Security Requirements

Computer system security requirements define what a computer system must do and what it must not do to be acceptable for processing classified information. They, therefore, form a prescriptive definition of a secure computer system, i.e., a definition which can be used as a basis for evaluating the security of individual computer systems. Intuitively a secure computer system is one that is resistant to unauthorized access to system resources (hardware, operating system, user programs, and data). The requirements define the protection capabilities a system must have to be acceptably resistant to this threat. For clarity and analysis purposes, it has been found beneficial to structure the requirements into four major categories and ten subcategories. They are as follows (the number in parentheses is the section number in which the subcategory is discussed in detail):

1. Policy Enforcement (2.3.1)
   - Identification. Defining, determining, and assuring the correct identity of users and system resources (2.3.1.1)
   - Controlled Access. Determining whether or not a user requesting access to information is authorized to have access to that information as determined by a set of rules which state the conditions under which access may be granted and the access privileges granted (2.3.1.2)

- Isolation. Restricting processing to the system resources to which access was granted (2.3.1.3)

2. Security Surveillance to Detect Attacks (2.3.2)
   - Threat Monitoring. Active monitoring of activity patterns for indicators of attacks (2.3.2.1)
   - Security Audit. Logging of security related events for post facto analysis (2.3.2.2)
3. Response to Detected Attacks (2.3.3)
   - Interception. Stopping of attacks before serious damage can be inflicted (2.3.3.1)
   - Recovery. Actions to restore the system to normal operation (2.3.3.2)
4. Integrity of the Security Controls (2.3.4)
   - Completeness. The property of the security controls being deterministic for every access request (2.3.4.1)
   - Correctness. The property of the security controls functioning as intended by the system designers (2.3.4.2)
   - Maintainability. The property of the security controls permitting update and repair (2.3.4.3).

### 2.3.1 Security Policy Enforcement Requirements

The security policy enforcement requirements are described in detail in the following subsections. Each category is given a mnemonic prefix: identification requirements have been identified by the prefix "ID"; controlled access requirements by the prefix "CA"; and isolation requirements by the prefix "IS."

### 2.3.1.1 Identification Requirements

ID1 Each computer system user shall be assigned one or more unique identifiers before his usage of the system. The list of assigned user identifiers defines the set of authorized users of the system.

Intent

A computer user is a person, generally identified by his (or her) proper name. Since more than one person may have the same proper name, the computer system is required to assign a unique identifier, such as his surname followed by an instance number, to each authorized new user. Administrative reasons may make it desirable to assign more than one identifier to a given user.

ID2 Each time the user uses the system, he shall be required to declare his identity upon initial entry. The system must verify that his identifier is on the list of authorized users before he is allowed any further use of system resources.

Intent

A secure computer system requires the user to "declare his identity", i.e., input his identifier, and the computer system then verifies that his identifier is on the list of authorized users "before he is allowed any further use of system resources" - i.e., positive verification results in tentative acceptance (pending authentication of his identity as required by ID3) as an active user.

ID3 Following establishment of a user's identity, that identity shall be authenticated so that the probability of erroneous identification is less than some specified number specified by the Designated Approving Authority for the system.

Intent

Authentication of a user's identity is interpreted to mean assignment of a second identifier to each authorized user and requiring the user to present the second identifier after his first identifier has been determined to be on the list of authorized users, before allowing his further use of system resources. This second identifier will generally be syntactically different from the first identifier. It may be a password. It could be a character string recorded on a computer readable identification card. It could be a computer coding of a finger print, etc.

ID4 Each controllable system resource shall be assigned a unique identifier, including all:

- tasks or processes,
- files and memory, and
- I/O devices.

Intent

Correct referencing of computer system resources can only be assured if they are assigned a unique identifier. The symbolic identifiers that the user references a file may not necessarily be unique but this name must be mapped into a unique internal identifier.

ID5 The identity of each controllable system resource shall be authenticated at the initiation of system operation, and for those resources which are removable or remote, before access is granted.

Intent

Authentication of the identity of a computer system resource means verification that the resource accessed when a

resource identifier is referenced is the actual resource denoted by that identifier.

Since a secure computer system is required (by physical security and administrative security requirements) to be located in a physically secure facility with access to it subject to administrative security controls, identity authentication of fixed resources would need to be carried out only at the time of initial security certification or accreditation and at such intervals as the Designated Approving Authority may designate. For removable resources, such as tapes and disk packs, the identity authentication should be carried out when the resource is installed on the system and during system restart.

2.3.1.2 Controlled Access Requirements

CA1 User's access to systems resources shall be subject to rules which implement the applicable security policy and which, in combination with authorization data, specify for all user-process-resource combination:

- Whether or not the combination is allowed
- Under what conditions it is allowed
- The access privileges allowed.

Intent

For the Department of Defense (DOD) computer systems, the rules which the computer system must meet to process multi-level security data are documented in E.O. 11652, DOD 5200.1-R, DOD 5200.28, DOD 5200.28-M, AFR 205-1, and AFR 300-8. These directives and regulations specify a set of access rules based upon a

stratified hierarchy of four categories: unclassified, CONFIDENTIAL, SECRET, and TOP SECRET. Additionally, there are a number of compartments, and finally need-to-know.

Users access system resources through processes. A process is a program or a unit of a program which is in execution or has been made immediately available for execution. Hence controlled access acts to constrain what processes can do on behalf of specific users. A process accesses a resource by referencing it - i.e., by naming the resource identifier together with an access verb. Access to a resource will involve an access privilege:

- Read the data from the resource
- Write data into the resource
- Execute the code in the resource

or some combination of these three basic privileges. Access to a resource referenced in a process may involve calling on a sequence of intermediate processes - e.g., the operating system services required to carry out a request for data on secondary (disk) storage.

The set of resources which a process can access is called its domain. This is defined by the possible values of parameters which must be set in order for a process to carry out its requested function. The set of possible outputs of a process is called the range of the process. The rule by which a process produces values in its range from values in its domain is called the function of the process. Access control of a process is exercised by limiting its domain, range, or function or some combination of them.

Access control is dependent on who the user is, what the process is, and what resources are to be accessed. It is therefore a ternary relation which specifies the allowed user-process-resource combinations. The access control rules cannot be expressed completely in terms of only two of the three combinations. Each process must therefore be associated with the user who initiated it when the access rules are applied. This association must be preserved whenever an access request requires a sequence of processes for its fulfillment.

To meet DOD security regulations, there must be a set of rules for the control, classification, declassification, and manipulation of data within the computer system users. These rules should be stated in terms of user-process-resource combinations. For example, for a given user the user-process-resource combination defines the system resources this user is allowed to access with this process, and what his access privileges are. All other accesses are disallowed. A system is considered secure if the user-process-resource combination is always enforced and is in accordance with the security policy.

The abstract implementation model (AIM) presented in Appendix A formalizes these rules for Secure IAS.

CA2 Any information leakage path shall be identified and its bandwidth determined. An engineering tradeoff study shall weigh the cost of closing it against the exposure by leaving it.

Intent

Classified information can move from one security level to another along an information leakage path in two different ways. The first way is that a user may be able to infer

classified information by observing certain system variables. The second way is that a process at one security level may be able to make system calls which change the contents of a system variable observable at a lower security level. This potentially allows one process to communicate with another process and possibly pass classified information provided the two processes operate in collusion. The bandwidth of the indirect channel is determined by the rate at which the sending process can "modulate" a variable which a receiving process can detect.

The design of a secure system should identify all indirect storage channels for passing information. Those engineering tradeoffs necessary to close indirect channels which involve a high performance penalty should concentrate on those channels which can pass information by indirect inference and then on minimizing the bandwidth of those channels which require collusion. Those channels which are not closed by system design should be specifically identified and consideration given to controlling the vulnerabilities through auditing the software run on the system and monitoring the system to detect unusual patterns of usage. Indirect channels have been further broken down as follows:

Legitimate - Knowledge of a higher classified state by a lower classified state specifically allowed for in the system, e.g., notifying a lower level user that mail has been received at a higher level. All such channels should be handled by trusted software.

Covert (Timing Dependent) - Knowledge gained through channels not intended for the passage of information. (e.g. varying the paging rate or CPU usage of a process.) Such channels are normally of low bandwidth and blocking them normally involves a serious performance

penalty. These channels should only be blocked when one or both of the above is not true. A security monitoring feature may be able to detect abnormal patterns of usage which may be associated with attempts to pass information along covert (timing dependent) channels.

Storage - Knowledge gained through system variables, e.g., testing the status of a tape drive. It is possible to systematically identify all possible storage channels by verification of the formal specification. Once the channels have been identified it is possible to do an engineering tradeoff study weighing the cost of closing a path through trusted software against the dangers of leaving it unblocked.

CA3 Each computer system user and each controllable system resource shall have sufficient authorization data declared before entry to (or already established in) the computer system so the access rules can be applied before any system resource is used.

Intent

The computer system must be supplied with a complete list of authorization data about all users and resources to be protected, the specific access rules by which a user may access a protected resource, and a description of the system configuration. This list of authorization data defines the specific implementation of the rules for the installation.

For DOD, the authorization data is necessary to verify each user's clearance level (TOP SECRET, SECRET, OR CONFIDENTIAL) as well as the special categories and need-to-know restrictions. For each protected resource, the authorization data defines the classification level and special categories. The access privileges define the authorized relationships

between users and resources. One user may have read-only access privileges to a resource, while another may have both read and write access privileges.

The description of the system configuration and arrangement includes such information as the unique identifier of each addressable device and the communication path(s) to that device. This data is required to ensure not only that the user has been authorized to access the resource, but that the resource exists and the source/destination of that data is to a resource that has been administratively cleared to the required classification level.

CA4 Each process shall be associated, when the access rules are applied, with the user who initiated it.

Intent

The users of the computer system must be accountable for their usage of the computer services. Consequently, when sensitive information is created, processed, disseminated, or destroyed in a computer system, the system must maintain accountability while the information is under its control. In a timesharing, multiprogramming computer system, accountability can only be maintained if the user-process-resource relationship is established and maintained.

CA5 The computer system shall have a method for creating, deleting, and changing the authorization data. This method will be subject to the access control rules specified by CA1.

Intent

The authorization data, as described by requirement CA2, reflects the specific implementation of the rules at an installation. This data changes as new projects are initiated or terminated. This means that new resources

must be controlled, and different people may have access to classified resources. As a consequence, the authorization data must be updated in a timely manner to reflect the new user-process-resource decisions.

The mechanism which updates the authorization data is commonly referred to as an authorization mechanism. This mechanism creates and maintains the data the access control mechanism and authentication mechanism use to grant or reject access to system resources and the computer system itself.

The information in the authorization data base is communicated to the computer system by special commands which permit the definition of users and system resources to the system and authorization of access to the system resources. New resources may be added, old ones deleted, and definitions and access privileges modified. A list command may be available to list the users, system resources, and relationships between the users and system resources.

CA6 Each computer system user shall be notified of the applicable security attributes for all data furnished to the user by the system.

Intent

The computer system must notify the user of the security level of the data furnished to the user so that the user can affix proper classification identifiers if necessary, and more importantly, so that the user is aware of the security classification in order to handle the data properly.

### 2.3.1.3 Isolation Requirements

IS1 The computer system shall restrict access of each active process to the domain authorized for the user-process-resource combination.

Intent

Every process operates within a domain. It is useful to consider two types of request, intra-domain and inter-domain. A process requesting access to a file is an example of inter-domain request. This request goes through the file system which checks the privileges of the process against the protection status of the file according to the security policy. This type of request is the intent of requirement CA1. In general, all inter-domain requests are mediated by a protection monitor which serves as a gate between the resource and requesting process and which enforces the security policy on every request. Therefore, the protection monitor is located somewhere in the linking path from the domain of the process to the resource in order to interrupt access attempts.

Intra-domain requests may be considered as requests within the authorized address space which will be executed without the intercontrol of the protection monitor. Requests are kept within the authorized address space by means of protection walls such as mapping registers, bounds registers, etc. The protection walls are also responsible for the protection of the protection monitor and hence protect it against tampering.

In the case of accidental malfunctions, isolation features serve two additional purposes: diagnostic and error isolation. Errors often result in the violation of one of these features, which can then aid in locating the error and isolating it so it cannot harm other components of the system.

The hardware isolation features include the following: execution state variables, separation of user and executive modes, memory bounds registers, memory mapping registers, and read, write, and execute access control.

IS2 Any data that is not to be retained at the completion of a process as controllable system resources shall not be determinable by any other process.

Intent

Recovering residual data must be considered as a potential means of obtaining classified data. Provisions which ensure that classified data or critical elements of the computer system do not remain as accessible residue after a job has been completed are therefore necessary. The specific areas of concern are main storage, direct access storage, tape storage, and video screens. Any device that has a buffer associated with it must have that buffer cleared or overwritten before reuse. Terminal devices which have no provision for suppressing the printing of the password or other sensitive data must also be considered.

IS3 There may be a need for data encryption.

Intent

Data encryption may be required by a number of applications and special devices. Generally, tools for the following should be considered:

- Data being transferred to storage media within the system. The handling of certain sensitive data objects may require this feature.
- Data being transferred outside the system, particularly over communications lines.

### 2.3.2 Security Surveillance Requirements

The security surveillance requirements are described in detail in the following subsections. Each subcategory is given a mnemonic prefix; threat monitoring requirements have been identified by the prefix "TM"; and security audit requirements by the prefix "SA."

### 2.3.2.1 Threat Monitoring Requirements

TM1 There shall be dynamic monitoring of security related events between the user and computer system as required by the Designated Approving Authority for any misuse, and for real-time notification upon detection of any misuse.

Intent

For dynamic monitoring of information indicative of illegal accesses by system penetrators, it is important that the computer system have a means for immediate notification if the penetrators are to be caught. Information analysis and decision criteria are necessary to provide such notification. Thus, after some number of unsuccessful log-on attempts, the system operator or security officer may be notified by the printing of a message or sounding of an alarm, or both. He can then take appropriate actions to investigate the user, the terminal, and the area where the log-on attempt was conducted. Similar notifications can be provided for illegal user file, program, hardware, and operating system accesses.

TM2 There shall be dynamic monitoring of specific protection features for their correct operation, for detection of any malfunction leading to security compromise, and for real-time notification upon the detection of any malfunction.

Intent

Any hardware and operating system malfunctions in the protection features must be detected to prevent possible security compromise. The malfunctions can arise either by unauthorized modifications to the control features during system maintenance or by accidental malfunctions such as hardware errors. Therefore, the specific protection

mechanisms should be analyzed to identify potential failure modes. The failure modes must either be eliminated, or independent failure detection mechanisms must be incorporated to prevent compromise due to a protection mechanism failure. Further analysis needs to be performed to determine if the detection mechanisms actually detect the identified failures. A failure detection mechanism must assume that a minimum of two independent and simultaneous failures occur before a compromise is possible.

#### 2.3.2.2 Security Audit Requirements

SA1 A computer system security log shall be generated and maintained describing security-related events in sufficient detail to establish individual accountability and damage assessment, as required by the Designated Approving Authority.

Intent

System resource abuse or potential abuse can only be identified from information relating to the access of the resource and the time of access. Thus, a user's first access to a time-sharing system (which may be regarded as access to a terminal hardware component and the operating system) can be recorded at system log-on time. Such information as the terminal used and the time of day can also be recorded. A user's subsequent accesses of hardware resources, key portions of the operating system, and the information managed by the system, can all be recorded to obtain a record of accountability in case of abuse of those resources. Without this record, there would be no way of identifying the source of abuse, such as disclosure, modification, or destruction of system-managed information as well as disruptions of the operating system and degradation of system performance.

The types of component accesses and associated information which can be recorded in a security audit log include:

a. User log-ons, including identifiers used, time of day, and terminal.

b. Hardware resource accesses and usage, including time, type of device, and location.

c. User program and file accesses, including identifiers, classification, storage media identifiers, and time.

d. Security officer accesses, including type of access, entity accessed, and time.

Although they may not be dynamically recorded, operator and maintenance activities involving system accesses should also be recorded in the security audit log. Other important entries include periods of discontinuity such as system startup, shutdown, and recovery.

SA2 The computer system shall provide programs for generating appropriate security-related reports from the system log which can be used to establish individual accountability and damage assessment.

Intent

Critical information recorded in the security audit log can be more easily ascertained by the use of special programs which automatically gather such data and generate appropriate security reports, which are important for the ex post facto analysis of security-related events occurring in the system throughout its operational phase. While the format and content of such reports may be largely installation-dependent, there are fundamental types of information of common interest to most installations. Other criteria for the design of report generation programs and report contents may include

such factors as DOD directives and regulations governing the operation of a particular installation. Any significant security events not recorded on the security audit log or susceptible to automatic report generation should be recorded and reported manually.

### 2.3.3 Response Requirements

The response requirements are described in detail in the following subsections. Each subcategory is given a mnemonic prefix: interception requirements are identified by the mnemonic prefix "IT," and recovery requirements by the prefix "RV."

#### 2.3.3.1 Interception Requirements

IT1 For each hardware or operating system security surveillance mechanism, there shall be one or more compensatory actions.

Intent

For any actual or potential security incident, there should be a corresponding compensatory action or actions to stop the penetration. Otherwise, there would be no deterrent to malicious users who could continually attempt to penetrate the system or deny the services of the system to authorized users.

#### 2.3.3.2 Recovery Requirements

RV1 For appropriate hardware component failures and operating system control component failure, there shall be a timely and operationally useful recovery mechanism.

Intent

If a system is to be secure, it must compensate for incorrect behavior. This will necessitate recovery. Some operating system control components will not have a corresponding recovery routine because of cost considerations. However, even when it is not possible to return quickly to an operational state, recovery mechanisms and routines provide an orderly shutdown of the computer system in addition to resource protection and purging of temporary sensitive data.

### 2.3.4 Integrity Requirements

The integrity requirements are described in detail in the following subsections. Each subcategory is given the same mnemonic prefix "IN."

#### 2.3.4.1 Completeness Requirements

IN1 The controls shall have a defined response (completeness) for every possible attempt to:

- Enter the system
- Request access to system resources
- Exceed, during processing, the boundaries of the resources authorized.

Intent

A computer system's enforcement security controls must be completely defined. They must receive no input for which the response (effect) is not defined in the specification. If there is an input for which the effect is not defined in the specification, the action which takes place when that input is presented to the controls may not conform to the security requirement on the controls. The action caused by such an input could, therefore, be exploited by a penetrator. There must be no way for a user to gain entry to the system without having his identity

established and authenticated. There must be no way a process can request access to resources without having the access controls applied to that request. There must be no way a process can be granted access to specific resources without having isolation controls applied.

Incomplete specification of the security controls of current operating systems is the source of much of their security problems.

IN2 For every possible security event of the type defined for security audit by the Designated Approving Authority, there must be a defined recording procedure.

Intent

A computer system's security audit facilities must be completely specified. There must be no security event for which the recording procedure is not specified in the specification. The term "event" in this context means user initiated events such as log-on, file access, start-up, etc.

IN3 For every possible security activity pattern of the type defined for threat monitoring by the Designated Approving Authority, there must be a response.

Intent

A computer system security response mechanism must be completely specified. There must be no security activity pattern of the type defined for threat monitoring that the corresponding response(s) are not completely specified.

IN4 The security controls (identification, controlled access, isolation, surveillance, and response mechanisms) shall be applied at all times.

Intent

Security must be enforced at all times, although computer systems do not operate continuously. There are periods of discontinuity such as scheduled system maintenance, start-up, shutdown, recovery, and simply idle periods. Idle periods must be handled by physical and administrative security. This leaves basically two cases: periods of continuity and periods of discontinuity.

For periods of continuity, this requirement implies that there is a verifiable method of identifying the source of each request and that the controlled access and isolation controls in combination completely mediate all attempts to access system resources. However, every access does not need explicit validation with the access rules. When a process is executing in some domain, it can exercise the capabilities which belong to that domain and are commonly validated by the isolation features as indicated by requirement IS1. When control passes from one domain to another, the capabilities given to the process will change as indicated by requirement CA2. That is, when a process requests a service of the computer system, the enforcement mechanism of the computer system must be invoked at its entry point. Then the request is either granted or rejected based on the security policy and the authorization data associated with the request.

During periods of discontinuity, which include startup, restart, shutdown, recovery, and maintenance, the computer system must purge sensitive residual data and protect the system resources.

2.3.4.2 Correctness Requirements

IN5 For every request of the security controls (identification, controlled access, isolation, surveillance, and

response mechanisms), the actual response must correspond to that specified in the design specification.

Security requirements are much more stringent than ordinary software verification requirements because:

- Integrity flaws missed in normal verification efforts may provide real security threats. Even though activated only by rarely used inputs, presenting for normal software only a minor operational nuisance, a single flaw, once it has been identified by a penetrator, can be exploited again and again until it is corrected.
- "Hidden" functional capabilities present in actual software in addition to required functional capabilities may be exploitable by penetrators.

Integrity flaws and hidden functional capabilities may be introduced at any level in the development process; hence, the verification at each level must be exceptionally thorough.

IN6 A validated bootstrap loader shall properly initialize the operating system and its security related data bases.

Intent

First, the bootstrap loader must assure that only the certified copy of the operating system is loaded. Secondly, for the operating system to operate correctly it must be properly initialized. This is possible if its initial state is consistent with the initial state prescribed by the system specifications.

### 2.3.4.3 Maintainability Requirements

IN7 The security controls shall be maintained only by procedures, acceptable to the Designated Approving Authority,

which preserve system compliance with the security requirements. Those which are not controlled by the access controls shall not be activated while the system is being used operationally for classified processing.

Intent

Maintenance programs are provided for hardware maintenance personnel and operating system maintenance personnel to keep the system's hardware and software operating properly. There are several concerns here. First, there is the issue that these programs/tools may have excessive privilege to perform their required function. Another issue is the introduction of errors into the protection mechanisms of the system, whether design errors or implementation errors deliberately introduced, such as trapdoors. In particular, provisions are needed for:

1. Constraining maintenance personnel to maintaining specific resources using only specified programs/tools.
2. Determining correct operation of the hardware components.
3. Controlling changes to the software routines which comprise the identification, controlled access, isolation, surveillance, and response mechanisms.

IN8 There shall be appropriate mechanisms and procedures for secure generation, backup and restore of copies of on-line data.

Intent

Secure generation, backup, and restoring of copies of on-line data is essential to security. Some of the more important points to consider follow.

- Generation of backup copies, by its frequency, should attempt to maximize the possibility that the current version of a data object has a backup. A common, and relatively sound, method for doing this is to periodically (where the period may be varied) generate backup copies of all data objects created or modified since the last periodic generation.
- Storage media containing backup copies of data objects should receive special care and handling, since it represents a central collection point for a vast amount of potentially sensitive information.
- It should be possible to easily locate a desired backup copy of a specific data object without compromising the privacy or security of any user. Too often, printed directories of backup storage media are formatted and handled in a way not consistent with good security practice.
- Users should, under special circumstances, be able to avoid the standard procedures for backup copying and retrieval, and to substitute their own. Certain particularly sensitive applications may demand a more complete isolation and tighter control than standard procedures can offer.

IN9 Strict configuration control shall be applied to the computer system security mechanisms.

Intent

Security configuration control is essentially the ability to ensure that all changes to the computer system are accounted for and to verify that these changes do not impair the system security mechanism. Any erroneous change to a security mechanism, whether hardware or software, can have serious

consequences. Thus, management of change to security mechanisms must take into account all versions of the computer system, not just the initial or starter system. Tools and procedures must be provided for maintaining the configuration control of the system; among these are the following:

1. Identification of the security-sensitive modules of the operating system and the security mechanisms in the hardware
2. Identification of all changes that impact security mechanisms
3. Inspection of new releases, such as binary object program comparison software routines
4. Special maintenance procedures for distributing, reporting, and applying changes to security-sensitive modules
5. Verification that the operational system has not changed.

## 2.4 Requirements Traceability

The computer system security requirements were developed as English statements representing DOD directives/regulations and security concepts applicable to computer processing as shown in Figure 2.4-1. The computer system security requirements define the protection features Secure IAS is required to have. The security model only represents in mathematical form the requirements (security policy enforcement) defining the security policy enforced by the security kernel.

The use of security kernel technology results in a design where it can be demonstrated that security compromise is prevented based on the structure of the software modules. The approach taken to satisfy the computer system security requirements is embodied in the reference monitor concept. The reference monitor concept arbitrates each reference made by each program in execution by checking the attempted access against a list of accesses authorized for that user.

The reference monitor is implemented as a reference validation mechanism. The validation of a reference has two contexts – a determination of whether access to a particular resource is permitted and a mechanism that enforces the access modes permitted. The PDP-11/70 hardware provides mapping registers to implement the second part. The PDP-11/70 also satisfies the tamperproof requirement of the reference monitor concept by providing three hardware states.

Security kernel technology must begin with a formal statement of security policy. A security model such as that presented in Appendix A is such a basis. The model provides a formal representation of information handling requirements found in DOD directives and regulations in terms of abstract entities that access information and entities that contain information. The model therefore provides a convenient set of abstractions for defining the security policy.

The security kernel therefore implements the reference monitor concept on the PDP-11/70. Moreover, it provides a subset of operating system functions that provide controlled sharing in a DOD environment.

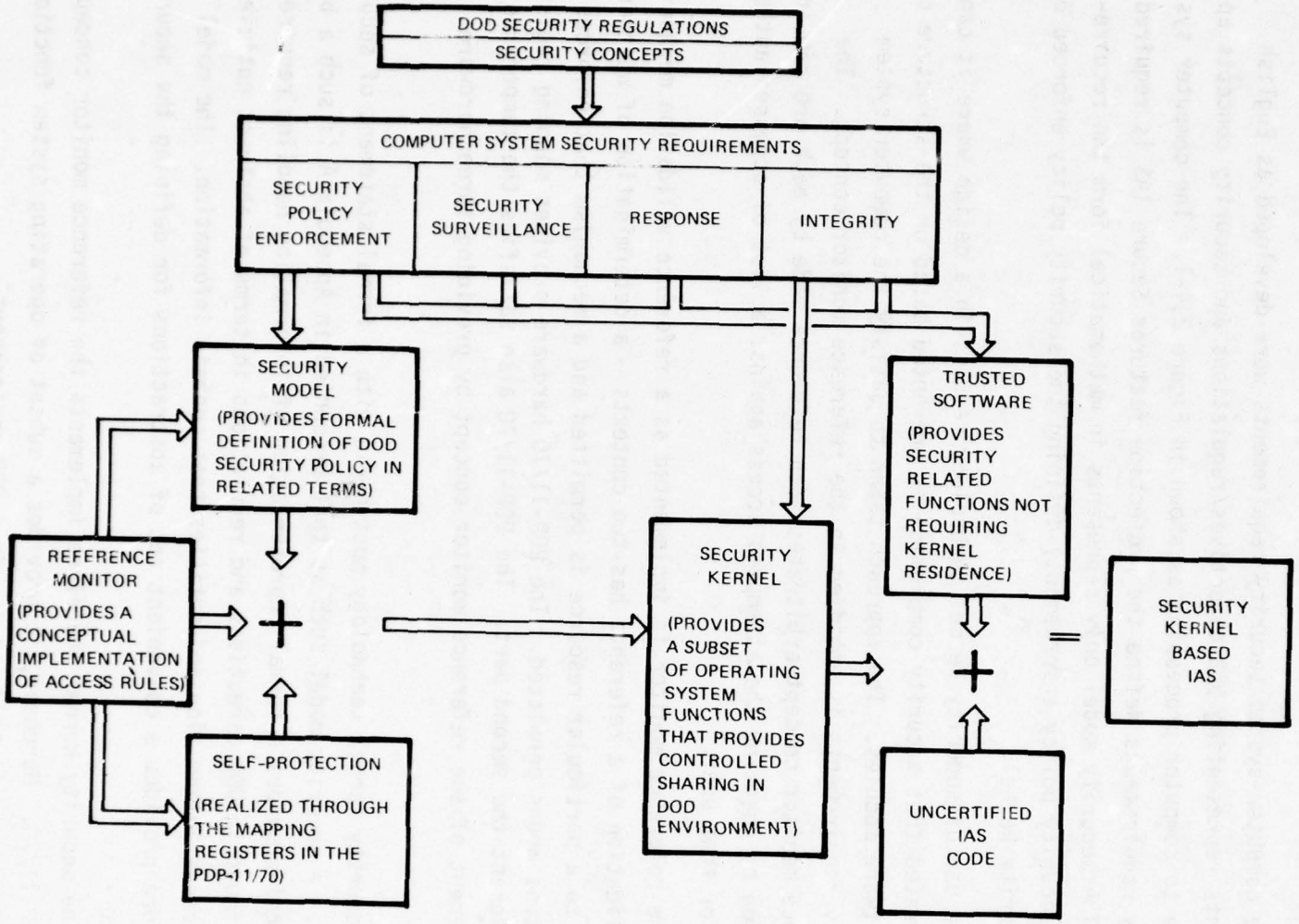



Figure 2.4-1. Computer System Security Requirements Traceability

The remainder of the computer system security requirements (non-kernel security-related software requirements) define functions performed by "trusted software," security sensitive software modules outside the kernel but protected by it and verified to correctly perform the security function.

The final component that comprises the Secure IAS system is the uncertified IAS Emulator code. This code simulates the standard IAS system directives by mapping each of the system directives into security kernel calls. This interface provides a nearly compatible interface with current IAS.

### 2.4.1 Security Policy Enforcement Requirements Traceability

The security policy enforcement requirements were derived mainly from Executive Order 11652 and DOD 5200.1-R. They are presented in summary of those presented in Section 2.3.1, together with references to security policy document paragraphs. These references define the correspondence of specific elements of each requirement statement to specific paragraphs in the security policy documents.

#### 2.4.1.1 Identification Requirements Traceability

ID1 Each computer system user shall be assigned one or more unique identifiers before his usage of the system. The list of assigned user identifiers defines the set of authorized users of the system.

Traceability

The requirement for unique identifiers is implicitly specified in DOD 5200.28: VI-A-1 and DOD 5200.28-M: 4-305f.

ID2 Each time the user uses the system, he shall be required to declare his identity upon initial entry. The system must verify that his identifier is on the list of authorized users before he is allowed any further use of system resources.

Traceability

User identification is specified in DOD 5200.28: VI-A-1 and DOD 5200.28-M: 4-305f.

ID3 Following establishment of a user's identity, that identity shall be authenticated so that the probability of erroneous identification is less than some specified number specified by the Designated Approving Authority for the system.

Traceability

The adequacy of the identification measures is specified in DOD 5200.28-M: 4-305f.

ID4 Each controllable system resource shall be assigned a unique identifier, including all:

- tasks or processes,
- files and memory, and
- I/O devices.

Traceability

Resource identification is only partially traceable to consideration for terminal identification in DOD 5200.28-M: 4-305e and 4-200j.

ID5 The identity of each controllable system resource shall be authenticated at the initiation of system operation and, for those resources which are removable or remote, before access is granted.

Traceability

This requirement is not explicitly traceable to any security policy document and implicit in AFR 300-8.

### 2.4.1.2 Controlled Access Requirements Traceability

CA1 User's access to system resources shall be subject to rules which implement the applicable security policy and which, in combination with authorization data, specify for all user-process-resource combinations:

- whether or not the combination is allowed,
- under what conditions it is allowed, and
- the access privileges allowed.

IS2 Any data that is not to be retained at the completion of a process as controllable system resources shall not be determinable by any other process.

Intent

Recovering residual data must be considered as a potential means of obtaining classified data. Provisions which ensure that classified data or critical elements of the computer system do not remain as accessible residue after a job has been completed are therefore necessary. The specific areas of concern are main storage, direct access storage, tape storage, and video screens. Any device that has a buffer associated with it must have that buffer cleared or overwritten before reuse. Terminal devices which have no provision for suppressing the printing of the password or other sensitive data must also be considered.

IS3 There may be a need for data encryption.

Intent

Data encryption may be required by a number of applications and special devices. Generally, tools for the following should be considered:

- Data being transferred to storage media within the system. The handling of certain sensitive data objects may require this feature.
- Data being transferred outside the system, particularly over communications lines.

### 2.3.2 Security Surveillance Requirements

The security surveillance requirements are described in detail in the following subsections. Each subcategory is given a mnemonic prefix; threat monitoring requirements have been identified by the prefix "TM"; and security audit requirements by the prefix "SA."

2.3.2.1 Threat Monitoring Requirements

TM1 There shall be dynamic monitoring of security related events between the user and computer system as required by the Designated Approving Authority for any misuse, and for real-time notification upon detection of any misuse.

Intent

For dynamic monitoring of information indicative of illegal accesses by system penetrators, it is important that the computer system have a means for immediate notification if the penetrators are to be caught. Information analysis and decision criteria are necessary to provide such notification. Thus, after some number of unsuccessful log-on attempts, the system operator or security officer may be notified by the printing of a message or sounding of an alarm, or both. He can then take appropriate actions to investigate the user, the terminal, and the area where the log-on attempt was conducted. Similar notifications can be provided for illegal user file, program, hardware, and operating system accesses.

TM2 There shall be dynamic monitoring of specific protection features for their correct operation, for detection of any malfunction leading to security compromise, and for real-time notification upon the detection of any malfunction.

Intent

Any hardware and operating system malfunctions in the protection features must be detected to prevent possible security compromise. The malfunctions can arise either by unauthorized modifications to the control features during system maintenance or by accidental malfunctions such as hardware errors. Therefore, the specific protection

mechanisms should be analyzed to identify potential failure modes. The failure modes must either be eliminated, or independent failure detection mechanisms must be incorporated to prevent compromise due to a protection mechanism failure. Further analysis needs to be performed to determine if the detection mechanisms actually detect the identified failures. A failure detection mechanism must assume that a minimum of two independent and simultaneous failures occur before a compromise is possible.

#### 2.3.2.2 Security Audit Requirements

SA1 A computer system security log shall be generated and maintained describing security-related events in sufficient detail to establish individual accountability and damage assessment, as required by the Designated Approving Authority.

Intent

System resource abuse or potential abuse can only be identified from information relating to the access of the resource and the time of access. Thus, a user's first access to a time-sharing system (which may be regarded as access to a terminal hardware component and the operating system) can be recorded at system log-on time. Such information as the terminal used and the time of day can also be recorded. A user's subsequent accesses of hardware resources, key portions of the operating system, and the information managed by the system, can all be recorded to obtain a record of accountability in case of abuse of those resources. Without this record, there would be no way of identifying the source of abuse, such as disclosure, modification, or destruction of system-managed information as well as disruptions of the operating system and degradation of system performance.

The types of component accesses and associated information which can be recorded in a security audit log include:

a. User log-ons, including identifiers used, time of day, and terminal.

b. Hardware resource accesses and usage, including time, type of device, and location.

c. User program and file accesses, including identifiers, classification, storage media identifiers, and time.

d. Security officer accesses, including type of access, entity accessed, and time.

Although they may not be dynamically recorded, operator and maintenance activities involving system accesses should also be recorded in the security audit log. Other important entries include periods of discontinuity such as system startup, shutdown, and recovery.

SA2 The computer system shall provide programs for generating appropriate security-related reports from the system log which can be used to establish individual accountability and damage assessment.

Intent

Critical information recorded in the security audit log can be more easily ascertained by the use of special programs which automatically gather such data and generate appropriate security reports, which are important for the ex post facto analysis of security-related events occurring in the system throughout its operational phase. While the format and content of such reports may be largely installation-dependent, there are fundamental types of information of common interest to most installations. Other criteria for the design of report generation programs and report contents may include

such factors as DOD directives and regulations governing the operation of a particular installation. Any significant security events not recorded on the security audit log or susceptible to automatic report generation should be recorded and reported manually.

### 2.3.3 Response Requirements

The response requirements are described in detail in the following subsections. Each subcategory is given a mnemonic prefix: interception requirements are identified by the mnemonic prefix "IT," and recovery requirements by the prefix "RV."

#### 2.3.3.1 Interception Requirements

IT1 For each hardware or operating system security surveillance mechanism, there shall be one or more compensatory actions.

Intent

For any actual or potential security incident, there should be a corresponding compensatory action or actions to stop the penetration. Otherwise, there would be no deterrent to malicious users who could continually attempt to penetrate the system or deny the services of the system to authorized users.

#### 2.3.3.2 Recovery Requirements

RV1 For appropriate hardware component failures and operating system control component failure, there shall be a timely and operationally useful recovery mechanism.

Intent

If a system is to be secure, it must compensate for incorrect behavior. This will necessitate recovery. Some operating system control components will not have a corresponding recovery routine because of cost considerations. However, even when it is not possible to return quickly to an operational state, recovery mechanisms and routines provide an orderly shutdown of the computer system in addition to resource protection and purging of temporary sensitive data.

### 2.3.4 Integrity Requirements

The integrity requirements are described in detail in the following subsections. Each subcategory is given the same mnemonic prefix "IN."

#### 2.3.4.1 Completeness Requirements

IN1 The controls shall have a defined response (completeness) for every possible attempt to:

- Enter the system
- Request access to system resources
- Exceed, during processing, the boundaries of the resources authorized.

Intent

A computer system's enforcement security controls must be completely defined. They must receive no input for which the response (effect) is not defined in the specification. If there is an input for which the effect is not defined in the specification, the action which takes place when that input is presented to the controls may not conform to the security requirement on the controls. The action caused by such an input could, therefore, be exploited by a penetrator. There must be no way for a user to gain entry to the system without having his identity

established and authenticated. There must be no way a process can request access to resources without having the access controls applied to that request. There must be no way a process can be granted access to specific resources without having isolation controls applied.

Incomplete specification of the security controls of current operating systems is the source of much of their security problems.

IN2 For every possible security event of the type defined for security audit by the Designated Approving Authority, there must be a defined recording procedure.

Intent

A computer system's security audit facilities must be completely specified. There must be no security event for which the recording procedure is not specified in the specification. The term "event" in this context means user initiated events such as log-on, file access, start-up, etc.

IN3 For every possible security activity pattern of the type defined for threat monitoring by the Designated Approving Authority, there must be a response.

Intent

A computer system security response mechanism must be completely specified. There must be no security activity pattern of the type defined for threat monitoring that the corresponding response(s) are not completely specified.

IN4 The security controls (identification, controlled access, isolation, surveillance, and response mechanisms) shall be applied at all times.

Intent

Security must be enforced at all times, although computer systems do not operate continuously. There are periods of discontinuity such as scheduled system maintenance, start-up, shutdown, recovery, and simply idle periods. Idle periods must be handled by physical and administrative security. This leaves basically two cases: periods of continuity and periods of discontinuity.

For periods of continuity, this requirement implies that there is a verifiable method of identifying the source of each request and that the controlled access and isolation controls in combination completely mediate all attempts to access system resources. However, every access does not need explicit validation with the access rules. When a process is executing in some domain, it can exercise the capabilities which belong to that domain and are commonly validated by the isolation features as indicated by requirement IS1. When control passes from one domain to another, the capabilities given to the process will change as indicated by requirement CA2. That is, when a process requests a service of the computer system, the enforcement mechanism of the computer system must be invoked at its entry point. Then the request is either granted or rejected based on the security policy and the authorization data associated with the request.

During periods of discontinuity, which include startup, restart, shutdown, recovery, and maintenance, the computer system must purge sensitive residual data and protect the system resources.

2.3.4.2 Correctness Requirements

IN5 For every request of the security controls (identification, controlled access, isolation, surveillance, and

response mechanisms), the actual response must correspond to that specified in the design specification.

Security requirements are much more stringent than ordinary software verification requirements because:

- Integrity flaws missed in normal verification efforts may provide real security threats. Even though activated only by rarely used inputs, presenting for normal software only a minor operational nuisance, a single flaw, once it has been identified by a penetrator, can be exploited again and again until it is corrected.
- "Hidden" functional capabilities present in actual software in addition to required functional capabilities may be exploitable by penetrators.

Integrity flaws and hidden functional capabilities may be introduced at any level in the development process; hence, the verification at each level must be exceptionally thorough.

IN6 A validated bootstrap loader shall properly initialize the operating system and its security related data bases.

Intent

First, the bootstrap loader must assure that only the certified copy of the operating system is loaded. Secondly, for the operating system to operate correctly it must be properly initialized. This is possible if its initial state is consistent with the initial state prescribed by the system specifications.

### 2.3.4.3 Maintainability Requirements

IN7 The security controls shall be maintained only by procedures, acceptable to the Designated Approving Authority,

which preserve system compliance with the security requirements. Those which are not controlled by the access controls shall not be activated while the system is being used operationally for classified processing.

Intent

Maintenance programs are provided for hardware maintenance personnel and operating system maintenance personnel to keep the system's hardware and software operating properly. There are several concerns here. First, there is the issue that these programs/tools may have excessive privilege to perform their required function. Another issue is the introduction of errors into the protection mechanisms of the system, whether design errors or implementation errors deliberately introduced, such as trapdoors. In particular, provisions are needed for:

1. Constraining maintenance personnel to maintaining specific resources using only specified programs/tools.
2. Determining correct operation of the hardware components.
3. Controlling changes to the software routines which comprise the identification, controlled access, isolation, surveillance, and response mechanisms.

IN8 There shall be appropriate mechanisms and procedures for secure generation, backup and restore of copies of on-line data.

Intent

Secure generation, backup, and restoring of copies of on-line data is essential to security. Some of the more important points to consider follow.

- Generation of backup copies, by its frequency, should attempt to maximize the possibility that the current version of a data object has a backup. A common, and relatively sound, method for doing this is to periodically (where the period may be varied) generate backup copies of all data objects created or modified since the last periodic generation.
- Storage media containing backup copies of data objects should receive special care and handling, since it represents a central collection point for a vast amount of potentially sensitive information.
- It should be possible to easily locate a desired backup copy of a specific data object without compromising the privacy or security of any user. Too often, printed directories of backup storage media are formatted and handled in a way not consistent with good security practice.
- Users should, under special circumstances, be able to avoid the standard procedures for backup copying and retrieval, and to substitute their own. Certain particularly sensitive applications may demand a more complete isolation and tighter control than standard procedures can offer.

IN9 Strict configuration control shall be applied to the computer system security mechanisms.

Intent

Security configuration control is essentially the ability to ensure that all changes to the computer system are accounted for and to verify that these changes do not impair the system security mechanism. Any erroneous change to a security mechanism, whether hardware or software, can have serious

consequences. Thus, management of change to security mechanisms must take into account all versions of the computer system, not just the initial or starter system. Tools and procedures must be provided for maintaining the configuration control of the system; among these are the following:

1. Identification of the security-sensitive modules of the operating system and the security mechanisms in the hardware
2. Identification of all changes that impact security mechanisms
3. Inspection of new releases, such as binary object program comparison software routines
4. Special maintenance procedures for distributing, reporting, and applying changes to security-sensitive modules
5. Verification that the operational system has not changed.

## 2.4 Requirements Traceability

The computer system security requirements were developed as English statements representing DOD directives/regulations and security concepts applicable to computer processing as shown in Figure 2.4-1. The computer system security requirements define the protection features Secure IAS is required to have. The security model only represents in mathematical form the requirements (security policy enforcement) defining the security policy enforced by the security kernel.

The use of security kernel technology results in a design where it can be demonstrated that security compromise is prevented based on the structure of the software modules. The approach taken to satisfy the computer system security requirements is embodied in the reference monitor concept. The reference monitor concept arbitrates each reference made by each program in execution by checking the attempted access against a list of accesses authorized for that user.

The reference monitor is implemented as a reference validation mechanism. The validation of a reference has two contexts — a determination of whether access to a particular resource is permitted and a mechanism that enforces the access modes permitted. The PDP-11/70 hardware provides mapping registers to implement the second part. The PDP-11/70 also satisfies the tamperproof requirement of the reference monitor concept by providing three hardware states.

Security kernel technology must begin with a formal statement of security policy. A security model such as that presented in Appendix A is such a basis. The model provides a formal representation of information handling requirements found in DOD directives and regulations in terms of abstract entities that access information and entities that contain information. The model therefore provides a convenient set of abstractions for defining the security policy.

The security kernel therefore implements the reference monitor concept on the PDP-11/70. Moreover, it provides a subset of operating system functions that provide controlled sharing in a DOD environment.

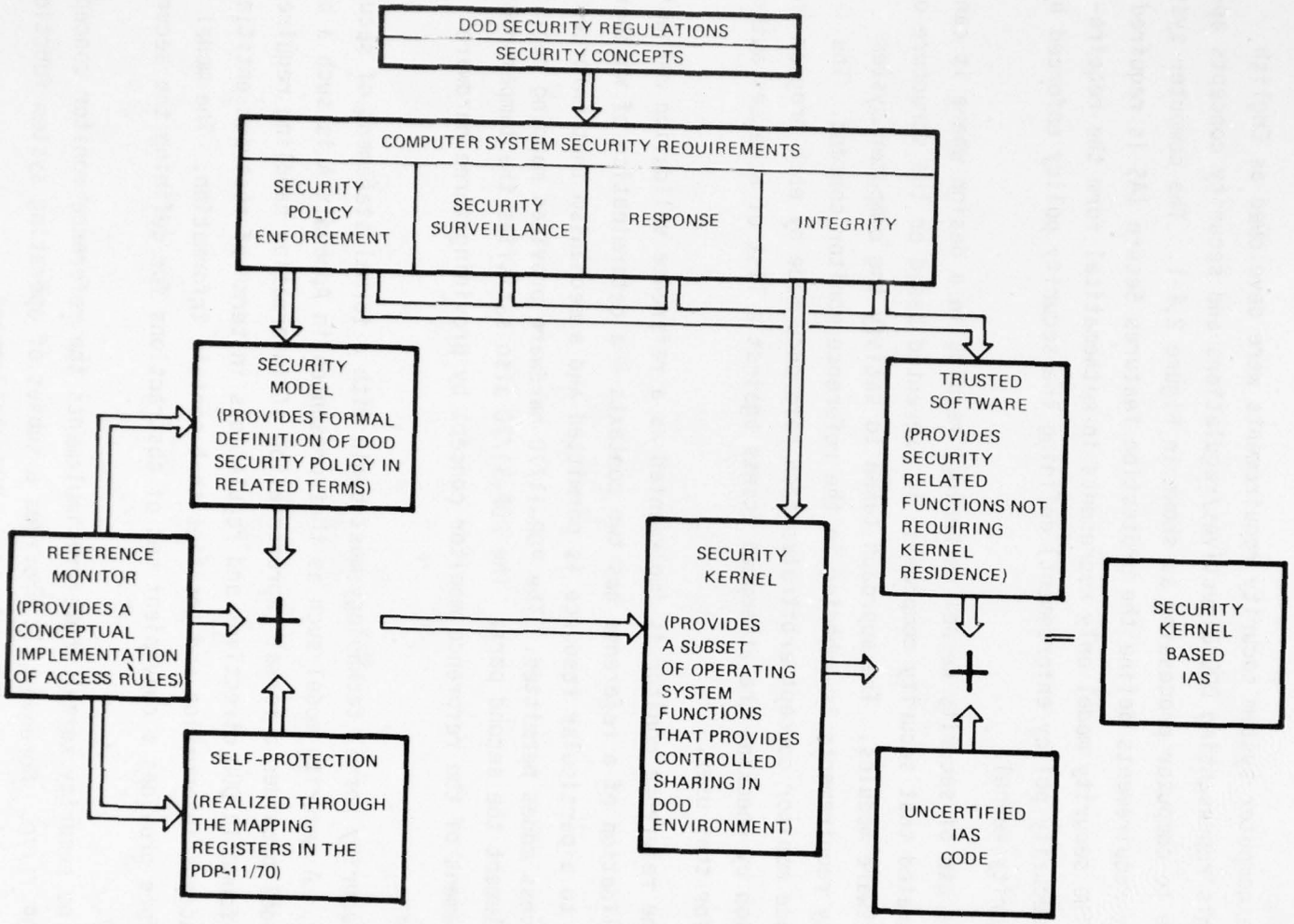



Figure 2.4-1. Computer System Security Requirements Traceability

The remainder of the computer system security requirements (non-kernel security-related software requirements) define functions performed by "trusted software," security sensitive software modules outside the kernel but protected by it and verified to correctly perform the security function.

The final component that comprises the Secure IAS system is the uncertified IAS Emulator code. This code simulates the standard IAS system directives by mapping each of the system directives into security kernel calls. This interface provides a nearly compatible interface with current IAS.

### 2.4.1 Security Policy Enforcement Requirements Traceability

The security policy enforcement requirements were derived mainly from Executive Order 11652 and DOD 5200.1-R. They are presented in summary of those presented in Section 2.3.1, together with references to security policy document paragraphs. These references define the correspondence of specific elements of each requirement statement to specific paragraphs in the security policy documents.

#### 2.4.1.1 Identification Requirements Traceability

ID1 Each computer system user shall be assigned one or more unique identifiers before his usage of the system. The list of assigned user identifiers defines the set of authorized users of the system.

Traceability

The requirement for unique identifiers is implicitly specified in DOD 5200.28: VI-A-1 and DOD 5200.28-M: 4-305f.

ID2 Each time the user uses the system, he shall be required to declare his identity upon initial entry. The system must verify that his identifier is on the list of authorized users before he is allowed any further use of system resources.

Traceability

User identification is specified in DOD 5200.28: VI-A-1 and DOD 5200.28-M: 4-305f.

ID3 Following establishment of a user's identity, that identity shall be authenticated so that the probability of erroneous identification is less than some specified number specified by the Designated Approving Authority for the system.

Traceability

The adequacy of the identification measures is specified in DOD 5200.28-M: 4-305f.

ID4 Each controllable system resource shall be assigned a unique identifier, including all:

- tasks or processes,
- files and memory, and
- I/O devices.

Traceability

Resource identification is only partially traceable to consideration for terminal identification in DOD 5200.28-M: 4-305e and 4-200j.

ID5 The identity of each controllable system resource shall be authenticated at the initiation of system operation and, for those resources which are removable or remote, before access is granted.

Traceability

This requirement is not explicitly traceable to any security policy document and implicit in AFR 300-8.

#### 2.4.1.2 Controlled Access Requirements Traceability

CA1 User's access to system resources shall be subject to rules which implement the applicable security policy and which, in combination with authorization data, specify for all user-process-resource combinations:

- whether or not the combination is allowed,
- under what conditions it is allowed, and
- the access privileges allowed.

Traceability

DOD security policy requires that:

- protected information shall be assigned a security classification category (classified) indicating the degree of protection required and marked with that security classification. EO 11652: 1; DOD 5200.1-R: 1-400, 1-500, 2-309, 4-100, 103, 202.
- access to classified information shall be authorized only if the user requesting the access has:
  - a security clearance denoting eligibility for access to the security classification category. EO 11652: 6(A); DOD 5200.1-R: 7-100, 7-101.
  - a formally determined need-to-know eligibility for access to compartmented information. (Eligibility for access to information assigned to more than one compartment requires user clearance for all of the assigned compartments.) EO 11652: 6(A); DOD 5200.1-R: 12-101.
  - a need-to-know for the particular information to be accessed. EO 11652: 6(A); DOD 5200.1-R: 7-100, 7-103, 12-100.

  The first two access authorization criteria may be called mandatory (nationally determined) security and the third may be called discretionary (locally determined) security.
- information created as a result of access to protected information shall be afforded the degree of protection indicated by the highest security classification of the information accessed, all of the compartments of the accessed information, and a need-to-know for all of the accessed information. DOD 5200.1-R: 2-309, 2-314, 2-315.

The security axioms specified in the security model in Appendix A state the DOD security rules. Seven of the axioms are cross-referenced to specific paragraphs in security policy documents.

Axiom A1. Simple security axiom.

- Security classification categories are defined in EO 11652: 1(a), (b), (c) and in DOD 5200.1-R: 1-500, 501, 502, 503.

- Security clearance (eligibility for access) is defined in EO 11652 6(a) and in DOD 5200.1-R: 7-101.
- Access authorization criteria are defined in EO 11652 6(a) and in DOD 5200.1-R: 7-100, 103, 12-101.

Axiom A2. Security * -property axiom

- The degree of protection required of information created as a result of access to classified information is defined in DOD 5200.1-R: 2-309, 314, 315. The *-property is derived from imposing the required degree of protection on the receiving object and forbidding automatic upgrading with its potential overclassification.
- The *-property relation has the effect of ordering the security classifications and compartment sets of the objects concurrently accessed by the same subject so that:
  - All objects modified but not observed by the concurrent accesses must have security classifications greater than or equal to,and compartment sets supersets of or equal to, the security classifications and compartment sets of all objects both observed and modified.
  - All objects both observed and modified must have the same security classifications and compartment sets.
  - All objects observed but not modified must have security classifications less than or equal to,and compartment sets subsets of or equal to, the security classifications and compartment sets of all objects modified.

Axiom A4. Simple integrity axiom

- Although the subject of information integrity is not specifically discussed in security policy documents, it is implicitly covered by the notions of protection of information in the interest of national security and extending them to cover information modification. Although the security classification assigned to an information object (denoting the degree of protection required against unauthorized observation) is not necessarily the same as the integrity classification assigned to the same object (denoting the degree of protection required against unauthorized modification), the security clearance (denoting the degree of trustworthiness of the user) is the same for observation and modification. Thus the paragraphs referenced for axiom A2 also apply to this relation.

Axiom A5. Integrity *-property axiom

- References are the same as for axiom A2, making an interpretation and extension similar to that for axiom A4.

Axioms A7 and A8. Discretionary observe and modify axioms

- Need-to-know eligibility is defined in EO 11652 6(a) and in DOD 5200.1-R: 7-103.

Axiom A10. Tranquility Axiom

- Partial references are discussed under CA5.

CA2 Any information leakage path shall be identified and its bandwidth determined. An engineering tradeoff study shall weigh the cost of closing it against the exposure by leaving it.

Traceability

This requirement is not explicitly traceable to any security policy in document but is included in need-to-know as discussed in CA1.

CA3 Each computer system user and each controllable system resource shall have sufficient authorization data declared before entry to (or already established in) the computer system so the access rules can be applied before any system resource is used.

Traceability

DOD security policy documents state that:

- authorization data (data used as the basis for authorizing access) for mandatory security -- security classification, and compartments -- shall be assigned to:
  - protected information by the creator of the information in accordance with classification guides and balanced judgment. EO 11652: 2; DOD 5200.1-R: 1-600, 2-100, 2-300, 2-400, 4-207.
  - users by a person specifically authorized to do so (the system security officer). DOD 5200.1-R: 7-100, 7-101- 703.

CA4 Each process shall be associated, when the access rules are applied, with the user who initiated it.

Traceability

This requirement is traceable to DOD 5200.28-M: 4-301d.

CA5 The computer system shall have a method for creating, deleting and changing the authorization data. This method will be subject to the access control rules specified by CA1.

Traceability

DOD security policy documents state that:

- changes to authorization data for mandatory security shall be made only by:
  - the system security officer for user authorization data. EO 11652: 5; DOD 5200.1-R: 1-401, 1-600, 1-602, 2-800, 2-801, 3-100.
  - the owner of the information object for authorization data relating to that object. DOD 5200.1-R: 7-100, 7-101, 103.
- authorization data for discretionary security (need-to-know) shall be determined (and changed) by the person responsible for the information. Generally, need-to-know data for particular information is a list of persons determined to have a need for access to that information in the performance of their duties. DOD 5200.1-R: 7-100.

CA6 Each computer system user shall be notified of the applicable security attributes for all data furnished to the user by the system.

Traceability

DOD security policy states the requirement for security labels. DOD 5200.28-M: 4-305d.

2.4.1.3 Isolation Requirements Traceability

IS1 The computer system shall restrict access of each active process to the domain authorized for the user-process-resource combination.

Traceability

The DOD requirements on hardware features are:

- execution state variables. DOD 5200.28-M: 4-200a.
- separation of user/executive mode. DOD 5200.28-M: 4-200b.
- privileged instructions. DOD 5200.28-M: 4-200c.
- automatic program interrupt. DOD 5200.28-M: 4-200i.
- read, write, and execute access control. DOD 5200.28-M: 4-200k.

The DOD requirements on software features are:

- self-protection of operating system. DOD 5200.28-M: 4-300.
- least privilege. DOD 5200.28-M: 4-301.

IS2 Any data that is not to be retained at the completion of a process as controllable system resources shall not be determinable by any other process.

Traceability

The activity and erasure axioms, A9 and A11, formally represent this requirement for a segment. Erase procedures are specified in DOD 5200.28-M and include:

- normal operations. 7-101.
- magnetic tapes. 7-201.
- rigid magnetic storage media. 7-202.
- internal memory. 7-204, 4-305b.
- magnetic storage media. 7-205.
- clear system procedures. 4-303.

IS3 There shall be provision for data encryption.

Traceability

Whereas data encryption is discussed in both DOD 5200.28 and DOD 5200.28-M, it is not discussed in the context of file encryption.

### 2.4.2 Security Surveillance Requirements Traceability

The security surveillance requirements were derived mainly from DOD Directive 5200.28 and companion manual DOD 5200.28-M. They are present in summary of those in Section 2.3.2, together with references to security policy document paragraphs. These references define the correspondence of specific elements of each requirement statement to specific paragraphs in the security policy documents.

#### 2.4.2.1 Threat Monitoring Requirements Traceability

TM1 There shall be dynamic monitoring of security related events between the user and computer system as required by the

Designated Approving Authority for any misuse and for real-time notification upon detection of any misuse.

Traceability

This requirement is traceable to DOD 5200.28-M: 5-100, which describes limited events to be monitored, and to DOD 5200.28-M: 4-305a.

TM2 There shall be dynamic monitoring of specific protection features for their correct operation, for detection of any malfunction, and for real-time notification upon the detection of any malfunction.

Traceability

The DOD requirements on hardware features are:

- register protection. 4-200e.
- loadable/storable registers. 4-200f.
- fetch cycle error detection. 4-200g.
- parity checks and memory bounds checks. 4-200h.
- two or more independent controls. 4-100.

2.4.2.2 Security Audit Requirements Traceability

SA1 A computer system security log shall be generated and maintained describing security-related events in sufficient detail to establish individual accountability and damage assessment, as required by the Designated Approving Authority.

Traceability

Requirements for recording of information to support security audit are found in DOD 5200.28-M: 5-100.

SA1 The computer system shall provide programs for generating appropriate security-related reports from the system log which can be used to establish individual accountability and damage assessment.

Traceability

No explicit requirements are stated in DOD directives; however, DOD 5200.28-M: 5-100 implies such programs in order "to permit a regular review of system activity."

### 2.4.3 Response Requirements Traceability

The format of this section follows that of the previous section.

#### 2.4.3.1 Interception Requirement Traceability

IT1 For each hardware or operating system security surveillance mechanism, there shall be one or more compensatory actions.

Traceability

This requirement is traceable to other fundamental features from DOD 5200.28-M: 4-305.

#### 2.4.3.2 Recovery Requirement Traceability

RV1 For appropriate hardware component failures and operating system control component failures, there shall be a timely and operationally useful recovery mechanism.

Traceability

This requirement is traceable to Shutdown and Restart from DOD 5200.28-M: 4-304.

### 2.4.4 Integrity Requirements Traceability

This section parallels the previous two in format.

#### 2.4.4.1 Completeness Requirements Traceability

IN1 The controls shall have a defined response (completeness) for every possible attempt to:

- Enter the system
- Request access to system resources
- Exceed, during the processing, the boundaries of the resources authorized.

Traceability

This requirement is traceable to DOD 5200.28: VI-A-3 on system stability and DOD 5200.28-M: 4-200d on hardware features.

IN2 For every possible security event of the type defined for security audit by the Designated Approving Authority, there must be a defined recording procedure.

Traceability

This requirement is traceable to DOD 5200.28: VI-A-3 on system stability.

IN3 For every possible security activity pattern of the type defined for threat monitoring by the Designated Approving Authority, there must be a response.

Traceability

This requirement is traceable to DOD 5200.28: VI-3-A on system stability.

IN4 The security controls (identification, controlled access, isolation, surveillance, and response mechanisms) shall be applied at all times.

Traceability

This requirement is partially traceable to DOD 5200.28: VI-A-1 on individual accountability.

#### 2.4.4.2 Correctness Requirements Traceability

IN5 For every request of the security controls (identification, controlled access, isolation, surveillance, and response mechanisms), the actual response must correspond to that specified in the design specification.

Traceability

This requirement is at most implied in DOD 5200.28: VI-A-3 on system stability.

IN6 A validated bootstrap loader shall properly initialize the operating system and its related data bases.

Traceability

This requirement is only partially traceable to DOD 5200.28-M: 4-304 on shutdown and restart.

2.4.4.3 Maintainability Requirements Traceability

IN7 The security controls shall be maintained only by procedures, to the Designated Approving Authority, which preserve system compliance with the security requirements. They shall not be activated while the system is being used operationally for classified processing.

Traceability

DOD 5200.28-M: 2-102 and 2-103 specify controls over system, operation, maintenance personnel and DOD 5200.28-M: 4-302 specifies controls on test and debugging programs.

IN8 There shall be appropriate mechanisms and procedures for secure generation, backup and restore of copies of on-line data.

Traceability

This requirement is traceable to DOD 5200.28: VI-A-4 on data integrity.

IN9 Strict configuration control shall be applied to the computer system security mechanisms.

Traceability

Software Configuration Management Policies and Requirements are described in applicable portions of MIL-STD-483, MIL-STD-490, MIL-STD-480, MIL-STD-1521A, and AFR 300-8.

# 3. CURRENT IAS ARCHITECTURE

## 3.1 Overview

IAS is a multi-function operating system designed to handle real-time, interactive, and batch processing simultaneously. It is the top end of the DEC PDP-11 operating systems in terms of functionality, language support, and protection.

IAS consists of four components as shown in Figure 3.1-1. The Kernel Executive and the Time-sharing Executive are the two major ones. The Kernel Executive and its interface run in kernel mode of the PDP-11/70 and provide facilities for executing real-time tasks. IAS execution of real-time tasks is fully compatible with that of the DEC RSX-11D real-time operating system.

The Time-sharing Executive and its interface run in user mode in the environment provided by the Kernel Executive. They control the execution of interactive and batch mode programs.

The third component of IAS, the Files-11 file system, provides system services which enable a user to construct, access, and modify a protected collection of records known as a file.

The fourth component, the system tasks, is a conglomerate of various system maintenance and utility tasks.

Each of the above mentioned components and their interaction is further described in the following sections.

## 3.2 Kernel Executive

The Kernel Executive and its data base (SCOM) make up the core resident portion of the IAS system. The Kernel Executive is in fact DEC's RSX-11D Operating System, with modifications to interface with the Time-sharing Executive. Characterized as a disk-based, event-driven, real-time multi-programming system, the Kernel is the only task that operates in kernel mode under IAS. It is the foundation on which the IAS Time-sharing Executive is

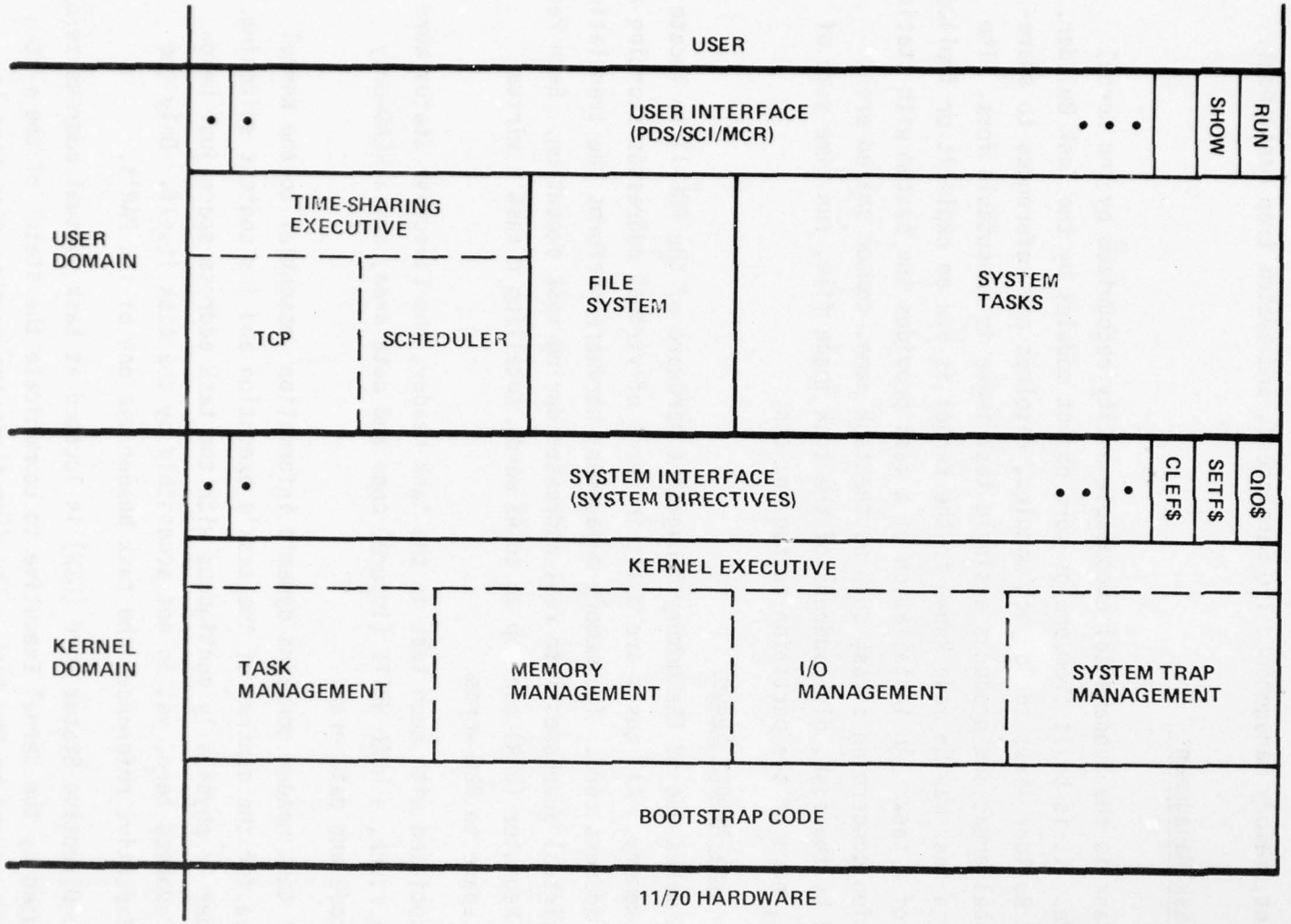



Figure 3.1-1. Current IAS Architecture

built. The major functions performed by the Kernel Executive are: task management, memory management, I/O management, and system trap management.

### 3.2.1 Task Management

A task is the fundamental executable entity recognized by the Kernel Executive. It is built from one or more object modules by the Task Builder. The Task Builder links the object modules, resolves any references to shareable global areas and produces a single task image in executable form. The only way a task may be made known to the Kernel is via an explicit or implicit INSTALL of a task. An installation of a task provides the system with static information concerning a task such as the task name, common shared areas accessed by the task, disk address of the task image file, run time size of the task, name of the partition to run in, etc.

#### 3.2.1.1 Task Memory Space

IAS makes use of the memory management hardware of the PDP-11 to locate a task in memory. All tasks are built in terms of virtual addresses starting at virtual address zero. The memory management hardware performs the translation of the virtual addresses into real addresses during task execution. Each Page Address Register (PAR) can map up to 4K words, totalling a task's virtual address space to 32K words.

Associated with each task is the task header, the Directive Status Word (DSW), a stack, a READ/WRITE (impure) code and data area, and a READ-only (pure) code and data area.

The task header contains dynamic information essential to the Kernel Executive for the control of the task's execution and its context switching. The header is physically contiguous with the task address space just below virtual address zero, yet is not accessible by the task itself. Only the Kernel Executive references the task header via one of its PAR's.

The Directive Status Word (DSW) is located at task virtual address zero and is used by the Kernel Executive to communicate the status of the system directives issued by the task. The directives are further discussed in Section 3.2.5.

### 3.2.1.2 Task Privileges

A task may execute under the Kernel Executive as a non-privileged or privileged task. Non-privileged tasks execute in user mode and are suitable for most non-real-time applications. Their means of communication with the Kernel Executive is limited to system directives.

Privileged tasks, on the other hand, have enormous control over the system. Although they execute in user mode, they have direct access to most of the Kernel Executive routines and its data base plus the I/O page. For each privileged task, three of its PARs map to SCOM and one maps to the I/O page. The address space for the task's own code, stack, and data is, therefore, limited by the PARS which are available for its own use. Device drivers under IAS, for example, are implemented as privileged tasks.

### 3.2.1.3 Task Scheduling

The Kernel Executive is responsible for real-time task scheduling. The scheduling of non-real-time tasks (i.e., Batch or Interactive) is governed by the Time-sharing Scheduler (see Section 3.3.2).

The scheduling of real-time tasks is determined by task priority (Ø through 25Ø; where 25Ø is the highest) and memory availability. All active tasks in the system are accounted for and maintained by the Kernel Executive in a priority-ordered Active Task List. The Active Task List is scanned from the top at each occurrence of a significant event, or is partially scanned whenever a task suspends its own execution. Significant events are usually caused by I/O initiation/completion, task termination, request for task execution, mark time expiration, and certain executive directives.

When a real-time task of a higher priority is competing for memory with the current lower priority active tasks, the Kernel Executive will attempt to checkpoint those lower priority tasks. Checkpointing occurs only for real-time tasks. One or more tasks may be rolled out to disk in order to free memory for a higher priority task, provided those lower priority tasks are built as checkpointable.

#### 3.2.1.4 Inter-task Communication

There are basically four techniques for two or more tasks to communicate information to one another:

- Global event flags
- Common areas
- SEND and RECEIVE data blocks
- Shared access to data files

Each task has access to 64 event flags. Event flags 1-32 are local and 33-64 are global. The last eight of both sets are reserved for executives and system task use. The remaining 24 global flags may be used for inter-task communication. Control of these global event flags is totally subject to the user's discretion and great care is required when they are used as a mechanism to synchronize inter-task communication.

Common areas are read/write shareable global areas that are created and installed in similar ways as tasks. Tasks may read from and/or written into these common areas to communicate.

Tasks may also utilize several SEND/RECEIVE system directives to communicate by transferring blocks of data. The data blocks are queued for the receiving task and are processed whenever the receiving task becomes activated.

Shared-access to data files functions in the same way as common areas. However, shared data files reside on a Files-11 volume while common areas are memory resident. Access to these files is governed by the Files-11 protection mechanism.

### 3.2.2 Memory Management

IAS utilizes only the kernel and user instruction space of the PDP-11 memory management hardware. The Kernel Executive is the only task that executes in kernel domain. Each task running in user domain has its own virtual memory which may be relocated, at execution time, throughout real memory.

Memory usage may be divided into two areas. The first area is used by the Kernel Executive and its supporting software such as the Bootstrap code,

the Kernel Executive itself, and the System Communication Area (SCOM). The second area is divided into partitions which are used to accommodate user or system tasks.

Partitions are physical areas of contiguous memory created at system generation time. They are static and are identified by their name, size, starting address, and type. There are three types of partitions supported by IAS which dictate to the Kernel Executive the memory allocation policies for task execution: user controlled, system controlled, and time-sharing partitions.

User controlled partitions can accommodate only real-time tasks and only one at a time. However, more than one task may be installed to run in the same user controlled partition. The user has complete control over activity in the partition. A higher priority task may checkpoint the current task provided the current task is checkpointable. A user controlled partition is well suited for permanently resident event driven tasks that require a critical response time.

System controlled partitions can accommodate only real-time tasks, but more than one at a time. The Kernel Executive determines the placement of tasks in memory and is responsible for task loading. Multiple tasks may be checkpointed to provide sufficient contiguous space for a higher priority incoming task.

Time-sharing partitions are similar to system controlled partitions except tasks may be shuffled to the bottom of the partition in order to provide sufficient contiguous space and to reduce memory fragmentation. However, checkpointing and swapping always precede shuffling. A time-sharing partition can accommodate both real-time and time-sharing tasks. Time-sharing tasks can run only in a time-sharing partition and are swapped (checkpointing is its real-time counterpart) to create space. Swapping is a function under the control of the Time-sharing Executive. The Time-sharing Executive interacts with the Kernel Executive to effect swapping, shuffling, and time-sharing task loads. In fact, the Time-sharing Executive is a good example of a real-time task executing in a Time-sharing partition. (See Section 3.3 for

detailed discussion on the Time-sharing Executive and time-sharing job scheduling.)

### 3.2.3 I/O Management

The Kernel Executive maintains device independence for its I/O operations. This is accomplished by associating each physical I/O device with a logical unit number (LUN). These logical units may be assigned or reassigned dynamically via a system directive or one of the terminal interfaces. These logical unit numbers have no connection with a physical device unless an assignment has been made for a particular task.

Each task maintains its own set of LUN's in its task header called the Logical Unit Table (LUT). Each entry in the table contains a pointer to the physical device that was last assigned to that LUN and a pointer to information concerning an opened file on that LUN.

I/O requests are intercepted by the Kernel Executive which dispatches each of the requests to the responsible I/O device handler to service. The IAS device handlers run as tasks under the Kernel Executive but not as a part of the Executive. They are privileged tasks executing in user mode having access to SCOM and the I/O page.

There are five basic components to a typical device handler task:

1) A communication area.
2) Device initialization code to execute upon loading.
3) Code to dequeue I/O requests.
4) An interrupt service routine.
5) A power fail recovery section.

When an I/O request is made via the QIO system directive, the Kernel Executive queues the request to the device handler task for the physical device associated with the LUN. Each I/O handler keeps its own I/O Request Queue (IRQ) and services requests on a priority basis. The Kernel Executive does not perform any interpretation or validation of the requests but merely

verifies that the LUN is assigned to a physical unit and that the handler task is in memory and loaded or running, before queueing the request.

Each I/O request contains an I/O function code that indicates to the handler which operation to perform. The device handler is responsible for all device dependent operations. A standard set of error conditions is returned in what is called an I/O Status Block whenever I/O is unsuccessful.

### 3.2.4 System Trap Management

The system trap management function of IAS provides the means for a user to request that his task be informed of certain events occurring in the system. Upon occurrence of such an event, the system transfers control to a user specified routine within the task. Such a transfer of control is known as a trap or interrupt.

There are two types of traps or interrupts; the Synchronous System Trap (SST) and the Asynchronous System Trap (AST). An SST notifies the task of events directly associated with the execution of instructions. An example of an SST would be an odd addressing error. If no routine is supplied to service the SST, the system aborts the task.

An AST notifies the task of events which occur asynchronously to the task execution; that is, the trap occurs independently of the execution of the task to receive the trap. An example of an AST is the completion of an I/O operation. If no routine is supplied to service the AST, the task continues normal execution.

### 3.2.5 System Interface

The user at his terminal may interactively communicate with the Kernel Executive via the Monitor Console Routine (MCR). The MCR is a terminal interface between the user terminal and the Kernel Executive. The interface is implemented by means of non-privileged and (UIC) privileged MCR system commands which execute in user domain. It is organized internally as a re-entrant privileged dispatcher task and a set of independent system command tasks. The

individual tasks are called by the dispatcher as needed and are released automatically after execution. The MCR commands allow the user to gain access to the system, initiate and terminate execution of user/system tasks, and adjust, modify, and control the system environment.

The Time-sharing Executive communicates with the Kernel Executive via the system directives or directly through executive privileged tasks that are mapped into the Kernel Executive space.

System directives are privileged and non-privileged. They enable a task to instruct the Kernel Executive to perform operations for that task. All real-time tasks have the right to use privileged directives. The directives are grouped under the following categories:

1) Task execution and control
2) Informational
3) Event Associated
4) Trap Associated
5) I/O Related

## 3.3 Time-Sharing Executive

The Time-sharing Executive (also known as the Time-sharing Scheduler) executes in user mode as a privileged real-time task under the control of the Kernel Executive. It provides an environment in which time-sharing tasks, both interactive and batch, may be run.

The Time-sharing Executive maintains its own data base (IASCOM) and interfaces directly with the Kernel Executive to control memory allocation, swapping, task termination, task suspension, and time-slice expiration.

Effectively, the Time-sharing Executive is the lowest priority real-time task. All true real-time tasks have a higher priority than the Time-sharing Executive while all time-sharing tasks are scheduled at a lower priority than the Time-sharing Executive. There are two major components in the Time-sharing Executive: the Time-sharing Control Primitive (TCP) and the Scheduler.

### 3.3.1 Time-sharing Control Primitive

The Time-sharing Control Primitive (TCP) is a privileged task which provides a software interface to the time-sharing facilities of the IAS system and is responsible for maintaining integrity between all time-sharing tasks. It transforms all requests initiated from time-sharing terminal interfaces, generally referred to as Command Language-Interpreters (CLI's), into Kernel Executive requests. The Time-sharing Executive acknowledges all time-sharing terminal requests as "jobs" rather than "tasks" and dispatches these jobs based on a different scheduling policy. This concept of a job is transparent to the Kernel Executive and is maintained strictly by the Time-sharing Executive. However, each job is eventually translated into a Kernel recognizable task and the Kernel is informed of its time-sharing origin.

TCP is implemented as a handler task that receives QIO requests from CLI's. Some of the functions serviced by TCP are:

1) Special functions
2) Task (recognizable to the Kernel) initiation and control
3) CLI control
4) Job node control
5) Device management
6) Terminal event control
7) CLI service
8) System management

A user program interface to TCP is provided under Time-sharing Control Services (TCS) for user support to write CLI tasks. TCS consists of two parts: a set of macros and a set of subroutines. The macros generate data blocks and calls to TCS subroutines. TCS provides to users a subset of TCP capabilities.

### 3.3.2 Scheduler

The Time-sharing Scheduler consists of the three main tasks TSS1, TSS2, and TSSNUL. The first of these, TSS1, appears on the Kernel Executive Active Task List (ATL) at priority 100. TSS1 is privileged and provides the interface to IASCØM and the kernel data base (SCØM). It schedules the user tasks

to run at a priority one lower than itself but one higher than TSS2. Thus TSS1 and TSS2 "sandwich" the user task in the kernel ATL.

TSS1 waits on four events. They are:

1) Time Slice Event - set in order to run a time-sharing task or to drop to some lower priority task in the system.

2) Disc Driver Event - set when a request by TSS1 to load or unload a task is complete.

3) Event flag set by TSS2.

4) Event flag set when the time-sharing session is complete.

The scheduling algorithm is based on TSS1's attempts to reduce as far as possible the average response time to all user demands. The Scheduler differentiates between various levels of user task importance and urgency of service (highly interactive task vs. CPU bound task) and establishes a fair order of service. At each of the levels, a round robin queue is maintained. The Scheduler always tries to schedule a task at the highest level. Tasks at the same level are treated equally in the round robin queue.

Included in the algorithm are provisions for lowering a task's level as it becomes more CPU bound and raising the task's level in order to prevent a task from never being executed.

Swapping takes place when the Scheduler selects a task to execute which is not in core. The Scheduler selects all tasks that need to be swapped out in order to load the selected task in one attempt. At the same time, the Scheduler will attempt to leave task(s) in memory so that a task may be executing asynchronously while the swapping operation is taking place. After a task has been swapped into memory, it will not be swapped out again until it has received processor time.

Batch tasks are normally run at the lowest scheduling level. They are normally marked as not swappable and not to be promoted in the scheduling level.

The task TSS2 is always activated by TSS1 when TSS1 gives up control to a time-sharing task. If there is no task to execute then TSS1 will not activate TSS2. TSS2 only executes if the time-sharing task scheduled by TSS1

issues some type of wait before its time slice becomes due. In that case, TSS2 will run and set an event flag that is recognized by TSS1. The setting of this event flag will cause a significant event which will in turn cause the ATL to be scanned. Finally, TSS1 will be executed.

TSSNUL is always runnable. It provides a point on the ATL below which no task will be dispatched by the Kernel Executive.

### 3.3.3 User Interface

A user at his terminal communicates with the Time-sharing Executive through the use of a Command Language Interpreter (CLI). A CLI is a time-sharing task that recognizes input from a terminal to which it has been assigned. The ability to recognize input from a terminal is the only fundamental property that distinguishes a CLI from another time-sharing task.

There are two DEC supplied CLI's: the System Control Interface (SCI) and the Program Development System (PDS). The SCI provides facilities for the run time control of the IAS System. It provides for operational control of the system's activities which depend on the number and types of users within the system. PDS provides a generalized interface for program development and execution which runs interactively for terminal users and in batch mode for batch jobs. There are two privileges associated with PDS. The PDS command privilege allows certain PDS commands or sets of commands to be executed. The PDS task execution privilege allows the user to initiate tasks which are executive privileged and/or tasks which perform privileged directives.

DEC also provides a facility with which the IAS users may write their own CLI's; the Time-sharing Control Services (TCS). A user need not use TCS to communicate with TCP. TCP, being a pseudo-handler task, will attempt to service any QIO system directive. TCS is recommended as a means to standardize all CLI interfaces to TCP.

## 3.4 File System

The IAS file control system is a collection of system services that permits the user to treat input/output operations (I/O) as transactions between a program and a named, protected collection of records known as a file. A file may be a permanent structure residing on a volume of magnetic media or a transient collection of records being input or output to a non-file structured device (e.g., card reader, line printer).

The standard IAS file system is called Files-11. This file system supports file structures on disk, DECtape and magnetic tape volumes. All I/O to such volumes is normally done via the Files-11 system. All volumes can also be mounted as foreign volumes meaning that the volume is not to be accessed via the standard file system.

All Files-11 disk and DECtape files have the same basic structure and the same protection information. Input and output to Files-11 files is accomplished by use of the File Control Services (FCS) macro calls.

### 3.4.1 File Structure

Each Files-11 disk volume contains two kinds of directory files:

1) Master File Directory (MFD)

2) User File Directory (UFD)

The MFD is an index of all UFDs stored on the volume. The MFD is automatically generated when the volume is initialized by Files-11. Once an MFD is established on a volume by the System Manager, the System Manager or user can establish UFD's on that volume.

The data structure for each Files-11 file consists of a file header and a data area. The file header information is contained in the index file for the particular volume. The file header information includes:

1) File Owner (User Identification Code)

2) File Name

3) File Type (e.g., BAS identifies a BASIC source file)

4) Version Number Field

5) File Protection Field

6) Data Pointer Field(s)

The file data area is pointed to by the data pointer field in the file header. The user data in the data area may be structured in any manner.

### 3.4.2 File Protection

Files-11 disks and DECpack volumes have both a protection mask for access to the volume and individual protection masks for each file within the volume. Default file protection is specified by the System Manager at volume initialization time and given by default to all files created on the volume.

IAS defines four modes of access: read, write, extend, and delete. The protection mask designates the type of access each ownership category is permitted. The file ownership categories are described in Table 3.2.4-1.

When a task attempts to access a volume or file, the file system checks the user task access in the following order:

1) Volume Access

2) User File Directory (UFD) Access

3) File Access

### 3.4.3 File Control

The IAS File Control Services (FCS) enable the user to perform record-oriented and block-oriented I/O operations and to perform additional functions (e.g., open, close, wait, delete) required for file control. To invoke FCS functions, the user issues macro calls to specify the desired file-control operations.

The macros enable the user to read and write files on file-structured devices and to process files in terms of logical, rather than physical, records. Data can be written to a file in a manner that enables the data to be retrieved at will. Data can be retrieved without the user having to know

the exact format in which it was written to the file.

TABLE 3.2.4-1

File Ownership Categories

| OWNERSHIP | DESCRIPTION |
|---|---|
| SYSTEM | All task that run under a system user identification code (UIC). |
| OWNER | All tasks that run under the same UIC as the owner of the file or volume. |
| GROUP | All tasks that run under a UIC that has the same group number as the UIC of the owner of the file or volume. |
| WORLD | Any task that is not included in one of the three above categories. |

## 3.5 System Tasks

The system tasks are a collection of installed tasks that provide system facilities. These facilities include system startup commands, utility programs, system recovery and file preservation procedures, device, volume, and library managers, and user authorization and privilege functions.

### 3.5.1 System Startup Commands

The system startup commands are used to override parameters specified at system generation time and to specify the swapping files. There are seven commands which are used to modify the batch and time-sharing processing parameters. These are:

1) BATCH - Specify batch parameters

2) DEVICE - Specify devices used by time-sharing users

3) PARTITION - Specify priorities and partitions for time-sharing tasks

4) QUANTUM - Specify scheduling parameters for time-sharing session

5) SWAP - Specify swap parameters

6) TERMINAL - Specify terminals for time-sharing sessions

7) START - Make available the time-sharing system

8) ZAP - Provides a facility for examining and modifying task image files and data files.

### 3.5.2 Utility Programs

The utility programs permit task building, task debugging, and file operations such as copying, editing, transferring, verifying, compressing, dumping, modifying, and creating libraries. Each of the utility programs is briefly described below:

1) Peripheral Interchange program (PIP). PIP copies files from one device to another.

2) Editor (EDI). EDI is used to enter source programs into the system and to modify them as needed.

3) File Transfer Program (FLX). FLX performs file format conversions

4) Verification (VFY). VFY checks the consistency and accuracy of system files on a Files-11 device.

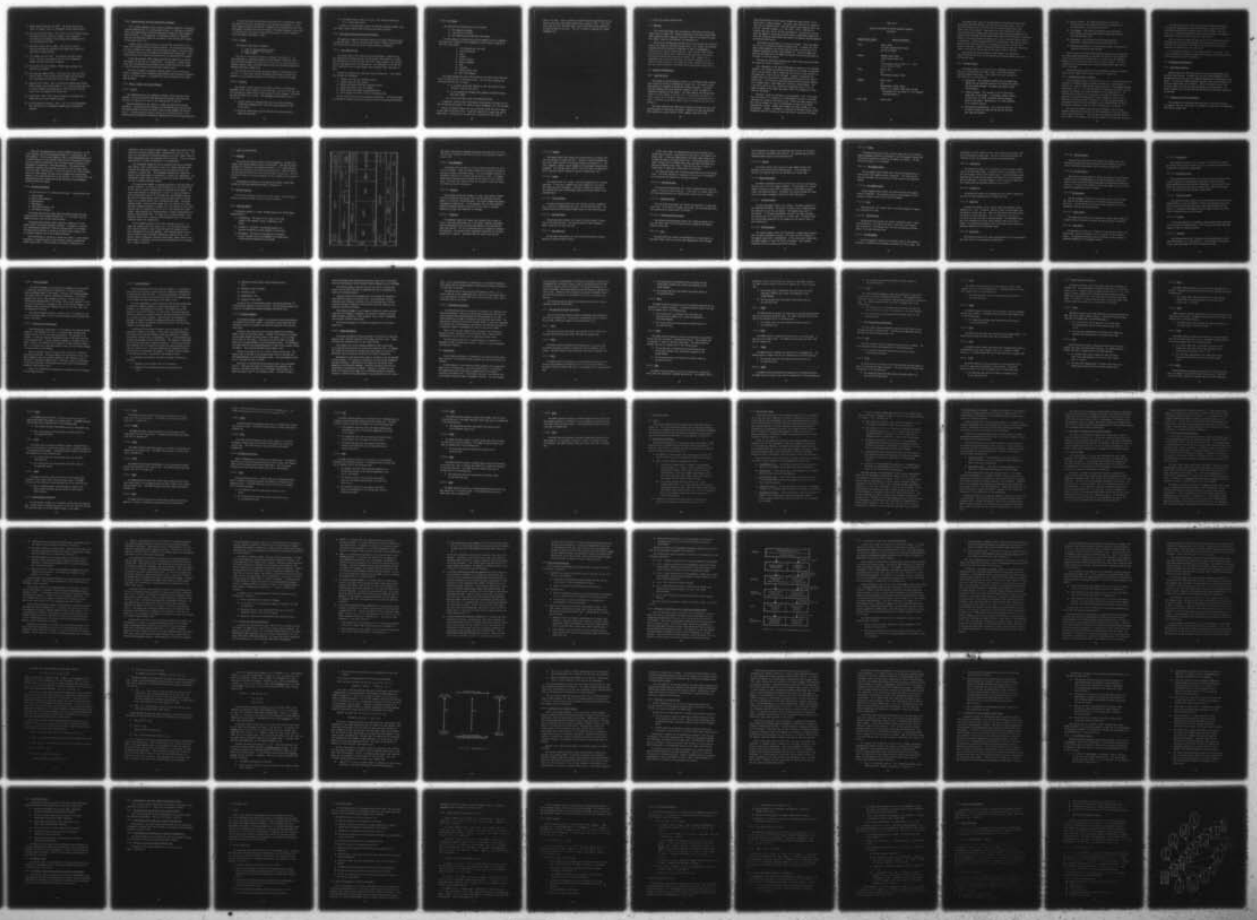
5) Source Language Input Program (SLP). SLP is a batch-oriented editing program that is used to create and maintain source language files on disk.

6) Dump utility (DMP). DMP enables the user to obtain a printout of files.

7) Librarian (LBR). LBR provides the user with the capability to create and maintain disk resident libraries of object and source modules.

AD-A070 141 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
MULTILEVEL SECURITY FOR INTELLIGENCE DATA PROCESSING SYSTEMS.(U)
APR 79 J COTRELL, C SHU, G SHORT F30602-77-C-0119
UNCLASSIFIED RADC-TR-79-88 NL

2 OF 3
AD
A070141



8) Object Module Patch Utility (PAT). PAT allows the user to patch, or update, code in a relocatable binary object module.

9) ZAP Utility Program (ZAP). ZAP allows a user to directly examine and modify files on a Files-11 volume. Task images and data files may be patched in an interactive environment without reassembling the files.

10) Bad Block Locator Utility (BAD). BAD verifies Files-11 disks and DECtapes by determining the number and location of bad blocks on those devices.

11) File Compare Utility (CMP). CMP compares two ASCII source files. The files are compared line by line to determine whether parallel records are identical.

12) Preservation Utility (PRESRV). PRESRV copies volumes from one device to another.

13) Disk Save and Compress (DSC). DSC copies the files on Files-11 disks to tape or disk for backup and storage, and reallocates and consolidates the disk area used for data storage.

14) Squeeze Utility (SQZ). SQZ is a source text compression program whose purpose is to conserve memory. It is used to reduce the size of source programs by eliminating all trailing blanks and tabs, blank lines, and comments from source text.

15) Task Builder (TKB). TKB is used to produce an executable task image from one or more compiler outputs.

16) On-line Debugging Technique (ODT). ODT is a run-time debugger that aids the user in debugging assembled and linked object programs.

### 3.5.3 System Recovery and File Preservation Procedures

IAS initiates automatic error recovery procedures whenever it encounters a device hardware error or a system power failure. In the case of a device error, the system retries the failed operation a number of times depending upon the device. If the operation is completed successfully, operation contirues uninterrupted though the occurrence of the error may be recorded in the system error file.

A power failure recovery routine is provided for occurrences of the power level dropping below the minimum required. The system users and operator will be unaware of a power failure providing the power is restored rapidly, otherwise the system will appear to be effectively switched off. Total system failures which cannot be recovered from are reported to the operator.

A core dump analyzer (CDA) program provides the capability to analyze the software at the time of a system crash. In addition, a single system error log is maintained on which memory error and device errors are logged by both the Kernel Executive and the device handler tasks.

Files may be copied to secondary storage volumes in one of two ways. Either a "snapshot" of the complete volume contents can be written to magnetic tape or its file contents, or selected files may be written onto any other back up medium.

### 3.5.4 Device, Volume, and Library Managers

#### 3.5.4.1 Devices

IAS supported devices are managed by resident, device specific device handlers. The device handler tasks perform the functions that enable the physical I/O operations to occur on the supported PDP-11 devices.

Devices configured in the system as a whole are specified at system generation. Devices which are available to time-sharing users are specified at system startup time using the DEVICE system startup command (see section 3.5.1). Time-sharing devices may be used by all time-sharing users in the system and may be shared or single-user, depending on the device characteristics.

In order that users do not have to wait for a serial processing, single-user device to be free, such devices can be declared as spoolable. In this case, output directed to the device is automatically redirected to disk files and requests to output the files are queued. The queued requests are serviced in order and the files output to the specified device as soon as possible after the request for output is made.

### 3.5.4.2 Volumes

IAS supports three types of volumes:

1. Files-11 disk and DECtape volumes
2. Files-11 magnetic tape volumes
3. Other volumes

The Files-11 file control system is discussed in Section 3.4. All file processing is controlled through Ancillary Control Processors (ACPs). There are three standard ACPs which are used for Files-11 disk and DECtape volumes, magnetic tapes, and installations requiring significant amounts of DECtape processing, respectively.

The user may also gain access to other volumes in formats other than Files-11. In this case, the user designates the volume as foreign and performs his own accessing functions not included in the IAS file system.

### 3.5.4.3 Libraries

IAS supports a simple means of providing a library of installation dependent system library tasks for use by the PDS or SCI user. Any task can be installed as a system library task by means of an SCI command. This makes the task available to all users. The advantages associated with such a library are:

1) A local library of installed tasks may be provided, allowing installation dependent facilities to be made available to PDS or SCI users.

2) Since the system tasks are pre-installed, they can be quickly loaded by the system.

3) The system library tasks can be run in the system as multi-user (i.e., single copy) tasks.

In addition to system library tasks, IAS provides shareable global areas (for code or data), object model libraries, and macro libraries.

### 3.5.5 User Authorization and Privilege Functions

IAS supports a number of different types of privileges that may be given to users and tasks. This section describes these privileges along with the user authorization mechanism and the system accounting procedures.

#### 3.5.5.1 User Authorization

IAS system usage in PDS interactive and batch modes is controlled by user authorization information held in the system's User Profile File (UPF). The information in the UPF includes each user's name, password, UIC, default devices, and default user file directories. The information in the UPF may only be modified by the System Manager (user and group codes both equal to one).

The UPF also contains the system accounting information. This information includes the following:

1. Connect time limit for accounting period
2. Connect time limit for session
3. System utilization limit for accounting period
4. System utilization limit for session
5. Total connect time used in accounting period
6. Total system utilization used in accounting period

The user may look at only his accounting information. The System Manager is permitted to inspect any one of any group of user's accounting information.

3.5.5.2 Privileges

IAS supports the following types of privileges:

1) PDS command privileges
2) Time-sharing privileges
3) User Identification Code (UIC) privileges

The PDS command privileges dictate which commands or set of commands a user may issue. There are 16 PDS command privilege classes, examples of which are given below:

1. File manipulation facilities
2. Task manipulation
3. Basic
4. Fortran
5. Submit to batch
6. Device management
7. Dump
8. Librarian
9. System Library tasks
10. Real time commands

The System Manager may place restrictions on the type of task that each user is allowed to run in time-sharing mode. The user may be granted the following time-sharing privileges:

1) To run tasks which make requests to the Time-sharing Control Primitive (see section 3.3.1)

2) To run executive privileged tasks (mapped to Kernel Executive data base, SCOM)

3) To run tasks which issue privileged directives.

In addition, several other miscellaneous time-sharing privileges such as task initiation and control, CLI control, device management, etc. are provided.

The UIC privilege allows a user to use certain system facilities based on the value of the user's UIC. A user UIC consists of a group code and an owner code. A system UIC is one which has a group code of less than or

equal to 10 octal. Such a system UIC allows the user to perform I/O on any device at any level and grants system access rights to files. A UIC with group code and user code both equal to one allows some unique features to be made available to the user. This UIC is normally reserved for system management use.

# 4. SECURE IAS DESIGN CONSIDERATIONS

## 4.1 Overview

This section describes the considerations that have led to the proposed design of Secure IAS. These considerations have been grouped under three categories which represent the major issues in the design of Secure IAS; namely, security, performance, and compatibility.

The security considerations dictate that Secure IAS must be verifiably secure with respect to the DOD security policy. As a result, the internal structure of current IAS must be modified in order to accommodate a Security Kernel and the trusted software. These internal modifications are minimized in Secure IAS in order to provide comparable performance and maximum compatibility with that of current IAS. However, it is unavoidable that a loss of performance and lack of compatibility will result in Secure IAS due to the access privileges currently available in IAS versus the necessity of verifiably enforcing the DOD security policy.

## 4.2 Security Considerations

### 4.2.1 Security Kernel

Enforcement of the DOD security policy in a verifiable manner requires that a security kernel be implemented in Secure IAS. The Security Kernel must mediate all attempts to access the hardware resources. The Kernel must be protected from the unverified software in order to prevent unauthorized alteration of the Kernel. For these reasons, the Kernel alone resides in kernel domain of the PDP-11/70. This permits the Kernel to control access to all hardware resources by explicitly protecting the I/O page. Protection of the kernel domain space is provided by the PDP 11/70 hardware.

The Kernel maintains the security attributes associated with each subject and object in the Secure IAS system. Subjects are tasks and devices while objects are tasks, devices, files, segments, and event flags.

Security attributes for each type of subject and object differ, but an access level is always present. The access level consists of a security level and an integrity level. Both the security and integrity levels consist of a security category (i.e., TOP SECRET, SECRET, CONFIDENTIAL, Unclassified) and a set of security compartments. The security level denotes the degree of protection required against unauthorized observation while the integrity level denotes the degree of protection required against unauthorized modification.

Associated with each task is a set of privileges. These privileges are not to be confused with those of standard IAS. In Secure IAS, privileges denote exemption from certain security axioms as explained in the Abstract Implementation Model (AIM, Appendix A). These privileges are normally associated with privileged, non-Kernel security-related tasks as discussed in the next section.

The security attributes associated with each Kernel supported subject and object are listed in Table 4.2-1.

As a consequence of the Kernel mediating all accesses to the hardware resources, the current IAS executive privilege will no longer be supported. That is, no non-Kernel task in Secure IAS will be able to map directly to the Kernel data base or to the I/O page. Instead, all executive privileged tasks must be modified to interface with the Kernel via a set of Kernel calls. In particular, the device handlers which currently operate as privileged tasks in user domain will now run in Kernel domain. The Time-sharing Executive (i.e., TSS1) will be modified since it will no longer be mapped to the Kernel data base. MCR will be modified to interface with the Kernel.

The Kernel itself will maintain a non-structured, "flat" file system. Thus, the Files-11 file system will be implemented in terms of the Kernel supported file system via the Kernel calls. Within the same computer system, foreign volumes will again be implemented in terms of the Kernel supported file system. Foreign disk volumes from other systems will not be permitted. Instead, a procedure is necessary to transfer foreign system information from disk to tape and then from tape to the Secure IAS system.

Table 4.2-1

Security Attributes of Kernel Supported Subjects and Object

| Subject and/or Object | Security Attributes |
|---|---|
| Tasks | Access level<br>UIC (User Identification Code)<br>Set of privileges |
| Devices | Maximum access level<br>Current access level UIC<br>UIC<br>Discretionary access rights (i.e., read, write, delete, extend) |
| Files | Access level<br>UIC<br>Discretionary access rights |
| Segments | Access level<br>UIC<br>Discretionary access rights<br>Domain (i.e., Kernel, Supervisor, or User corresponding to the three PDP-11/70 hardware supported domains) |
| Event Flags | Access level |

The global event flags will be maintained by the Kernel through Kernel calls associated with setting, clearing, and testing the flags. A set of thirty-two global event flags will exist for each different active task access level in the system. Tasks which are at the same access level (i.e., share the same set of global event flags) are the only tasks permitted to communicate with each other through global event flags. This restriction is necessary because of the system directives which return a flag's polarity before setting or clearing it. Returning the polarity of a global event flag requires observe access to the flag while setting or clearing the same flag requires modify access to the flag. Together, observe and modify access imply that the access level of the task must be the same as the access level of the flag.

Since the kernel alone resides in kernel domain, the supervisor domain of the PDP-11/70 will be utilized by the trusted software and by the IAS Emulator tasks.

### 4.2.2 Trusted Software

The trusted software is a collection of independent tasks that executes with extraordinary privilege in order to provide non-Kernel security-related capabilities. These capabilities are provided by seven functions as outlined below:

1) System Boot - This function transfers from the controlled system disk the certified copies of the IAS Security Kernel and the Terminal Manager into memory and begins system initialization.

2) Terminal Manager - This function is the initial privileged task in the system. It initiates, either directly or indirectly, all other non-Kernel security-related tasks in the system. Additionally, it is the custodian of all terminal devices.

3) Discretionary Authenticator - This function performs the login protocol by prompting the user for both his user name and password.

4) Secure Dialoger - This function provides a secure path for the user to enter protection data and to communicate in a trusted manner with the Security Kernel.

5) File Manager - This function provides a secure mechanism for creating, accessing, modifying, and deleting directory files and ordinary files.

6) Maintenance - This function provides the maintenance facilities to dump/restore files, set the clock, etc.

7) Time-sharing Executive - This function controls the execution of interactive and batch mode programs.

The inclusion of the non-Kernel security-related tasks in Secure IAS will necessitate modifications to the MCR, PDS, and SCI login procedures as well as the system maintenance tasks. In Secure IAS, the Terminal Manager will initially have control of the user's terminal. The login procedure will be completed by the Terminal Manager before MCR, PDS, or SCI is initiated.

System maintenance functions will be performed by the secure maintenance function rather than by system tasks as in current IAS.

### 4.2.3 Time-sharing Control Primitive

The Time-sharing Control Primitive (TCP), as it is currently implemented, represents a potential security problem in Secure IAS. As described in Section 3.3-1, the TCP receives QIO requests from all of the Command Language Interpreters (CLI's). It transforms these QIO requests into Kernel Executive requests and returns error code information to the requesting CLI. The fact that all of the CLI's are connected to the same TCP results in a potentially non-secure communication channel between the various CLI's. Although no security relevant information is currently passed through TCP, the potential exists for such information to be routed through TCP due to deliberate alteration of the TCP and/or inherent errors in the TCP.

There are two possible solutions to this problem. The first one involves identifying all parts of the TCP that handle or route the information to and from the CLI's. Each of these parts would be isolated from the rest of the TCP and formally verified with respect to the DOD security policy.

The other solution eliminates the potential communication channel. This would be accomplished by the Secure Dialoguer (See Section 4.2) initiating a modified TCP for each terminal in the system. Each user task, its CLI, and the modified TCP would execute together in a separate, protected, virtual environment supported by the Kernel.

Each CLI would run under the same copy of the TCP code portion and a different copy of the TCP data portion. Neither portion of any of the modified TCP's would have to be formally verified since the potential communication channel would no longer exist.

In the design of Secure IAS, the second solution is utilized. This choice eliminates the potential communication channel and the necessity for verifying portions of the TCP software.

## 4.3 Performance Considerations

### 4.3.1 Real-Time Processing

Response times for real-time tasks can be critical depending on the particular application. In Secure IAS, the real-time performance will be degraded due to the fact that all hardware accesses must be mediated by the Kernel. In order to provide reasonable response times, real-time tasks will be able to make Kernel calls directly. The set of Kernel calls will be carefully chosen to allow for rapid scheduling of real-time tasks. For example, Kernel calls for declaring and waiting upon significant events will be provided.

### 4.3.2 Interactive and Batch Processing

The scheduling of interactive and batch tasks is not time critical. The response times for these tasks is intended in Secure IAS to be comparable with that of current IAS.

## 4.4 Compatibility Considerations

In order to preserve the external interface between the user and the standard IAS system facilities, an IAS Emulator will be employed. The objective of the Emulator is to maximize compatibility with no compromises in security.

The IAS Emulator will consist of non-trusted software executing in supervisor domain. It will be responsible for maintaining compatibility at several levels: the system directives, the Files-11 system, and the time-sharing user terminal interfaces.

Under standard IAS, system directives are serviced by the Executive Kernel in kernel domain. To maintain the same interface under Secure IAS will require a translation of all standard IAS system directives into Security Kernel calls. This will be accomplished by the Emulator.

The Files-11 file system will be supported by a non-Kernel security-related task, the Directory Manager, as well as the Security Kernel. Security relevant file information in the standard Files-11 system (i.e., file headers) will be moved into the kernel domain. The supporting software (i.e., F11ACP, MTAACP, etc.) for Files-11 volumes is currently executive privileged and therefore must be integrated and modified into Kernel call privileged tasks.

The Terminal Manager is designed to assure a secure communication path between a terminal user and the trusted software. The Command Language Interpreters (CLI's) will no longer have control over the terminals. Instead, all time-sharing user terminals will be under the control of the Terminal Manager.

The login procedure that is currently part of the PDS terminal interface will be eliminated and assigned to a non-Kernel security-related task known as the Discretionary Authenticator.

PDS and SCI of standard IAS are non-privileged tasks executing in user domain and interface with the Kernel Executive via system directives or system tasks. Changes to either at the user level will be minimal.

Certain facilities under PDS will no longer be supported. For instance, users will no longer be able to build executive privileged tasks with the LINK command under Secure IAS; certain system installed tasks will not be accessible to PDS even given the real-time capabilities, etc..

The SCI interface will be restricted to only the System Manager under Secure IAS. This may mean physically isolating the operator's terminal.

## 4.5 ARPANET Connection Design Considerations

The function of an ARPANET connection program is to allow users to communicate to each other in different hosts. This program performs the following basic functions when service between a terminal and a host is required:

- Make the initial connection between the user and the desired host.
- Transmit data over the connection.
- Clear the connection using connection clearing control.

### 4.5.1 Secure ARPANET Connections

There are two ARPANET connection protocols: the current Network Control Program (NCP) and Transmission Control Program (Cerf-77). The protocols, although similar in that they both permit implementation of the above functions, are very different at the lower level. The Transmission Control Program (TCP) allows more in the way of error detection, flow control, and retransmission. Also, the state transmissions for sockets (i.e., CONNECTIONS) in TCP and NCP are different. TCP provides different connection establishment and acknowledgement. The concepts of "pocket", "fragment", "segment", and "letter" in TCP generalizes the concept of "message" in TCP. Perhaps most importantly, TCP has no concept of IMP/host or host/host protocol. All protocol is carried within a single header. For these reasons, TCP protocol is selected.

There are two approaches for a Secure ARPANET connection. In the first approach, data is encrypted/decrypted as it leaves/enters the host computer. In the second approach, data is encrypted/decrypted as it is moved from/to the users address space to/from TCP-accessible storage. The first design is potentially more efficient because encryption and decryption is handled by a front-end processor. The second design potentially requires less trusted code since the TCP can never access clear-text data. Specifically, in the first design, the TCP must manage the flow of data between the user and the network with trusted code, whereas in the second design this is not required. On the other hand, the second design must be concerned with key distribution and management functions within the system.

### 4.5.2 Functional Breakdown

The TCP consists of six interactive functions. These functions are:

- Packet Maker,
- Output Packer Handler,
- Retransmitter,
- Reassembler,
- Input Packet Handler, and
- TCP Command Processor.

The packet maker breaks output data into packets based upon send window size and current buffer availability. It is called by the TCP command processor and the input packet handler.

The output packet handler takes a packet and applies it to the output side of the network hardware interface. The packet is then put on a retransmission queue and a time out function shall be initiated. The output packet handler is called by the packet maker process, the input packet handler, and the TCP command processor.

The retransmitter is called at a fixed time interval. Each network connection is associated with a data structure called a transmission control block (TCB). A TCB contains all pertinent information to a

connection; local and foreign socket number, transmit and receive window, transmit and receive sequence number, retransmission time out, etc.. All TCB's will be examined to determine whether any of their packets have not been acknowledged within the specified time out period. If so, the output packet handler is given the packet(s) for retransmission.

The reassembler examines a TCB for packets which may be coalesced into letters. When a user buffer is full or a letter have been reconstructed, the data shall be passed to the user process through the connection's receive port. The reassembler process shall update the receive left window edges of connections as letters are reassembled. The reassembler process shall be called by the input packet handler process and the TCP command processor process.

The input packet handler is executed when data is received from the network interface. It verifies that the packet is for an existing TCB. If this fails, a "RESET" message shall be returned to the sender. The input packet handler is responsible for control or error information. This includes RESET, ACKS, FIN, SYN, ARQ, RSN, and INTERRUPT commands. Any ACKS in incoming packets shall be used to update send left window edges, and to remove packets from a retransmission queue. The packet sequence number, current receive window size, and receive left window edge shall determine whether the packet lies in or outside of the window. A packet shall be rejected only if all of it lies outside the current window. Specifications for checking packet to window correspondence are found on Page 80 (Cerf-1977). If there are no outgoing packets, input interface control shall construct an ACK message and pass it to output interface control for transmission.

The TCP command processor is the user interface to the TCP. It reads user commands which are input to it. The TCP command processor shall dispatch control to one of the individual TCP command drivers which implements the TCP user commands. User commands should be available to establish and relinquish a connection, open and close a connection, abort a connection, send and receive data, obtain status information about a connection, and initiate the transmission of an interrupt over a network connection.

# 5. SECURE IAS ARCHITECTURE

## 5.1 Overview

The Secure IAS system consists of three components; the Security Kernel, the Trusted Software, and the IAS Emulator. The Security Kernel mediates all accesses to the hardware resources. The Trusted Software provides all non-kernel security-related capabilities. Both the Kernel and the trusted software are verified with respect to security. The IAS Emulator provides an interface between IAS user programs and the Security Kernel.

The proposed architecture of the Secure IAS system is described within the following sections and shown in Figure 5.1-1.

## 5.2 Verified Software

The verified software consists of a kernel domain resident Security Kernel and supervisor domain resident trusted software.

### 5.2.1 Security Kernel

The Security Kernel is a small software module with the following characteristics:

1) Completeness: The access of all users to all data stored in the computer system is mediated by the Kernel.

2) Isolation: The Kernel is protected against unauthorized alteration by other software in the system.

3) Verifiable: The Kernel is designed and implemented in such a manner so as to be amenable to a formal proof of the security properties of the system.

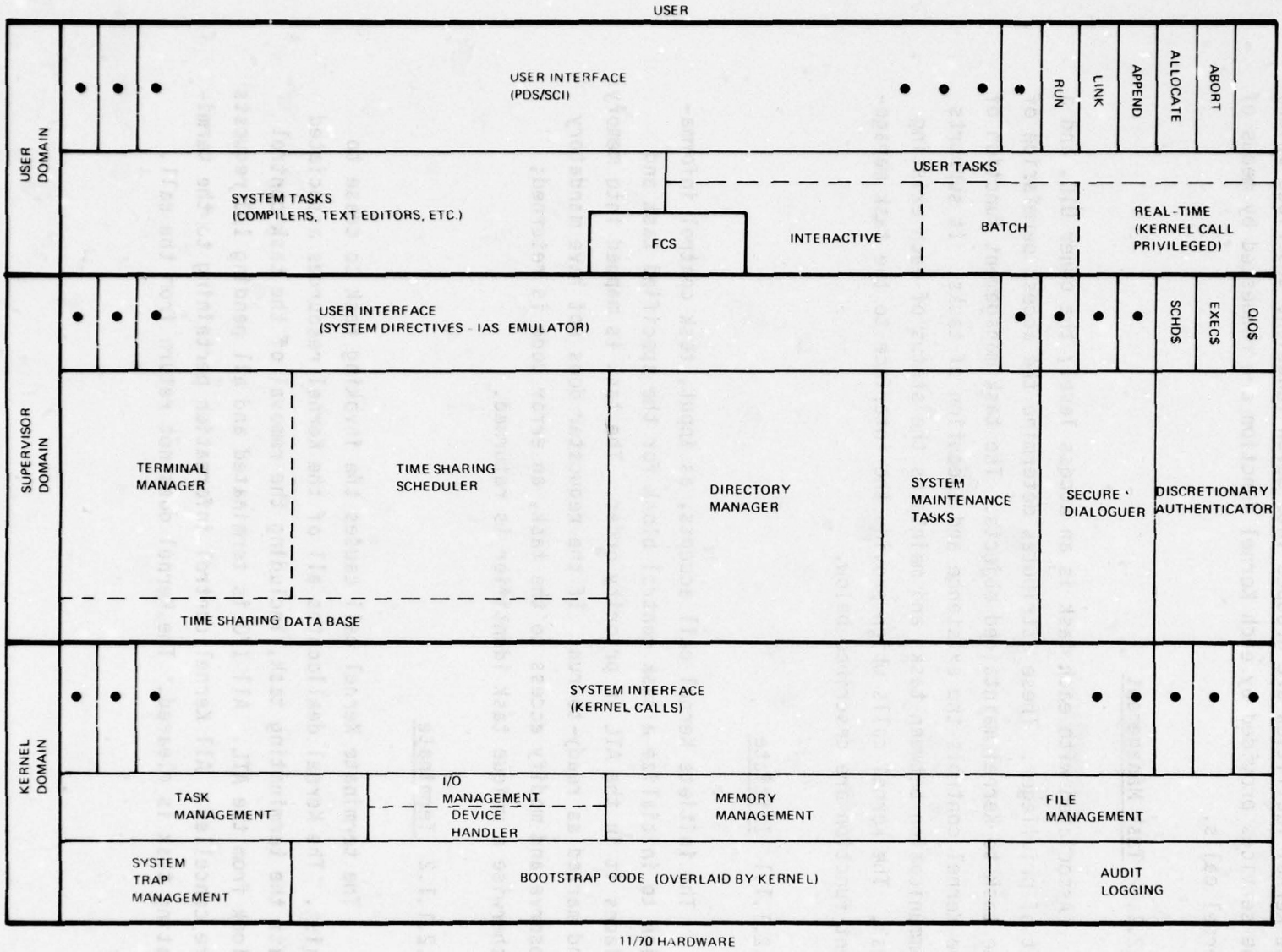



Figure 5.1-1. Secure IAS Architecture

The Kernel facilities are grouped into seven functions described below. The services provided by each Kernel function are requested by means of Kernel calls.

#### 5.2.1.1 Task Management

Associated with each task is an access level, the owner UIC, and a set of privileges. These attributes determine the access permission of the task to Kernel maintained objects. The task management function of the Kernel controls the existence and execution of tasks. It supports communication between tasks and maintains the status of each existing task. The Kernel calls which provide the interface to the task management function are described below.

##### 5.2.1.1.1 Initiate

The initiate Kernel call accepts, as input, task control information to initialize a task control block for the specified task and places it in the ATL in priority order. The task is mapped into memory and marked as ready-to-run. If the requester does not have mandatory observe and modify access to the task, an error code is returned; otherwise a unique task identifier is returned.

##### 5.2.1.1.2 Terminate

The terminate Kernel call causes the invoking task to cease to exist. The Kernel deallocates all of the Kernel resources associated with the terminating task, including the removal of the task control block from the ATL. All I/O is terminated and all pending I/O requests are cancelled. All Kernel control information pertaining to the terminating task is cleared. The Kernel does not return from the call.

#### 5.2.1.1.3 Suspend

The suspend Kernel call causes the invoking task to relinquish control of the CPU for a specified time. The task is marked suspended but will still retain control of the system resources allocated to it. The task's segments, however, may be swapped out to secondary storage while suspended. The task may regain control of the CPU after the expiration of the specified time or through an explicit resume Kernel call.

#### 5.2.1.1.4 Resume

The resume Kernel call causes a specified suspended task to resume execution. If the task's segments have been swapped out, the Kernel will remap the task into memory. The task is marked as ready-to-run. The invoking task must have mandatory observe and modify access to the suspended task.

#### 5.2.1.1.5 Set-Task-Status

The set-task-status Kernel call sets the task control information (i.e., priority, checkpointability, etc.) for the specified task. The invoking task must have mandatory modify access to the specified task.

#### 5.2.1.1.6 Get-Task-Status

The get-task-status Kernel call returns the task control information of the specified task. The invoking task must have mandatory observe access to the specified task.

#### 5.2.1.1.7 Set-Event-Flag

The set-event-flag Kernel call sets the specified event flag and reports its polarity before setting.

Global event flags are implemented by the Security Kernel as a dynamic data structure. Associated with each task in the system is an access level. Inter-task communication via global event flags is restricted to tasks of the same access level. The Security Kernel allocates a unique set of 32 global event flags upon each initiation of a task at a different access level. There can be at most n sets of global flags if there are n active tasks in the system.

Each task has access only to the set associated with its own access level. The Kernel mediates all accesses to the set and is responsible for maintaining the integrity of the data structure.

5.2.1.1.8 Clear-Event-Flag

The clear-event-flag Kernel call clears a specified event flag and reports its polarity before clearing. In the case of a global event flag, the flag associated with the invoking task's access level is cleared.

5.2.1.1.9 Read-Event-Flag

The read-event-flag Kernel call returns the polarity of a specified event flag. In the case of a global event flag, the flag associated with the invoking task's access level is returned.

5.2.1.1.10 Declare-Significant-Event

The declare-significant-event Kernel call causes the Kernel to rescan the Active Task List (ATL) from the top to dispatch the highest priority active task.

5.2.1.1.11 Send

The send Kernel call causes the transmission of a data block of a specified length, known as an Inter-Task Communication (ITC) message.

The ITC message is queued to the specified task in order of the specified priority or the sender task priority. The sending task must have mandatory modify access to the receiving task.

5.2.1.1.12 Receive

The receive Kernel call obtains the next highest priority ITC message on the invoking task's message queue. If no messages are waiting, the invoking task will be notified via a return code.

5.2.1.2 Memory Management

The memory management function of the Kernel controls the creation, deletion, and accessing of memory segments. The attributes of a memory segment include an access level, an owner UID, discretionary access rights, and execution domain. The Kernel calls which provide the interface to the memory management function are described below.

5.2.1.2.1 Allocate-Segment

The allocate-segment Kernel call creates a new memory segment with the specified attributes. The Kernel allocates a set of memory management registers to map the new segment. The segment usage count is incremented. If the memory management registers have been exhausted, the invoking task will be notified via a return code. The access level and discretionary permission of the segment are the same as those of the invoking task.

5.2.1.2.2 Release-Segment

The release-segment Kernel call dissociates a mapped memory segment from the memory management register of the task that has allocated it. The segment usage count is decremented. If the usage count reaches zero, the memory segment is freed for future allocations. Only segments associated with the invoking task may be released.

#### 5.2.1.2.3 Remap

The remap Kernel call allows a task to remap one of its allocated memory management registers to another of its task segments without the reallocation procedure; redefining the segment attributes. The outgoing task segment is not released.

#### 5.2.1.2.4 Set-Segment-Status

The set-segment-status Kernel call allows the invoking task to modify Kernel kept segment control information. The invoking task must have mandatory and discretionary modify access to the specified segment.

#### 5.2.1.2.5 Get-Segment-Status

The get-segment-status Kernel call returns Kernel kept segment control information. The invoking task must have mandatory and discretionary observe access to the specified segment.

#### 5.2.1.2.6 Save

The save Kernel call ensures that all modified segments in memory are reflected on to disk.

#### 5.2.1.2.7 Get-Partitions

The get-partitions Kernel call returns information about a specified partition or, if no partition name is specified, the invoking task's partition. The invoking task must have discretionary observe access to the specified partition.

### 5.2.1.3 File Management

The file management function of the Kernel controls the creation, accessing, and deletion of the Kernel maintained flat file system. The

attributes of a file include a unique file identifier, an access level, and a discretionary access mode. The Kernel calls which provide the interface to the file management function are described below.

5.2.1.3.1 Create-File

The create-file Kernel call initializes the file control information on the specified volume. The access level of the created file is set to the access level of the invoking task. The discretionary access modes are set to read, write, extend, and delete for the system/owner and null otherwise. The owner identity is set to the user identity of the invoking task and the group identity is set to null.

5.2.1.3.2 Delete-File

The delete-file Kernel call releases the file control information and the file data associated with the specified file.

5.2.1.3.3 Open-File

The open-file Kernel call is used to grant the requested access capability to a file. The Kernel returns an open file descriptor upon successfully opening the file. Open file descriptors are maintained by the Kernel on a per task basis. Thus, a unique open file descriptor exists for each file that a task has opened. The requested access may be any combination of read, write, delete, extend, and status update. The appropriate mandatory and discretionary access checking is performed depending on the requested access.

5.2.1.3.4 Close-File

The close-file Kernel call causes the specified file to be inaccessible until the file is successfully opened again.

5.2.1.3.5 Get-File-Status

The get-file-status Kernel call returns the access level, discretionary access mode, owner identifier, group identifier, and file control information associated with the previously opened file.

5.2.1.3.6 Set-File-Status

The set-file-status Kernel call allows the owner of the previously opened file or a task privileged to violate discretionary access controls to alter the discretionary access mode and owner/group identity of the file. If the file was opened for status and update access, the file control information may also be altered.

5.2.1.4 I/O Management

The I/O management function of the Kernel provides the facilities for creating and modifying devices and performing input and output operations. The Kernel calls which provide the interface to the I/O management function are described below.

5.2.1.4.1 Create-Device

The create-device Kernel call associates a logical device name with a physical device unit and initializes the device control information.

5.2.1.4.2 Open-Device

The open-device Kernel call is similar to the open-file Kernel call except that a device, instead of a file, is opened for the requested access. The appropriate mandatory and discretionary access checking is performed depending on the requested access.

5.2.1.4.3 Close-Device

The close-device Kernel call is similar to the close file Kernel call except that a device, instead of a file, is made inaccessible until the device is successfully opened again.

5.2.1.4.4 Get-Device-Status

The get-device-status Kernel call returns the maximum access level, current access level, discretionary access mode, owner identifier, group identifier, and device control information. Device control information may include the value of the system clock, the sense switches, etc.. The device must have previously been opened.

5.2.1.4.5 Set-Device-Status

The set-device-status Kernel call allows the owner of the previously opened device or a task privileged to violate discretionary access controls to alter the discretionary access mode and owner/group of the device. If the device was opened for status update access, the device control information may also be altered.

5.2.1.4.6 I/O-Read

The I/O-read Kernel call initiates a data transfer from a specified file to a specified buffer address. The file must previously have been opened for read or read/write access.

5.2.1.4.7 I/O-Write

The I/O-write Kernel call initiates a data transfer from a specified buffer address to a specified file. The specified file must previously have been opened for write or read/write access.

5.2.1.4.8 I/O-Function

The I/O function Kernel call performs a device specific, non-I/O transfer operation on the specified device. The device must previously have been opened.

5.2.1.4.9 Mount

The mount Kernel call informs the Kernel that a new volume on a specified device is to be made available to users of the system. The invoking task must have mandatory observe and modify access to the device and be privileged to violate discretionary access checking.

5.2.1.4.10 Dismount

The dismount Kernel call informs the Kernel that the volume mounted on the specified device is to be made unavailable to users of the system. The invoking task must have mandatory observe and modify access to the device and be privileged to violate discretionary access checking.

5.2.1.5 System Trap Management

The system trap management function of the Kernel reflects all synchronous system traps and all asynchronous system traps to the IAS Emulator for further processing.

5.2.1.6 Kernel Interface

The software executing outside the Kernel domain interfaces with the Kernel via the Kernel calls described above. In the PDP-11/70, the TRAP instruction is used to implement the various Kernel calls.

#### 5.2.1.7 Auditing

The Kernel is responsible for monitoring and recording, on the audit log file, security-relevant operations such as login attempts (successful and unsuccessful), invalid attempts to access Kernel maintained objects, etc..

The audit log file is an ordinary file whose discretionary owner is the System Manager, whose access level is at system high, and whose discretionary access mode is set as read/write for system/owner/group only.

An entry can be made on the audit log file in two ways. One may be made internally by the Kernel as a result of an invalid access detection while servicing a Kernel call. The other may be an explicit write request to the audit log file via an I/O-write Kernel call by one of the System Manager owned tasks running at system high access level. The audit log file can also be read via an I/O-read Kernel call by one of the System Manager owned tasks running at system high.

### 5.2.2 Trusted Software

The trusted software is a collection of independent tasks that execute with extraordinary privilege in order to provide non-Kernel security-related capabilities. These capabilities are provided by six functions as described below.

#### 5.2.2.1 System Boot

The system boot function transfers certified copies of the Secure IAS Security Kernel and the Terminal Manager from the controlled system disk into memory. The internal Kernel data is initialized after which the existence and status of all Kernel controlled devices is determined. Finally, a task environment for the Terminal Manager is created so that control may be transferred to the Terminal Manager.

#### 5.2.2.2 Terminal Manager

The Terminal Manager is the initial privileged task in the system. It initiates, either directly or indirectly, all other non-Kernel security-related tasks in the system. Initially, the Terminal Manager has control over all devices, including the terminals. The Terminal Manager waits until a user attempts to log-on. At that time, the Discretionary Authenticator is initiated. When a user logs-off or fails to log-in successfully, a message is sent to the Terminal Manager. This message notifies the Terminal Manager to wait for the next input from the appropriate terminal.

The Terminal Manager requires a superset of all privileges so that it can hand some subset of them to each non-Kernel security-related task that it initiates.

#### 5.2.2.3 Discretionary Authenticator

The Discretionary Authenticator is initiated by the Terminal Manager whenever a user attempts to log-in. The Discretionary Authenticator prompts the user for his user name and password. The password is encrypted. If the user name and encrypted password do not match an entry in the password file, a "login incorrect" message is printed on the terminal. After 3 unsuccessful log-in attempts, an audit record is made, a wait is initiated for a specified period of time, and the Terminal Manager is given control of the terminal.

Upon a successful log-in, an audit record is made for accounting and security purposes. The owner identity of the terminal is set to the newly logged-in user and the group identity of the terminal is set to the group of the newly logged-in user. The access level of the terminal is set to the minimum of the maximum user access level and the maximum (terminal) device access level.

Lastly, the Discretionary Authenticator initiates a Secure Dialoguer for the logged-in user.

#### 5.2.2.4 Secure Dialoguer

A Secure Dialoguer exists for each user logged in. It provides a secure path for the user to enter protection data and to communicate in a trusted manner with the Kernel. The Secure Dialoguer is exempt from discretionary controls so that, acting on behalf of the System Manager, it can change the protection data on other user's files.

The user establishes a secure connection with the Secure Dialoguer by typing in a special sequence of three break characters. These characters are intercepted by the Kernel. The Kernel sends an ITC message to the Secure Dialoguer. At this point, the Secure Dialoguer locks the terminal to prevent extraneous I/O and informs the user that it is ready to accept a command. After the user has completed his dialogue, he types a special quit command which causes the Secure Dialoguer to unlock the terminal and return control to the task interrupted by the break sequence.

A dialogue with the Secure Dialoguer may be initiated by a task executing on behalf of the user. In this case, the task transmits an ITC message to the Secure Dialoguer asking for permission to perform some special function (e.g., downgrading a file). The Secure Dialoguer receives the message and locks the terminal. The request is identified to the user and the user is asked to confirm it. The user may execute other commands such as Secure-Print to help him decide whether to confirm the request or not. The user terminates the dialogue by either confirming or not confirming the request. In either case the terminal is unlocked and control is returned to the requesting task.

The operations performed by the Secure Dialoguer include the following:

1) Changing a user's access level for the session.
2) Changing the discretionary permission associated with a file.

3) Changing the owner and/or group associated with a file.

4) Changing a user's password.

5) Printing a file.

6) Downgrading a file.

7) Logging off the system.

As part of the initialization process, the Secure Dialoguer initiates the IAS Emulator for the user task. The IAS Emulator begins execution of the appropriate Command Language Interpreter (CLI).

5.2.2.5 Directory Manager

The Directory Manager supports the directory structure provided by the standard IAS Files-11 system. It provides a secure mechanism for creating, accessing, modifying, and deleting directory files and ordinary files.

Directory files can be one of two types; Master File Directory (MFD) or User File Directory (UFD). A Master File Directory is created by the System Manager for each volume in the system. It is an index to all UFD's on the volume. The access level of all MFD's is system high with the System Manager as the discretionary owner. The discretionary access mode for the system/owner/group is read, write, extend, delete, and search. The world discretionary access mode is null.

The UFD is an index to all of a user's files on the volume. The access level of the UFD is the maximum access level of the user with the System Manager as the discretionary owner. The discretionary access mode is the same as that of the MFD.

Each entry in the MFD and the UFD has an access level associated with it. Since only two levels of directories are supported, the entries in the MFD's and UFD's are at different access levels at or below the access level of the directory itself. In order to read or

search a directory when operating at an access level that is less than that of the directory, the privileged Directory Manager must be initiated.

The Directory Manager ensures that only entries at or below the access level at which the user is operating are read or displayed as a result of a directory search.

Likewise, the Directory Manager must be initiated for operations that require writing on a directory file. Such operations include creating a new directory, creating a new file and entering it into an existing directory, deleting an entry from an existing directory, and deleting an existing directory.

The Directory Manager is privileged to violate the simply security condition, the integrity *-property, the security *-property, and the simple integrity condition. Note, however, that no actual information flow security violation takes place.

The Directory Manager is initiated by execution of the initiate Kernel call.

#### 5.2.2.6 System Maintenance

The system maintenance function provides the facilities to save and restore files, maintain the Kernel file system, and print files. These facilities are accessible only to the System Manager.

The standard IAS utility programs PRESERV and DSC (see Section 3.5.2) provide the facilities for saving and restoring files. In Secure IAS, these utilities must be trusted since they access all files on a particular volume and modify the Kernel file control information.

Maintenance of the Kernel file system is provided by the VFY and BAD utility programs. VFY checks the consistency of the Directory Manager's directory structure. Specifically, VFY checks for any Kernel files appearing in the Directory Manager's directory structure which are not identified as allocated by the Kernel. Similarly, it checks for any Kernel files which do not appear in the Directory Manager's directory structure. The BAD utility determines if any bad blocks exist on a disk

pack. VFY is trusted because it accesses all of the Kernel maintained file system; BAD is trusted because it destroys the previous contents of the disk pack in question.

The DMP utility is used to print the contents of files. It is trusted because it must communicate with tasks at various access levels. If it were feasible to have a DMP utility for each access level, none of them would have to be trusted.

#### 5.2.2.7 Time-Sharing Executive

The Time-Sharing Executive controls the execution of interactive and batch mode programs. Each of the IAS Emulators associated with the user tasks transmits Inter-Task-Communication (ITC) messages to the Time-Sharing Executive in order to schedule time-sharing and batch tasks. The Time-Sharing Executive maintains the various levels of user task importance and urgency that are maintained by the current IAS Time-Sharing Executive. A round-robin queue is implemented for each of the various levels.

After a task has been scheduled for execution, the Time-Sharing Executive issues a Kernel call to initiate the task. Time-slices are implemented by use of Kernel calls to suspend and resume tasks.

The Time-Sharing Executive is trusted because it transmits ITC messages to tasks at lower access levels. There is no real information flow security violation.

### 5.3 IAS Emulator

The IAS Emulator provides an interface to the Security Kernel whose major function is to maintain maximum compatibility with the existing IAS software.

The IAS Emulator is non-trusted software executing in the supervisor domain and is virtualized on a per user terminal basis. It simulates the standard IAS system directives by mapping each of the system directives into Kernel calls. The Emulator data base is protected from user programs through the memory management hardware. The user programs

may trap either to the Emulator or directly to the Kernel; the PDP-11/70 interdomain EMT instruction causes a trap to the Emulator and the TRAP instruction causes a trap to the Kernel. The former interface is utilized to implement the system directives and the latter to implement the Kernel calls. Kernel-call-privileged user tasks may interface directly with the Security Kernel via the Kernel calls thus circumventing the Emulator.

The following sections describe the system directives as they are supported under the IAS Emulator.

### 5.3.1 Task Execution Control Directives

The Secure IAS task execution control directives remain compatible with that of standard IAS. The Emulator accomplishes this by invoking the initiate, terminate, suspend, and resume Kernel calls.

#### 5.3.1.1 EXIT$

The EXIT$ directive terminates the execution of the issuing task. The EXIT$ directive does not differ from that of standard IAS.

#### 5.3.1.2 EXST$

The EXST$ directive terminates the execution of the issuing task and enables the task to return a success or failure condition to the system. The EXST$ directive does not differ from that of standard IAS.

#### 5.3.1.3 RQST$

The RQST$ directive initiates an installed task and executes it if there is enough memory available and if it is of sufficient priority. The RQST$ directive differs from that of the standard IAS in the following ways:

1) The issuing task is prohibited from overriding the specified task's default UIC, unless the requester is the System Manager.

2) The issuing task must have observe and modify access to the specified task.

5.3.1.4 EXEC$

The EXEC$ directive activates a task if the memory required for its execution is currently available. The EXEC$ directive differs from that of the standard IAS in the following ways:

1) The issuing task is prohibited from overriding the specified task's default UIC, unless the requester is the System Manager.

2) The issuing task must have observe and modify access to the specified task.

5.3.1.5 SCHD$

The SCHD$ directive requests a task to be executed at a specified future time and, optionally, repeated periodically. The schedule time is expressed in terms of absolute time-of-day. The SCHD$ directive differs from that of standard IAS in the following ways:

1) The issuing task is prohibited from overriding the specified task's default UIC, unless the requester is the System Manager.

2) The issuing task must have observe and modify access to the specified task.

5.3.1.6 RUN$

The RUN$ directive requests a task to be executed at a specified future time and, optionally, repeated periodically. The schedule time is

expressed in terms of time after the issuance of the RUN$ directive. The RUN$ directive differs from that of standard IAS in the following ways:

1) The issuing task is prohibited from overriding the specified task's default UIC, unless the requester is the System Manager.

2) The issuing task must have observe and modify access to the specified task.

5.3.1.7 CSRQ$

The CSRQ$ directive cancels the execution of the specified requested task by the specified requester task. The CSRQ$ directive differs from that of the standard IAS in the following way:

1) The issuing task must have observe and modify access to the cancelled task.

5.3.1.8 SPND$

The SPND$ directive suspends the execution of the issuing task. A task can only suspend itself. The SPND$ directive does not differ from that of standard IAS.

5.3.1.9 RSUM$

The RSUM$ directive resumes the execution of a suspended task. The RSUM$ directive differs from that of standard IAS in the following way:

1) The issuing task must have observe and modify access to the specified task.

5.3.1.10 ABRT$

The ABRT$ directive terminates the execution of the specified task. The ABRT$ directive differs from that of standard IAS in the following way:

1) The issuing task must have observe and modify access to the specified task.

5.3.1.11 SYNC$

The SYNC$ directive requests a task to be executed at a specified future time and optionally repeated periodically. The schedule time is expressed in terms of a time interval from specified clock unit synchronization. The SYNC$ directive differs from that of standard IAS in the following ways:

1) The issuing task is prohibited from overriding the specified task's default UIC, unless the requester is the System Manager.

2) The issuing task must have observe and modify access to the specified task.

## 5.3.2 Task Status Control Directives

The task status control directives provide the facilities to alter the task execution characteristics. The Emulator accomplishes this by invoking the set-task-status Kernel call.

### 5.3.2.1 FIX$

The FIX$ directive fixes an inactive, installed task in memory. The FIX$ directive differs from standard IAS in the following way:

1) The requesting task must have observe and modify access to the task to be fixed.

### 5.3.2.2 UFIX$

The UFIX$ directive reverses a FIX$ directive that has been issued to make a task permanently memory-resident. The UFIX$ directive differs from standard IAS in the following way:

1) The requesting task must have observe and modify access to the previously fixed task.

5.3.2.3 DSBL$

The DSBL$ directive instructs the IAS system to reject future attempts to run or fix an indicated task. The DSBL$ directive differs from standard IAS in the following way:

1) The requesting task must have observe and modify access to the indicated task.

5.3.2.4 ENBL$

The ENBL$ directive instructs the IAS system to make an indicated diabled task runnable. The ENBL$ directive differs from standard IAS in the following way:

1) The requesting task must have observe and modify access to the indicated task.

5.3.2.5 DSCP$

The DSCP$ directive makes the issuing task non-checkpointable. The DSCP$ directive does not differ from that of standard IAS.

5.3.2.6 ENCP$

The ENCP$ directive makes the issuing task checkpointable after its checkpointability has been disabled, that is, it reverses a DSCP$ directive. The ENCP$ command does not differ from that of standard IAS.

5.3.2.7 ALTP$

The ALTP$ directive alters the priority of the specified active task to the new priority indicated in the directive. The ALTP$ directive differs from that of standard IAS in the following way:

1) The requesting task must have observe and modify access to the specified task.

### 5.3.3 Event Associated Directives

The Secure IAS local event flags remain compatible with that of standard IAS. The implementation of global event flags at the Security Kernel level is, however, greatly altered (see Section 5.2.1.1.7). The Kernel is solely responsible for maintaining the integrity of global event flags and mediates all inter-task communication which uses these flags. The Kernel is invoked by the Emulator to set/clear event flags, declare-significant events, and to send/resume ITC messages.

#### 5.3.3.1 SETF$

The SETF$ directive sets a specified event flag and returns in the DSW, the flag's polarity before setting. The SETF$ directive differs from that of standard IAS in the following ways:

1) If a global event flag is specified, the global event flag corresponding to the issuing task's access level is set.

2) For inter-task communication via global event flags, the communicating tasks must be at the same access level.

#### 5.3.3.2 CLEF$

The CLEF$ directive clears a specified event flag and returns, in the DSW, the flag's polarity before clearing. The CLEF$ directive differs from that of standard IAS in the following ways:

1) If a global event flag is specified, then the global event flag corresponding to the issuing task's access level is cleared.

2) For inter-task communication via global event flags, the communicating tasks must be at the same access level.

5.3.3.3 RDEF$

The RDEF$ directive returns the polarity of a specified flag in the DSW. The RDEF$ directive differs from that of standard IAS in the following way:

1) If a global event flag is specified, the status of the global event flag corresponding to the issuing task's access level is returned.

5.3.3.4 RDAF$

RDAF$ directive returns the polarity of all 64 event flags in the specified 4-word buffer. The RDAF$ directive differs from that of the standard IAS in the following way:

1) The global event flags returned are those associated with the access level of the issuing task.

5.3.3.5 WTSE$

The WTSE$ directive suspends the execution of the issuing task until the specified event flag is set. The WTSE$ directive differs from that of standard IAS in the following ways:

1) If a global event flag is specified, the task is suspended until the global event flag corresponding to the issuing task's access level is set.

2) For inter-task communication via global event flags, the communicating tasks must be at the same access level.

5.3.3.6 WTLO$

The WTLO$ directive suspends the execution of the issuing task until any number of the specified event flags are set. The WTLO$ directive differs from that of standard IAS in the following ways:

1) If global event flags are specified, the global event flags corresponding to the task's access level are checked.

2) For inter-task communication via global event flags, the communicating tasks must be at the same access level.

5.3.3.7 EXIT$

The EXIT$ directive terminates the execution of the issuing task if the specified event flag is not set. The EXIT$ directive differs from that of the standard IAS in the following way:

1) If a global flag is specified, then the global event flag corresponding to the issuing task's access level is tested.

5.3.3.8 MRKT$

The MRKT$ directive causes the system to declare a significant event after a specified interval from the time of issuance. If an event flag is specified, it is cleared at the time of issuance and is set at the time of the event. The MRKT$ directive differs from that of the standard IAS in the following way:

1) If a global event flag is specified, the global event flag corresponding to the issuing task's access level is cleared and set accordingly.

5.3.3.9 CMKT$

The CMKT$ directive cancels one or more mark time (MRKT$) requests made by the issuing task that result in the setting of the specific event flag and/or cause an AST at the specified location. If no parameters are specified, all mark time requests made by the issuing task are cancelled. The CMKT$ directive does not differ from that of standard IAS.

#### 5.3.3.10 DECL$

The DECL$ directive causes the system to declare a significant event and, optionally, sets a specified event flag and return its polarity before setting. The DECL$ directive differs from that of standard IAS in the following way:

1) If a global event flag is specified, the global event flag corresponding to the issuing task's access level is set.

#### 5.3.3.11 WSIG$

The WSIG$ directive causes the suspension of the issuing task until the next significant event occurs. The WSIG$ directive does not differ from that of standard IAS.

### 5.3.4 Informational Directives

All standard IAS Information directives are supported by the Secure IAS Emulator. The Emulator accomplishes this by invoking the get-device-status, get-partitions, get-task-status, and get-segment-status Kernel calls.

#### 5.3.4.1 GTIM$

The GTIM$ directive returns the current time and date in the specified 8-word buffer. All tasks have observe access rights to the system clock. The GTIM$ directive differs from that of standard IAS in the following way:

1) The issuing task must have observe and modify access to the specified buffer.

5.3.4.2 GCOM$

The GCOM$ directive returns information related to the specified System Global Area (SGA) in an 8-word buffer. The GCOM$ directive differs from that of standard IAS in the following ways:

1) The issuing task must have observe access to the global area.

2) The issuing task must have observe and modify access to the specified buffer.

5.3.4.3 GPRT$

The GPRT$ directive returns information about a specified partition, or the partition in which the issuing task is currently executing, in a specified 3-word buffer. The GPRT$ directive differs from that of the standard IAS in the following ways:

1) The issuing task must have observe access to the specified partition.

2) The issuing task must have observe and modify access to the specified buffer.

5.3.4.4 GSSW$

The GSSW$ directives returns the status of the console sense switches in the issuing task's directive status word. The GSSW$ directive differs from that of the standard IAS in the following way:

1) Only the System Manager, the discretionary owner of the console sense switches, may have access to the console sense switches.

5.3.5 Trap Associated Directives

The IAS Emulator handles all synchronous system traps and asynchronous system traps by transferring control to the user specified SST end AST routines within its virtual address space in user domain.

5.3.5.1 IHAR$

The IHAR$ directive inhibits recognition of asynchronous system traps (AST) for the issuing task. The IHAR$ directive does not differ from that of standard IAS.

5.3.5.2 ENAR$

The ENAR$ directive allows recognition of asynchronous system traps (ASTs) for the issuing task. The ENAR$ directive does not differ from that of standard IAS.

5.3.5.3 SPRA$

The SPRA$ directive specifies whether or not power recovery ASTs are desired for the issuing task. The SPAR$ directive does not differ from that of standard IAS.

5.3.5.4 SFPA$

The SFPA$ directive specifies whether or not floating point exception ASTs are desired for the issuing task. The SFPA$ directive does not differ from that of standard IAS.

5.3.5.5 SRDA$

The SRDA$ directive allows the issuing task to specify the AST service routine which is to be executed when any data is sent to the issuing task by another task. The SRDA$ directive does not differ from that of standard IAS.

5.3.5.6 SVDB$

The SVDB$ directive allows the issuing task to specify the virtual address of a table of synchronous system trap service routine entry

points. The table may be used as an inter-task debugging aid. The SVDB$ directive does not differ from that of standard IAS.

#### 5.3.5.7 ASTX$

The ASTX$ directive terminates execution of an asynchronous system trap service routine. The ASTX$ directive does not differ from that of standard IAS.

#### 5.3.5.8 SVTK$

The SVTK$ directive specifies the virtual address of a table of synchronous system trap service routine entry points for use by the issuing task. The SVTK$ directive does not differ from that of standard IAS.

### 5.3.6 I/O Related Directives

Device independence is preserved by the IAS Emulator. The Emulator maintains a per task data base associating a task's Logical Unit Numbers (LUNs) with the unique Kernel device identifiers. All I/O related directives are translated into Kernel I/O calls by the Emulator.

#### 5.3.6.1 GLUN$

The GLUN$ directive fills a 6-word buffer with information about a device to which the specified Logical Unit Number is assigned for the requesting task. The GLUN$ directive differs from that of a standard IAS in the following ways:

1) The requesting task must have observe access to the device.

2) The requesting task must have observe and modify access to the specified buffer.

5.3.6.2 QIO$

The QIO$ directive places an I/O request for an indicated device in a queue of priority-ordered requests for that device unit. The QIO$ directive differs from that of standard IAS in the following ways:

1) The requesting task must have the proper mandatory and discretionary access to the device depending on the type of I/O request.

2) The requesting task must have observe and modify access to the address for the 2-word I/O status block.

3) The requesting task must have observe and modify access to the address for the I/O done AST service routine entry point.

5.3.6.3 QIOW$

The QIOW$ directive performs the functions of both the QIO$ directive and the WTSE$ directive. The QIOW$ directive differs from that of standard IAS in the following ways:

1) The request task must have the proper mandatory and discretionary access to the device dependency on the type of I/O request.

2) The requesting task must have observe and modify access to the address specified for the 2-word I/O status block.

3) The requesting task must have observe and modify access to the address for the I/O done AST service routine entry point.

#### 5.3.6.4 ALUN$

The ALUN$ directive assigns a Logical Unit Number (LUN) to a physical device unit. The ALUN$ directive differs from that of standard IAS in the following way:

1) The requesting task must have observe and modify access to the physical device unit.

#### 5.3.6.5 VSDA$

The VSDA$ directive queues a variable length data block according to priority for a task to receive. The VSDA$ directive differs from that of standard IAS in the following way:

1) The requesting task must have modify access to the receiving task.

#### 5.3.6.6 VSDR$

The VSDR$ directive queues a variable length data block according to priority for a task to receive and requests or resumes the execution of the receiving task. The VSDR$ directive differs from that of standard IAS in the following way:

1) The requesting task must have modify and observe access to the receiving task.

#### 5.3.6.7 VRCD$

The VRCD$ directive receives a variable length data block which has been queued for it in priority order. The VRCD$ directive does not differ from that of standard IAS.

#### 5.3.6.8 VRCS$

The VRCS$ directive receives a variable length data block which has been queued according to priority for it or suspends the requesting task if no data blocks can be retrieved. The VRCS$ directive does not differ from that of standard IAS.

#### 5.3.6.9 VRCX$

The VRCX$ directive receives a variable length data block that has been queued for it according to priority or exits if no data block can be retrieved. The VRCX$ directive does not differ from that of standard IAS.

# 6. VERIFICATION PLAN

## 6.1 Scope

This section contains a verification plan for the verification of Secure IAS. This verification plan defines the verification objectives, the issues to be addressed in the verification, the development components for verifiable software, the verification methodology, and the specific verification tasks to be performed in the development of Secure IAS.

## 6.2 Verification Objectives

Assurance that Secure IAS is verifiably secure with respect to DOD security policy as defined by Executive Order 11652, DOD 5200.1-R, DCID 1/16, and AF 205-1 is obtained through the successful satisfaction of the following verification objectives.

- AF approval of the security model developed as the basis for security verification and traceability to DOD security policy.
- Clear, concise, deductive demonstration that
  - (1) The security kernel behavior conforms to the approved security model--i.e., there is clear traceability of the verification products at each level (top level formal specifications, lower level formal specifications, HOL code, test execution) back through higher level verification products to the security model and that
  - (2) The trusted software behavior conforms to trusted software definition--i.e., clear traceability of verification products at each level (top level formal specifications, lower level formal specifications, HOL code, test execution) back to the trusted software definition provided by the security requirements.
- Production of verification data capable of supporting AF security certification or accreditation of Secure IAS.

## 6.3 Verification Issues

The principal purposes of the verification task are (1) to eliminate (or at the very least, identify) all aspects of Secure IAS which do not satisfy DOD security requirements and (2) to provide persuasive documentation that Secure IAS in its final form does satisfy DOD security requirements (except possibly in certain precisely identified circumstances determined to be acceptable by the AF). Any aspect of Secure IAS which fails to satisfy DOD security requirements is either an error or an approved exception; verification serves to eliminate errors and identify exceptions requiring approval.

Errors in the context of security are not merely aesthetically displeasing; they can have substantial, even dire, pragmatic consequences. It is appropriate, therefore, to marshal considerable diverse resources to the verification task. On the other hand, economic constraints require that each distinct aspect of the verification activity be carefully designed to detect, efficiently, distinct kinds of errors and exceptions.

There are several sources of errors in software; they may be organized into several useful categories (see [Goodenough and Gerhart 75]):

- requirements errors: errors which arise if the stated requirements for the software do not match the actual requirements;
- design errors: errors which arise if the design of the software does not match the stated requirements;
- specification errors: errors which arise if the specification of the software does not match the design in ways critical to the stated requirements;
- construction errors: errors which arise if the (source language) software, as programmed, does not match the software specifications;
- implementation errors: errors which arise if the tools (compilers, loaders, etc.) which produce the running programs actually produce target software which does not match the source software as programmed.

The TRW verification methodology consists of a full range of the known verification techniques in order to discover these kinds of errors efficiently. The techniques include:

- Semi-formal description of the security requirements in terms of a precise mathematical model of security kernels.
- Construction of formal specifications of the Security Kernel and trusted software using a formal specification language (SP-EUCLID) which is closely related to the programming language (EUCLID) in which this software will be programmed.
- Systematic construction of the software in a programming language EUCLID, which not only supports program design by the method of successive refinements of abstraction, but also supports formal deductive reasoning about programs.
- A detailed formal verification methodology by which one may form deductions about the correspondence between requirements (as stated in the model), specification, and programmed software.
- A detailed testing methodology, supported by automated tools, for detecting implementation errors and safeguarding against errors in the formal deductive methodology.

The model provides a bridge between the DOD security requirements, which were not designed to be applied to computer systems, and the specifications for the Secure IAS Kernel. Since it is a precise restatement of security requirements, it can be examined carefully (though informally) to see that it does in fact represent the DOD requirements faithfully. In this way, requirements errors can be eradicated. If the specifications are carefully written to capture exactly the security relevant aspects of the design, there are no specification errors. This can be established by careful (though, again, informal) review of the specifications. The task is simplified enormously since the specification is formulated in a well defined formal specification language having a clearly defined semantics. This fact also allows the use of a precise methodology for formally verifying that the specified kernel satisfies

the requirements of the model. If formal verification is "unsuccessful," it serves to identify potential specification or design errors. If verification is successful, and is performed correctly, these types of errors are unlikely to remain.

Since the Kernel will be constructed in EUCLID, a systematic programming methodology can be established which tends to reduce construction errors. Furthermore, the formal verification methodology provides a check that such errors have actually been eliminated.

Finally, the testing methodology will tend to reduce the number of undetected implementation errors, as well as provide a redundancy test on all other kinds of software errors.

It must be pointed out that formal verification and testing are activities which are themselves subject to error. In fact, both are sure to be in error if not pursued in an extremely systematic way--thus the development of careful methodologies. There are other kinds of errors in these methodologies:

- intrinsic errors - the methodology is unsound, leading to invalid conclusions, or it is incomplete, making potentially valid conclusions unattainable;
- application errors - some steps in the methodologies are correctly or incompletely performed by the practitioner.

The TRW approach deals with both sources of errors. The formal verification methodology is composed of several distinct verification tasks; each of these is intentionally a small deviation from an already well-known verification technique reported in the literature. Therefore, many of the soundness and completeness properties of those techniques are thought to hold for the TRW methodology. A detailed argument has not been constructed to support this belief. However, it was a principal design goal in the construction of the methodology.

Application errors arise from the number and complexity of the steps to be performed. The TRW methodologies are designed to permit use of automated tools to reduce the number of steps to be performed manually. Some of these are already in existence and are considered trustworthy; others need to be built, but are simple in construction (for the most part).

The complexity of the steps to be performed is drastically reduced by the design of SP-EUCLID. The model and the top level specification have a common underlying view of the semantic nature of a security kernel which simplifies the demonstration of their correspondence. Furthermore, the correspondence of types, scoping, and primitive operations between EUCLID and SP-EUCLID greatly simplifies the demonstration of correspondence between the programmed kernel and its specification.

## 6.4 Development Components for Verifiable Software

The security verification of Secure IAS is based upon a model of security kernels (Section 6.4.1), formal specification of the Secure IAS Kernel and trusted software (Section 6.4.2), use of the EUCLID programming language, specifically designed for the systematic production of verifiable system software (Section 6.4.3), and use of testing and deductive techniques, supported by software tools to support the verification process.

### 6.4.1 Security Model

The security model, or the abstract implementation model (AIM), represents in mathematical form the security policy to be enforced by the Secure IAS Kernel. It is derived from the model developed by TRW for KSOS [KSOS 78], and involves some specialization for secure IAS implementation, thus providing a precise statement of the security policy to be used in security verification of Secure IAS.

The Kernel is modeled as an abstract finite-state machine with inputs (representing access requests) and outputs (representing kernel responses) [LaPadula and Bell 73], [Millen 76], [Feiertag, et all 77], and [KSOS 78]. The inputs include information defining (1) the function f to be performed in a particular Kernel call, (2) the arguments a of f, and (3) the process p - an instance of a program (executing on behalf of a user) making the Kernel call. The function, f, and its arguments, a, define the kind of access request being made, and implicitly specify the information objects to which p is requesting access. The AIM does not articulate the specific functions to be performed by each Kernel call. That is left for the formal specification (and kernel design).

The AIM identifies a transition function, T, from abstract kernel state s to state T(s,i), and an output or result function R(s,i), both stimulated by the application of an input i. At any moment, the state includes authorization data to be used in processing future access requests (i.e., kernel calls); this data has been established by prior kernel calls. The security requirements are specified in the AIM as "axioms," relations which impose conditions on the transition and output functions of any satisfactory kernel automaton.

### 6.4.2 Formal Specification Technique

The top level specification technique is based on the Parnas specification methodology [Parnas 72] combined with ideas from Price [Price 73] and the SPECIAL group at SRI [Robinson-Roubine 77]. In this methodology, a software module is specified as an abstract machine (Parnas module) with abstract states, inputs and outputs. An abstract state is specified by a set of primitive V-functions. An input corresponds to an actual call of an O-function (OFUN) or OV-function (OVFUN). An output is returned as the value of a V-function (VFUN) or an OV-function call. The O-function and OV-function descriptions define the state transitions caused by the various inputs.

The lower level formal specification technique is based on the more standard pre and post assertion technology developed in [Floyd 67] and [Hoare 69]. The state of each module is implicit in the value of variables accessible to that module. The result of each function call is specified as a derivation of its value from variables and other function calls. The effects of a procedure are specified by the post assertion as changes in values of variables accessible to the procedure.

The choice of a formal specification language was made after a study existing formal specification languages - GYPSY and SPECIAL - and the informal combinations of predicate calculus and PASCAL used by some investigators. It was concluded that no existing formal specification language was in a state where it is usable as is. Development of both GYPSY and SPECIAL is in a state of flux and the manuals and definition documents available to TRW are incomplete definitions of the languages. Thus a choice of either language would have required TRW to complete the

language definition in order for the language to have sufficient precision to be usable as for formal specification of a verifiably secure operating system; i.e., proof of correspondence of the formal specification to the AIM and of the HOL code to the formal specification are dependent on the semantics of the formal specification language.

GYPSY has the novel feature that it comprises a formal specification language and a programming language having a compatible syntax and semantics, simplifying both the development of programs from specifications written in GYPSY and the verification that such programs conform to their specifications. Since the GYPSY programming language was not in a state where a production quality compiler implementation could be exptected to be available for use, TRW concluded that completing the definition of the GYPSY formal specification language and adapting it to the verification of EUCLID programs would be a significant task. Similarly, TRW concluded that completing the definition of SPECIAL and adapting it to the verification of EUCLID programs would be a significant task. The informal combinations of predicate calculus and PASCAL constructs were also considered to be too imprecise to be usable for KSOS verification.

Accordingly, TRW developed SP-EUCLID [KSOS 78], a variant of the syntax and semantics of EUCLID, for the language for writing formal specifications. SP-EUCLID differs from EUCLID principally in that procedural elements of EUCLID are deleted and a specific definition of assertions (not included in the EUCLID Report [Lampson et al 77] is added. There are a few other minor changes as well. SP-EUCLID provides a syntax and semantics for writing both Parnas modules and for writing Floyd-Hoare specifications for EUCLID routines.

Use of SP-EUCLID for Security Kernel and trusted software formal specifications will simplify development of the EUCLID programs implementing them, because many aspects of a EUCLID program will directly correspond to aspects of the SP-EUCLID formal specifications. The syntax of names and many expressions is the same, so that in many cases only copying, not translation, is involved. For the same reason, verification that a EUCLID program corresponds to a specification in SP-EUCLID is simplified, for one of the main sources of formal verification errors has been translation from the HOL implementation language to the formal

specification language. Problems of scoping, naming, structuring programs, and verifying related semantic constructs will be minimized.

### 6.4.3 HOL For Implementing the Verifiable Software

For implementing the verifiable software, the Security Kernel and the trusted tasks, TRW has chosen the programming language EUCLID. The principle reasons for selecting EUCLID are:

- EUCLID provides features aiding program verification.
- EUCLID has an axiomatic definition, providing precise semantic definitions needed in program verification.
- EUCLID has machine-oriented features needed for efficient operating system implementation.
- EUCLID's structured control and abstract data facilities promote the use of modern software development practices found to contribute to production of reliable software.
- EUCLID is rich enough to be easily extended to a compatible specification language (SP-EUCLID).
- A production quality translator for EUCLID will be available for Phase 1 of the project.

Because of its current position as a well-defined and well-known language, and because of the current existence of its compilers for the PDP-11/70, PASCAL is EUCLID's major competitor. Some of the verifiability features of EUCLID that influence its selection over PASCAL are:

- The scope of procedure effects in EUCLID programs is limited by IMPORT-EXPORT and closed scope concepts. This simplifies verification conditions, because much of verification is concerned with showing that certain variables are not affected by procedures. EUCLID also bars function side effects with syntactic devices.
- EUCLID solves the aliasing problem with syntactic devices and compiler generated assertions, allowing the semantics of procedures and assignment to be stated precisely and simply in axiomatic form.
- EUCLID supports strong type checking, localizes type conversion, and requires the compiler to generate assertions guaranteeing type compatibility.

- EUCLID functions can return non-simple types, eliminating one of the needs for pointers which complicate verification.
- The EUCLID collection concept treats a pointer like an index into an array, simplifying pointer verification. Additionally, EUCLID does not allow the use of routines as parameters.
- The EUCLID compiler will produce legality assertions for such things as index range checking and supports run time checked assertions. EUCLID also provides a built-in syntactic hook for pre- and post-assertions.
- EUCLID solves the variant record problems of PASCAL with its discriminating case feature. The EUCLID case statement is well defined in all cases.

These verifiability supporting features are clearly superior to those available in PASCAL. Most of them are additions or restrictions made to the PASCAL "base" of EUCLID.

The axiomatic definition of EUCLID appears to be the best - the most precise and complete - axiomatic definition of any programming language produced to date. Its published axiomatic definition is designed to match implementation semantics. PASCAL also has an axiomatic definition. It is less complete than the one prepared for EUCLID and is known to contain errors and inconsistencies with the implementation specifications.

Several machine-oriented features of EUCLID make it superior to PASCAL for operating system implementation. It provides the ability to assign a fixed address to a word, a feature needed for PDP-11 input-output. The in-line procedure call feature of EUCLID will aid efficiency, while machine-dependent records allow user definition of field and bit positions within a word. EUCLID modules are a convenient facility for implementing Hoare monitors, should we decide to use them.

Although every attempt will be made to write the Security Kernel and the trusted tasks completely in EUCLID, it is anticipated that some code, estimated at less than 5% of the total, will need to be written in machine language. EUCLID allows the inclusion of machine code when that is necessary.

EUCLID's structured control and facilities for implementing abstract data types go beyond PASCAL's in promoting the use of modern software development techniques. EUCLID does not have a GOTO statement, but has a loop exit statement and other looping constructs. EUCLID has all of the data structure facilities of PASCAL plus the module feature for implementing operations clusters and abstract data types.

The extensibility of EUCLID to SP-EUCLID, the language developed by TRW for formal specification, will substantially aid the verifiability of the HOL code. Since both the formal specifications and the HOL code will be written in compatible languages, with nearly identical syntax and semantics, this will simplify the mapping from implementation level constructs to those on the various specification levels. It will also eliminate one of the main sources of formal verification errors: translation from the HOL implementation language to the formal specification language. Problems of scoping, naming, structuring programs, and verifying related semantic constructs will be minimized, allowing verifiers to concentrate on the overall logic design.

A dominant feature in the selection of EUCLID is the expected availability of a production quality compiler in time for the Phase 2 implementation. The I. P. Sharp - University of Toronto Group has produced a transliterator for "Small EUCLID", which has been successfully installed on the PDP-11/70 at TRW. The EUCLID translator will be available in Phase 1; the code produced by the translator will have relatively good efficiency. Compilers for PASCAL for the PDP-11/70 exist, but are not of exceptional quality. Confidence that the EUCLID compiler will be available has been enhanced by the delivery of a component of the compiler, the parser, in March 1978. This component also appears to be usable as a base for building the SP-EUCLID Syntax Checker tool.

Although GYPSY has most of the verification advantages of EUCLID, plus the addition of a number of automated verification aids, it has several major disadvantages which caused us to disquality it for this project. GYPSY has no published axiomatic definition. It is still a changing research tool and, indeed, we have not seen any reference manual for GYPSY 2. GYPSY lacks some desired operating system features such as the ability to define machine-dependent records. Most importantly, the state of compilers

for GYPSY appears to be less sound; i.e., the availability of a production quality compiler for GYPSY for the PDP-11/70 in time for use in Phase 2 implementation appears to TRW to be in doubt, so that it would not be appropriate to make a decision depending on satisfactory delivery and installation of a GYPSY compiler.

The choice of EUCLID, however, does have some disadvantages: the complete EUCLID compiler, supporting generation of legality assertions, will not be available until January 1979, although a Translator supporting almost all EUCLID features was expected to be delivered in the summer of 1978. EUCLID currently has no automated verification condition generator. PASCAL has the XIVUS tool and GYPSY has a similar, though more integrated, tool. Since Phase 2 does not require verification of the entire security sensitive code, but only a demonstration of its verifiability, the lack of its availability should not affect Phase 1 or Phase 2 schedules. A further problem is that EUCLID has features not found in current programming languages, making it somewhat difficult to learn, and there is no language primer to help; however, documentation produced by I. P. Sharp is improving the situation.

In summary, it is believed EUCLID will serve the purposes of this project well. EUCLID is:

1) A good implementation programming language

2) A rich basis for a specification language, SP-EUCLID, that TRW has developed

3) Reasonably certain to be implemented when, and in the form, needed for Phases 1 and 2 of this project

4) Adequately supported by documentation and verification aids.

### 6.4.4 Verification Tools and Techniques

To support the verification process, TRW will use its OSTEST tool and will modify its NODAL tool so that it will process EUCLID statements. We propose to implement in Phase 2 an SP-EUCLID SYNTAX CHECKER and a TOP LEVEL VERIFICATION TABLE GENERATOR, and to investigate implementation of a TOP LEVEL VERIFICATION CONDITION GENERATOR. We propose that the AF separately fund implementation of a EUCLID VERIFICATION CONDITION GENERATOR.

- OSTEST is a software tool for supporting security testing of operating systems. It currently operates on PDP-11/70 code at the load module level; it analyzes the structure of the machine code and monitors test execution. It collects information on each machine level branch taken and helps guarantee testing completeness.

- NODAL is a software tool designed to aid in the testing of large FORTRAN programs. It analyzes the structure of a program into code segments and transfers, and it instruments the program to record, during test execution, the exercising of these structural elements. A version of NODAL also has been developed for processing JOVIAL (J3) programs. We estimate that four man-months time be required to convert NODAL to process EUCLID programs. We plan to do this conversion in Phase 2 and use NODAL to help us validate the completeness of our testing efforts.

- An SP-EUCLID SYNTAX CHECKER is a software tool for checking the syntax of top level specifications and lower level specifications written in SP-EUCLID. Such a tool is useful to help debug such specifications and can be extended to a top level verification aid as described below. Currently, we have no such tool, but by modifying the EUCLID parser to match the syntax of SP-EUCLID (KSOS 78, Appendix C), we estimate a two-man-month effort will be adequate to implement the tool. We propose to do so at the very beginning of Phase 2.

- A TOP LEVEL VERIFICATION TABLE GENERATOR for Parnas specifications written in SP-EUCLID is a software tool based on the one described in [Ames-Millen 77]. For each externally visible (exported) OFUN, OVFUN, or VFUN, say f, such a tool generates a list of all primitive VFUNs referenced directly or indirectly by f. For each of these references, V, the tool prints out

  1) Those conditions on the argument list of f and the state before the call of f which imply that the result of f depends on V.

  2) Those conditions on the argument list of f and the state before the call of f which imply that V is modified by f.

3) Those conditions on the argument list of f and the state before the call of f which imply that the next state value of V depends on other primitive VFUN references and what those references are.

All of this information can be directly applied to top level verification. Currently, we do not have such a tool, but using the SP-EUCLID syntax checker as a base, we estimate that we can produce one in two man-months. One of these months can overlap with the syntax checker development. We propose to implement the tool early in Phase 2 to aid the top level verification.

- A TOP LEVEL VERIFICATION CONDITION GENERATOR for Parnas specifications written in SP-EUCLID is a software tool which would take top level kernel specifications (annotated with access levels and activity conditions for each primitive VFUN), and produce for each externally visible (exported) OFUN, OVFUN, and VFUN a set of top level verification conditions. Proving these conditions by hand or using an automatic theorem prover would then guarantee that the top level specification satisfies all of the axioms of the Abstract Implementation Model. Again, we have no such tool. We believe that the output from the top level verification table generator described above, combined with hand verification condition generation and proof, is a reasonable top level verification methodology for the Secure IAS Kernel. MITRE, using a similar table generator and a hand proof methodology, was able to verify the MULTICS kernel which is substantially larger than our proposed Secure IAS Kernel [Ames-Millen 77.]

- A EUCLID VERIFICATION CONDITION GENERATOR is a software tool which takes EUCLID programs, including their lower level specification and annotated with inductive assertions and module invariant assertions, and generates for each routine a set of verification conditions which, when proved by hand or by machine, guarantee the consistency of the programs with their specifications. Such a VERIFICATION CONDITION GENERATOR does not exist. We believe that such a tool is entirely feasible for EUCLID because one exists

for GYPSY and for PASCAL (ISI XIVUS system) and because EUCLID was designed for verifiability. We would recommend that the AF fund its development during that time period so that its existence is assured for Phase 3 and for other projects using the EUCLID language. We only propose to verify two routines in Phase 2 and believe that these proofs can be done by hand. Were this tool available before the end of Phase 2, however, we would certainly use it.

## 6.5 Verification Methodology

The verification methodology to be used by TRW in Secure IAS security verification is based on:

- Establishing a security definition used as the basis for the verification, consisting of:
  - An abstract implementation model interpreting the security policy model in kernel implementation terms,
  - Security requirements defining each trusted process.
- Validating the:
  - Abstract implementation model by establishing the correspondence of its axioms and primitive elements to the security policy,
  - Trusted process security requirements by reviewing their correspondence to security concepts.
- Demonstrating that each intermediate development product - top level formal specifications, lower level formal specifications, HOL code, and executable machine code - corresponds to its predecessor product,
  - Employing formal verification techniques on the higher level products - top level formal specifications to security model, lower level formal specifications to top level formal specifications, and HOL code to lower level formal specifications,
  - Using testing in the verification of the executable machine code, relating test cases to corresponding formal verification conditions, and

- Maintaining traceability of the verification at each level to preceding levels, back to the security model and security requirements.

Figure 6.5-1 shows the intermediate development products and verification of correspondence of adjacent level products.

The verification steps illustrated in Figure 6.5-1 are described in the following subsections:

- 6.5.1 - Informal validation of the security model and requirements.
- 6.5.2 - Formal verification of correspondence of top level formal specification (SP-EUCLID specification of top level kernel functions) to security axioms in abstract implementation model.
- 6.5.3 - Formal verification of correspondence of lower level formal specification (SP-EUCLID specification of kernel routines) to top level formal specification.
- 6.5.4 - Verification of trusted software.
- 6.5.5 -Establishing the formal verifiability of the HOL code (EUCLID code of kernel routines) to lower level formal specifications.
- 6.5.6 - Verification of the executable machine code by program testing.

The trace relating the details of these verification steps is discussed in 6.5.7.

### 6.5.1 Validation of the Security Model and Security Requirements

Since the verifiable security of Secure IAS is defined by a security model and security requirements, the model and the requirements must be validated and approved by the AF for use as the basis for IAS security verification. The security requirements define protection features Secure IAS is required to have. The security model represents in mathematical form the requirements (security policy requirements) defining the security policy enforced by the Security Kernel. The remainder of the requirements (trusted software requirements) define functions performed by "trusted software," security sensitive software modules outside the Kernel, but protected by it and verified to correctly perform the required functions.

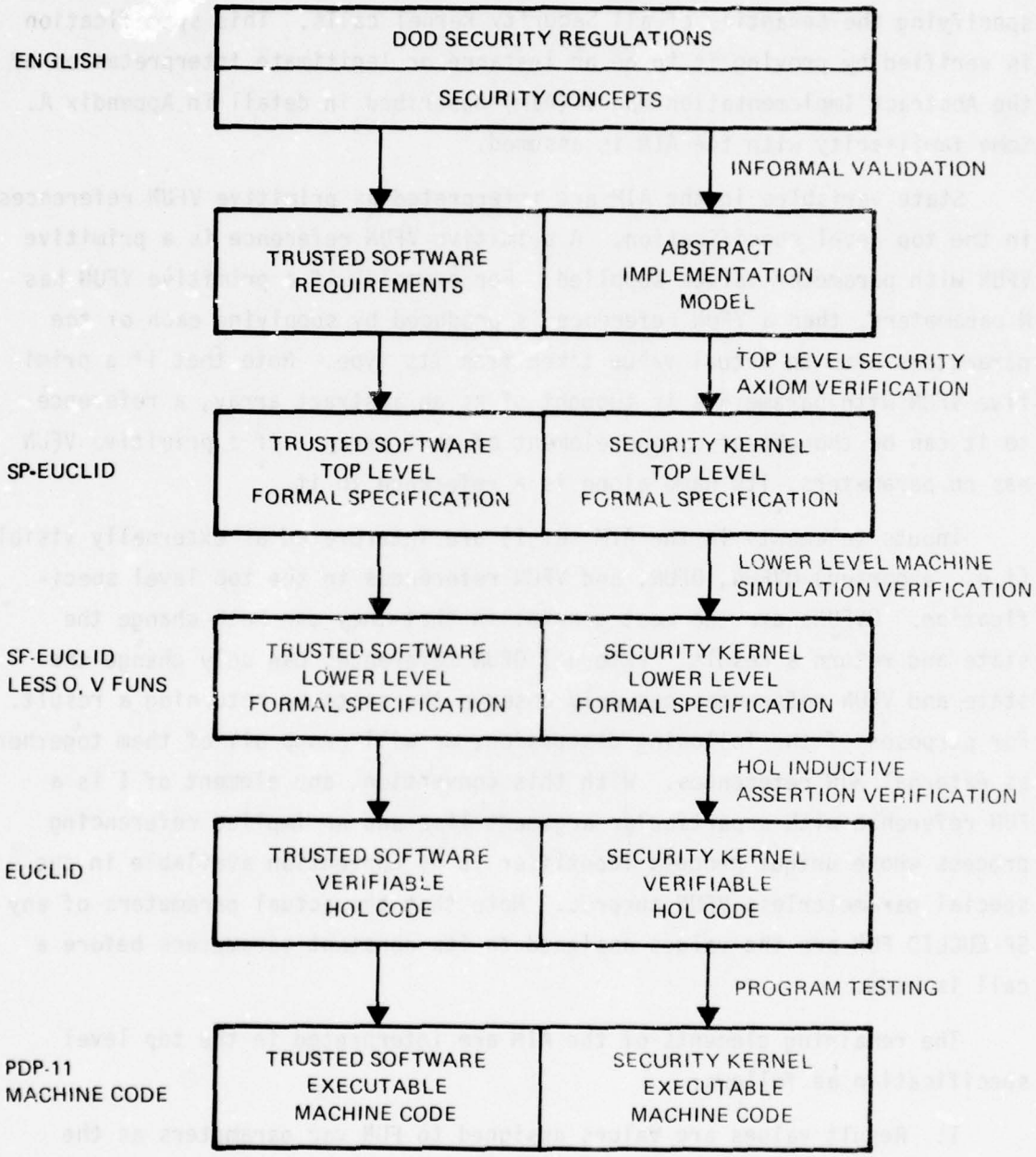



Figure 6.5-1. Verification Methodology Relationships

### 6.5.2 Verification of Top Level Formal Specifications

The top level formal specification of the Security Kernel is a Parnas type specification, written in SP-EUCLID (summarized in Section 6.4.2), specifying the semantics of all Security Kernel calls. This specification is verified by proving it to be an instance or legitimate interpretation of the Abstract Implementation Model (AIM) described in detail in Appendix A. Some familiarity with the AIM is assumed.

State variables in the AIM are interpreted as primitive VFUN references in the top level specification. A primitive VFUN reference is a primitive VFUN with parameter values supplied. For example, if a primitive VFUN has N parameters, then a VFUN reference is produced by supplying each of the parameters with an actual value taken from its type. Note that if a primitive VFUN with parameters is thought of as an abstract array, a reference to it can be thought of as one element of that array. If a primitive VFUN has no parameters, its name alone is a reference to it.

Inputs (elements in the AIM set I) are interpreted as externally visible (i.e., exported) OVFUN, OFUN, and VFUN references in the top level specification. OVFUNs are the most general in that they can both change the state and return a result. External OFUN references can only change the state and VFUN references can only observe the state by returning a result. For purposes of the following discussion, we will group all of them together as external FUN references. With this convention, any element of I is a FUN reference with a particular argument list and an implied referencing process whose unique process identifier is by convention available in the special parameterless VFUN curproc. Note that the actual parameters of any SP-EUCLID FUN are the values assigned to its constant parameters before a call is made.

The remaining elements of the AIM are interpreted in the top level specification as follows:

1. Result values are values assigned to FUN var parameters as the result of a kernel call.

2. The state transition function and the output function of an AIM instruction are specified by the post condition of the FUN to which it corresponds.

3. Security levels, integrity levels, and activity values assigned to each primitive VFUN reference are specified as functions of the state in comments next to each primitive VFUN specification.

4. The AIM initial state $s^{(o)}$ assigns an arbitrary undefined value to most primitive VFUN references and the initially returns value to some others as specified in their definition.

With this interpretation of the AIM elements by the top level specification, the remaining verification step is to demonstrate that the top level specification satisfies each of the AIM axioms.

In [KSOS 78] is presented an example of what is involved in such a demonstration for a particular OVFUN create-file and, hence, all of its references. The methodology involves considering each such FUN and checking the axioms one by one for it. Application of the axioms to a specific FUN involves determining the values of the security level functions, L(s,p) and L(s,x), and the integrity level functions, J(s,p) and J(s,x), for the state s of the system, the specific process p (curproc) making the kernel call, and the state variables x referenced in the argument list a, and by the kernel function f. Since a particular FUN may reference other FUNs, application of the axioms also involves determining the values of the state variables in the chain of FUN references. The axioms are then evaluated for those values of the security and integrity level functions and state variables. Since the AIM axioms are stated as exceptions, evaluation of the disjunction of the axioms to be "true" (i.e., if one or more of the axioms detects an exception) should prevent the kernel function from executing. This could be incorporated in the top level formal specifications by placing all 11 AIM axioms in the exceptions section of every FUN. In practice, however, not all axioms apply to every FUN, for the variables in a particular axiom may not appear in FUN or in any chain of FUN referenced in it. Thus, for each FUN, a smaller set of axioms applies. Security verification of a FUN therefore involves determining the axioms applicable to it and verifying that they are correctly stated in the exceptions portion of the FUN specifications. In the following paragraphs we outline what is required.

In TRW's KSOS Kernel and the Secure IAS Kernel TRW will produce, the mandatory security and mandatory integrity axioms are verified together. To do this, the concept of access level is defined as an ordered pair consisting of a security level and an integrity level. Then the concept of $\leq$ is extended to access levels by defining alevel 1 $\leq$ alevel 2 if, and only if, security-level (alevel 1) $\leq$, security-level (alevel 2), and integrity level (alevel 2) $\leq$ integrity-level (alevel 1). With these definitions, the mandatory security axioms are proved using the access level concept instead of the security level concept. Having done this, mandatory integrity automatically follows.

Let us consider the verification of the mandatory security axioms for some particular OVFUN, say f. First we compile all primitive VFUNs referenced by f directly (in the post condition) or indirectly (by non-primitive VFUNs referenced in the post condition). For each of these references V, we compile:

- Those conditions on the argument list of f and the state before the call of f which imply that the result of f depends on V.
- Those conditions on the argument list of f and the state before the call of f which imply that V is modified by f.
- Those conditions on the argument list of f and the state before the call of f which imply that the next state value of V depends on other primitive VFUN references and what those references are.

We note that all of this information can be compiled automatically by a Top Level Verification Table Generator.

Having compiled this information, we generate top level verification conditions for each of the AIM mandatory security axioms (Appendix A). These conditions are SP-EUCLID boolean valued formulas which, if shown to be true in all states, guarantee the satisfaction of the axioms. The first axiom (Simple Security) states that for each of the primitive VFUN references V and each referencing process p, if access-level (V) $\not\leq$ access level (p), and p does not have privilege $PR_1$, then Observe(p,x) does not hold. The information (1) described in the previous paragraph, plus the properties of the process p (referenced as curproc), and the access-level of V suffice to generate a verification condition. This condition may be a predicate

calculus theorem or it may have to be proved with the help of some top level invariant (a condition true in all states). Similarly, verification conditions needed to prove the other two AIM mandatory security axioms are formed with the help of the information pieces (2) and (3) described in the previous paragraph. These conditions must all also be proved.

The discretionary Observe Axiom and the Discretionary Modify Axiom, since they involve access rights values, need only be verified for those FUNs which reference segment values and devices. This fact limits the proof volume immensely. For those FUNs which reference segment values, the proof amounts to showing that the FUN has exceptions which guarantee the axiom or that the FUN has exceptions which guarantee that a previous discretionary check has already been done.

The Activity Axiom states that no read reference can be made to an inactive primitive VFUN during any FUN call. The information compiled for the Mandatory Security Axioms indicates under which conditions each primitive VFUN is read referenced by a given FUN. To prove the Activity Axiom, it must be shown that these conditions are met only if the given primitive VFUN is active. Generally, FUN exceptions will guard against Activity violations, but invariants may be required to complete some proofs.

The Tranquility and Erasure Axioms are easily shown for most OVFUN calls because most such calls do not alter the security and activity levels of primitive VFUNs. Two obvious exceptions are the create and delete kind of OVFUN. Generally, a create (or similarly an initiate task), OVFUN will activate many primitive VFUNs and assign them new security levels. To prove tranquility, it can be shown that they are the only such security level changes. To prove Erasure, it must be shown that all primitive VFUNs so activated must be assigned values immediately upon activation (before they can be referenced).

Having completed verification of all of the AIM axioms for some FUNs, the next FUN in the top level specification is considered. In fact, all axioms must be verified for all FUNs to complete the top level verification. Not only that, but during the verification process, it will be discovered that there are certain invariant assertions that must be assumed in order

to prove that all of the top level verification conditions are true. When the complete set of invariant assertions is collected, the verifier must go back through each possible FUN and prove that if all of the assertions in the set are true before any call of that FUN, then they are all true after any call of that FUN. Once this is done (and it may take several iterations), the top level verification is complete.

Once a top level kernel specification has been verified against the security axioms of the AIM, security considerations are reduced to showing that the lower levels correctly implement the top level specification. There is no need to verify security axioms against any lower levels.

In the verification of the top level formal specifications, and in all lower level verifications, it is anticipated that errors will be found. For each problem encountered in the verification, the nature of the problem, its diagnosis (i.e., the error, if any, causing the problem) and the resolution of the problem will be documented. Then the portion containing the error will be corrected and the corrected portion reverified.

The following list records a few shortcomings of our top level verification methodology:

- It does not eliminate time as an illicit information channel, but there seems to be a general agreement (e.g., see [Feiertag et al 77]) that an attempt to eliminate this channel leads to a system that is too inefficient.
- As with any proof (even automated ones which depend on the correctness of the proof software and the correctness of its use), it is possible for human errors to occur.

So far in this section, we have been discussing top level verification of the security kernel. Top level verification of the non-kernel security-related software is somewhat different because by their very nature, non-kernel security-related software can be exempt from some of the formal information flow concepts. A non-kernel security-related task may be able to transfer information from a higher to a lower access level, perhaps under direct control of a terminal user, perhaps under other authority. Each non-kernel security-related task is different from the other ones. As each is designed, it must have its own model which will be embodied in its top level specifications. The top level verification of this model will be

informal much as is the verification of the kernel AIM, but one principle should help: Whenever an information flow violation occurs, it certainly must be documented and justified.

Our top level verification methodology is an adaptation to SP-EUCLID of ideas from several sources. The pioneering efforts in the field where made by Jon Millen of the MITRE Corporation and documented in such references as [Ames-Millen 77], [Millen 77a], [Millen 77b], [Millen 77c], and [Millen 76]. Some refinements to the methodology have been made by [Feiertag et al 77], [Neuman et al 76], and ourselves.

### 6.5.3 Verification of Lower Level Specifications

The kernel lower level formal specifications describe a machine which is a more concrete form of the kernel than the top level machine. The machine's state is held in the values of concrete variables (scalars, array elements, record fields) instead of in primitive VFUNs as at the top level. No lower level types are pending. Each lower level routine specification is written in SP-EUCLID, but in the Floyd-Hoare style instead of the Parnas style. For example, each routine is a EUCLID function or procedure (still pending procedural code) instead of an OFUN, VFUN, etc. Each routine has a pre assertion and a post assertion written in the SP-EUCLID assertion language with variables instead of primitive VFUNs referenced for new and old state values. There is an invariant assertion present with each submodule and one for the kernel module as a whole. Every routine and submodule which will appear in the implementation has assertions describing it in the lower level specifications.

Section 6.5.2 has already shown how a top level specification describes an abstract machine.

$$M_u = (S_u, I_u, O_u, T_u, R_u, S_u^{(o)})$$

where the u stands for upper. Here $S_u$ is an abstract set of states, $I_u$ is a set of inputs, $O_u$ is a set of outputs, $S_u^{(o)} \in S_u$ is an initial state, $T_u$: $S_u \times I_u$ $O_u$ is a result (output) function.

The lower level specifications also describe a machine

$$M_L = (S_L, I_L, O_L, T_L, R_L, S_L^{(o)})$$

where the subscript L stands for lower. Again, $S_L$ is an abstract set of states, $I_L$ is a set of inputs, $O_L$ is a set of outputs, $S_L^{(o)} \in S_L$ is an initial state, $T_L$: $S_L \times I_L \rightarrow S_L$ is a state transition function, and $R_L$: $S_L \times I_L \rightarrow O_L$ is a result function. Note that in general, the top level and lower level machines are different, e.g., $S_L \neq S_u$ and $I_L \neq I_u$.

The process of lower level verification demonstrates that the machine described by the top level specifications is implemented (simulated) by the machine described by the lower level specifications. Note that not all lower level routines are considered during this verification because many of them are called to implement the effects of others. In fact, the routines that come into the lower level verification are 1) those that directly implement the externally visible OFUNs, OVFUNs, and VFUNs described at the top level, and 2) those that implement top level non-primitive hidden FUNs. Those routines not involved in lower level verification will, of course, make their appearances during the HOL verifications.

We now give a formal definition of simulation as we will use it. A machine $M_L = (S_L, I_L, O_L, T_L, R_L, S_L^{(o)})$ is said to simulate a machine $M_u = (S_u, I_u, O_u, T_u, R_u, S_u^{(o)})$ if there exists a mapping function (here called map) which has the following properties:

1) Map: $S_L \rightarrow S_u$ maps each state in $S_L$ into some state in $S_u$.

2) Map: $I_L \rightarrow I_u$ maps each lower level input into an upper level input.

3) Map: $O_L \rightarrow O_u$ is 1-1 and onto, and is in fact the identity function.

4) Map: $(S_L^{(o)}) = S_u^{(o)}$

5) For all $s_L \in S_L$ and for all $i_L \in I_L$

$$T_u(map(s_L), map(i_L)) = map(T_L(s_L, i_L))$$

6) For all $s_L \varepsilon S_L$ and for all $i_L \varepsilon I_L$

$R_L$ (map($s_L$), map ($i_L$)) = map($R_L(s_L, i_L)$) = $R_L$ ($s_L$, $i_L$))

The above properties are sufficient to define the simulation of machines, but in order to prove verification conditions at each level, the concept of invariant is required. We introduce the two global invariant assertions $INV_u$ and $INV_L$:

a) $INV_u$: $S_u \rightarrow$ {true, false} is true or false for each $s_u \varepsilon S_u$ and limits the actual state possibilities at the top level. As described in Section 3.2, part of the top level verification involves showing that the truth value of $INV_u(s_u)$ is preserved by each state transition and that $INV_u(s_u^{(o)})$ = true.

b) $INV_L$: $S_L \rightarrow$ {true, false} is true or false for each $s_L \varepsilon S_L$ and limits the possible lower level states.

Having defined the upper and lower invariants, we can state the final requirements on them and the map function to guarantee correct simulation:

7) $INV_L$ ($s_L^{(o)}$) = true.

8) For all $s_L \varepsilon S_L$

$INV_L(s_L)$ implies $INV_u(map(s_L))$

9) $INV_L$ is preserved by each element of $I_L$.

Let us consider a more concrete interpretation of each of the component parts of the map function. First consider map: $S_L \rightarrow S_u$. Recall that each state in the lower specification is an assignment of values to variables, while each state in the top level specification is an assignment of values to primitive VFUN references. Thus, to define the state part of map, we can specify the values of each top level primitive VFUN reference as a function of lower level variables. This guarantees that each lower level state is mapped to some upper level primitive VFUN value assignment.

Part of the mapping takes elements in $I_L$ to elements in $I_u$. Each element element $i_u$ $I_u$ can be interpreted as a triple $i_u = (f_u, a_u, p_u)$ with $f_u$ the name of an externally visible OFUN or OVFUN, $a_u$ an argument list, and $p_u$ the name of the executing process. Similarly, $i_L \varepsilon I_L$ can be interpreted as a triple $i_L = (f_L, a_L, p_L)$ with $f_L$ a lower level procedure name, $a_L$ a lower level argument list, and $p_L$ a lower level executing process name. Then we require that

$$\begin{aligned} \text{map}\ (i_L) &= \text{map}\ ((f_L, a_L, p_L)) \\ &= (f_u, a_L, p_L) \\ &= (f_u, a_u, p_u) \end{aligned}$$

The map function applied to argument lists and process names is the identity, and applied to lower level elements $f_L \varepsilon F_L$, it is a 1-1 onto map $F_u$, the set of externally known OFUN and OVFUN names. Hence, $I_L$ is mapped 1-1 onto $I_u$. In practice, this means that each externally visible OFUN and OVFUN $f_u$ at the top level is implemented by some unique procedure $f_L$ in the lower level specifications.

The abstract model we have given so far implicitly requires that types at the lower level be mapped into types at the top level. In practice, all pending types at the top receive concrete type values at the lower level. Other than this, there is a 1-1 mapping of types and their values from the lower level to the upper level. Also, non-primitive VFUNs at the top level are are duplicated by equivalent functions at the lower level. The lower level functions are mapped into the corresponding VFUNs by map.

Consider the proof that $\text{map}(s_L^{(o)}) = s_u^{(o)}$. The proof need only deal with those primitive VFUNs which have initially returns clauses. For each of these, we evaluate its associated mapping function by plugging in the lower level initial values. If the resulting value matches the initially returns clause, the proof is complete. There are at least three ways this can fail to happen:

1) The mapping functions are incorrect.

2) There has been a failure to assign correct inital values to lower level variables.

3) The lower level specifications do not implement the top level machine.

In any case, the appropriate revision must be accomplished.

Next, consider the proof that for all $s_L \varepsilon S_L$ and for all $i_L \varepsilon I_L$

$$T_u(map(s_L), map(i_L)) = map(T_L(s_L, i_L)).$$

This proof is the heart of the lower level verification effort and is a demonstration that each externally visible OFUN and OVFUN commutes as shown in Figure 6.5-2. That is to say that for some $s_L \varepsilon S_L$ and $i_L \varepsilon I_L$, it makes no difference whether: 1) $s_L$ is first mapped to the top level and map ($i_L$) is applied to it, or 2) $i_L$ is applied at the lower level and the result is then mapped to the top level. The proof is generally carried out by considering each OFUN or OVFUN $f_u = map(f_L)$ separately and having separate cases for judiciously chosen partitions of the arguments and states.

Consider the proof that for all $s_L \varepsilon S_L$ and for all $i_L \varepsilon I_L$

$$R_u(map(s_L), map(i_L)) = R_L(s_L, i_L))$$

This proof is required because the lower machine may simulate the state transitions of the upper machine and yet not produce the same outputs. The first part of the proof is to show that the upper and lower instances of each externally visible function have the same number and types of outputs. This can be easily done syntactically since types have been mapped 1-1 from lower to upper specifications. The next part is carried out by considering each externally visible OVFUN and VFUN separately, and again doing a case analysis of state and argument partitions. This part of the proof will generally be done along with the previous part.

The last three parts of the proof involve the invariants $INV_L$ and $INV_u$. Note that these invariants are used in order to complete other parts of the lower level verification. In fact, to complete other proofs, more invariant pieces may be conjoined to early versions of $INV_L$ and $INV_u$. Thus, the invariant property proofs are left until last, showing that:

- $INV_L(s_L^{(o)}) = true$ is a simple matter of plugging in initial values specified in the lower level specifications and evaluating

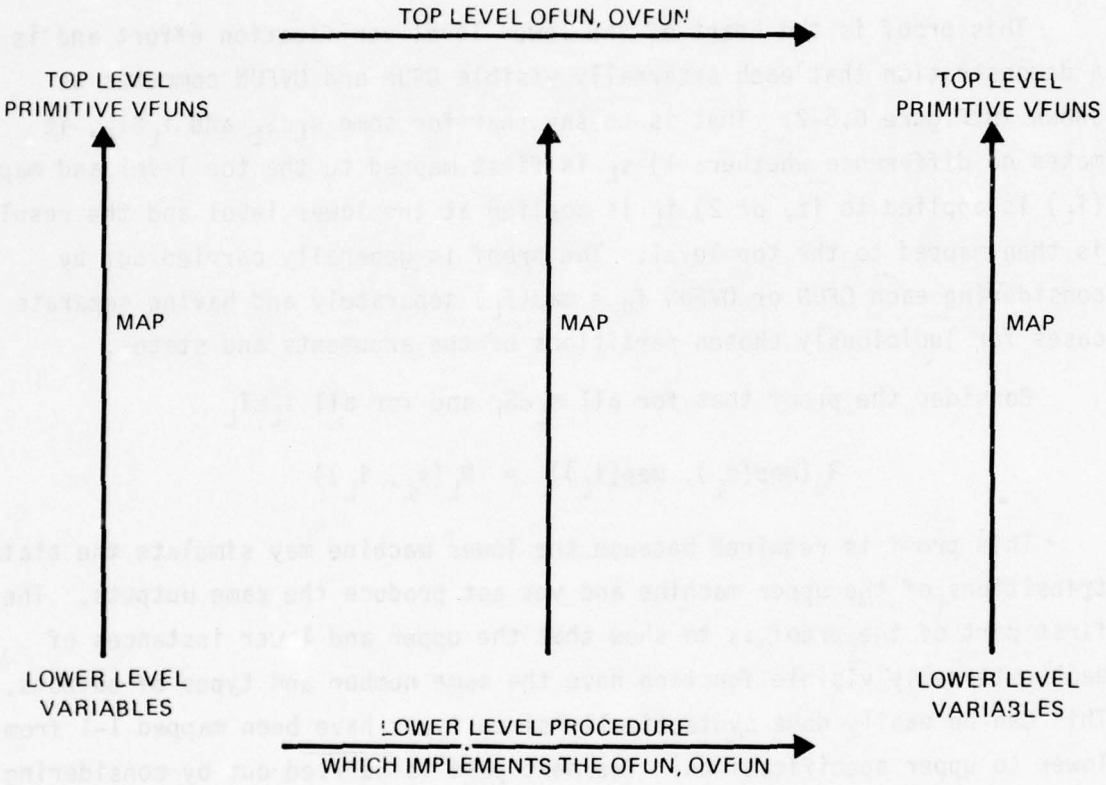



Figure 6.5-2. Commutativity of T

- For all $s_L$ $S_L$ $INV_L(s_L)$ implies $INV_u(map(s_L))$ may simply require some substitutions, but in general, will be much more difficult
- $INV_L$ is preserved by each element of $I_L$ requires a case analysis similar to that mentioned in the previous two paragraphs.

The trusted software top level and lower level specifications are somewhat different from those of the kernel. It should be noted, however, that lower level verification is like that just described, and the same principles apply to the lower level verification of trusted processes.

The sources which have influenced our lower level verification methodology include [Kemmerer 77], [Hoare 72], [Robinson-Levitt 77], and [Wulf et al 76]. The last source, which describes verification of Alphard forms, was probably the most influencial.

### 6.5.4 Verification of Trusted Software

Secure IAS trusted software is a set of security related software modules residing outside the kernel environment. Because of their security sensitivity, they must be verified to be correct. Additionally, some of the trusted software modules perform functions requiring access to information at different security levels and different integrity levels, and hence appear to violate some of the AIM axioms. Although they violate the formal aspects of the security axioms, they comply with the security requirements the axioms represent, for the functions they are required to perform do not result in a security compromise. Therefore, they must be verified to perform their functions correctly, and they are given extraordinary privilege: an exemption from the security axioms they appear to violate.

Because of its security sensitivity, all trusted software is formally specified.

The trusted software top level formal specifications, defining the functions the trusted software is required to perform, are verified by carefully reviewing them with respect to their English specification definitions, and specifically examining them with respect to the security axioms from which they are exempt. Since trusted software is protected by the security kernel, security verification relates only to the axioms from which

the specific trusted task is exempt. The security axiom exemption review involves relating the security levels of all outputs to the security levels of the information accessed and assuring that no security violation occurs in forming those outputs.

The correctness of the trusted software top level formal specifications having been established, the remainder of the trusted software verification follows the process defined for the security kernel: verification of the lower level formal specifications, verification of the EUCLID code, and testing the executable machine code.

### 6.5.5 Verification of the HOL Code

The HOL implementation of the security kernel and trusted tasks will be written in the programming language EUCLID, chosen for its capabilities for producing formally verifiable code.

Verification of the HOL code will be aided by the following practices:

- EUCLID code will be mainly restricted to EUCLID structures known to be relatively easy to verify. TRW will avoid excessive use of parameterized types, exporting of module types, and user defined zones.
- The HOL code will be designed so that one routine in the code corresponds to each routine in the lower level formal specifications.

Accordingly, a mapping will be developed of routines in the HOL code to routines in the formal specifications. Because of the design practices to be followed, this mapping will be one-to-one. Verification of the HOL code will therefore involve proving that each routine in the HOL code correctly implements its corresponding routine in the formal specification.

The basic HOL routine verification methodology has evolved from the papers of Floyd [Floyd 67] and Hoare [Hoare 69]. Some of its variations are discussed in [Hantler-King 76]. Dijkstra [Dijkstra 76] has articulated a methodology to prove termination of loops. Some of the highlights of these ideas are discussed in the paragraphs that follow.

The inductive assertion technique allows one to prove partial correctness of a EUCLID routine (i.e., if the routine terminates execution, it will produce the formally specified effects and/or result). In order to apply the technique to a EUCLID routine annotated with pre and post assertions, each loop must be cut by adding to its body an assert line containing an inductive assertion. Determining where to make each cut and developing the correct assertion for each cut point requires human insight. The process is aided, however, by TRW's functional programming methodology which we will apply. Every executable path between any two assertions (pre, post, or assert) is then considered. It must be shown that if the assertion at the start of the path is true before executing the path, then the assertion at the end of the path will be true after executing all statements along the path. The TRW methodology, as do most methodologies, forms a verification condition as an intermediate step in the proof. The verification condition is a predicate calculus formula which implies the truth of what is to be shown. To complete the proof, the verification condition is shown (by a human or automatic theorem prover) to be a predicate calculus theorem or to follow from some global invariant of the system.

Verification conditions can be produced in a number of similar ways. One of them is the standard method described in [Hoare 69]; it requires an axiomatic definition of the HOL. A second method was invented by Dijkstra and is described in [Dijkstra 76]; it uses a variation of axiomatic semantics in which the meaning of each HOL statement is defined as the function which transforms any post assertion into its weakest pre-assertion. There is also the symbolic execution method of Deutsch described in [Hantler-King 76].

Partial correctness does not imply that a routine is guaranteed to terminate. A separate proof is required for each loop, recursive or iterative, to show that it terminates. Using the methodology of Dijkstra [Dijkstra 76] to prove termination, the verifier invents a non-negative integer-valued function of the state for each loop. To prove that a loop terminates, it it then suffices to show that its associated function is always decremented by at least one each time through the loop. The creation of such a function for each loop generally requires insight and we know of no way to automate the process.

The inductive assertion technique so far described assumes non-interacting processes. If a common variable can be changed by another process executing in parallel, then some extensions to the methodology must be provided. Owicki has developed a methodology in [Owicki 75] and [Owicki-Gries 76] for handling the problems of parallel execution within the Floyd-Hoare framework. The Owicki methodology, although still in its infancy, provides a basis for verifying parallel executing processes. Although our initial design objectives call for a kernel with non-interruptible kernel call executions, it is possible that some of the kernel calls may be of sufficiently long duration to require, for performance reasons, their interruption. If so, those kernel calls requiring interruption will be organized in non-interruptible steps, with interruptions only by device kernel calls allowed at these steps. We will use the Hoare monitor technique to handle parallel execution relationships. We will apply the Owicki methodology supplemented by the concepts of Howard for proving monitors [Howard 76].

Recall that some routine specifications are not verified against top level specifications by the lower level verification methodology because they are specifications for routines called to implement the effects of other routines. These specifications are verified during HOL code verification of their calling routines. If the calling routine A of some routine B cannot be verified using the code and specification for A and the specification for B, then either the code for A or the specification for B is generally at fault and must be changed. When all calling routines for B have been verified, its specification is then implicitly shown to be correct.

In a paper on the fallibility in applications of modern programming methodologies [Gerhart-Yelowitz 76], Gerhart and Yelowitz make a number of suggestions to help guarantee the correctness of specifications. Actually, these suggestions can be applied during top level and lower level verification; but they become especially useful during HOL verification to help detect as yet uncaught specification errors. We will follow these suggestions whenever feasible:

1) Make all assumptions explicit. This includes assumptions about interruptibility, whether termination is assumed, etc.

2) Structure the formal specifications to read like informal specifications if possible.

3) Apply some tests to the specifications, including the Absurd Program Test (trying to find the shortest program satisfying the specifications and seeing whether it does what is expected), the case test (breaking the specifications into cases and determining for each case whether its specification matches intuitive concepts), and the alternate specification test (writing an alternate form of the specifications and convincing oneself that it is formally and intuitively equivalent to the original specifications).

4) Get an independent and knowledgeable person to apply the above suggestions as well.

### 6.5.6 Verification of the Executable Machine Code

A substantial burden for establishing that the delivered product meets security requirements, with security kernel behavior conforming to the approved security model, rests on the security testing, which accordingly must be exceptionally thorough. The security testing methodology to be employed in the development phase achieves the desired degree of thoroughness. It involves an objective proof that all segments and executable segment pairs in the security kernel and trusted software have been executed in the testing and it relates each test case to specific portions of the formal specifications and security model, providing a final link in the demonstration that the Secure IAS security kernel and trusted software execution conform to the security model, correctly enforcing DOD security policy. This security testing will also complement the formal verification of the HOL code so that the combination of formal verification and security testing will provide exceptionally strong assurance that Secure IAS will provide multilevel security in appropriate DOD operational environments. Testing to verify Secure IAS compliance with compatibility and performance requirements is described in Section 7.

The principal elements of the security testing methodology to be applied to Secure IAS are:

- Testing each routine to meet an objective measure of test effectiveness and relating correct execution of each test case to verification of specific components of a corresponding function in the lower level formal specifications.
- Integration of the routines into functionally significant and testable increments, verifying that the software in each increment operates satisfactorily before integrating the next increment.
- Testing the final integrated Secure IAS system, demonstrating that security kernel and trusted software execution enforces the DOD security policy defined in the security model.
- Testing the EUCLID compiler, providing assurance that it correctly compiles EUCLID code.

This testing will be supported by TRW's automated test tools - NODAL and OSTEST - which have been shown, by their use in the development of deliverable software, to substantially increase the thoroughness and cost-effectiveness of software testing.

6.5.6.1 Unit (Routine) Testing

TRW has found that thorough testing of each software unit (routine) is fundamental to testing effectiveness. By finding and removing errors at this early stage, it simplifies testing of the software integration increments and the final system. The unit testing methodology consists of the following:

- Each routine is programmed to implement a specific function in the lower level formal specifications. The formal specification of that function is used as the basis for interpreting the testing of that routine.

- Analyzing the structure of each routine, using the NODAL tool, in terms of in-line code segments and transfers (executable pairs of segments).
- Using the structural data developed by NODAL and applying TRW's Functional Programming Methodology, each executable logic path in the routine is identified along with the corresponding set of inputs (input domain partition) that causes the logic path to be executed.
- Associating the input domain partition corresponding to each logic path with a functional requirement defined by the formal specification.
- Test cases for the routine are developed by choosing inputs from the input domain partitions (at least one from each partition).
- Correct execution of a test case from an input domain partition demonstrates that the routine satisfies the functional requirement (post condition) for the input of the test case and by inference for all inputs in the input domain partition.
- Monitoring test execution with NODAL, which records the segments and segment-pairs executed, computes the fraction of segments executed, and notes the segments not executed. Using this data and the input domain partition definitions, test cases are developed to collectively exercise all segments and executable segment pairs, assuring that all code segments and branching conditions have been tested.
- Any routine containing machine code and selected others will be tested using the OSTEST tool, which operates on code at the load module level, providing a measure of test effectiveness related to the structure of the machine code.

- For each test case where an error is detected, the error will be diagnosed and corrected, usually involving changes to a limited number of code segments. That the error has been corrected will be demonstrated not only by executing the corrected code with the test case that detected the error but also with at least one more test case, involving (if possible) an input from an input domain partition corresponding to the corresponding logic path containing the corrected code but also containing at least one segment not in the logic path for which the error was detected. This approach to testing corrected code has been shown to detect and correct most errors introduced in the process of correcting detected errors.

The result of this testing will be exceptionally thoroughly tested routines, assuring that essentially all errors local to the routines have been detected and corrected at the unit test level.

By associating each test case with an input domain partition and logic path a corresponding in the routine, and with a corresponding functional requirement in the formal specification, the testing provides executable evidence that the routine correctly implements the specified lower level function. This provides a correspondence between unit test results and verification conditions in formal verification.

6.5.6.2 Incremental Integration

Collections of related tested routines are integrated into "increments" performing definable testable functions.

Testing each increment involves assembling the individually tested routines into the increment, getting the increment to operate, and using NODAL to measure test effectiveness and to aid in generating test cases.

The testing of software increments will detect interface errors not evident in the unit testing and produce, at the end of testing of each increment, solidly operating software usable as a base for integrating the next increment.

#### 6.5.6.3 Security Testing

Security testing will make use of test data collected in unit and incremental integration and will execute test cases specifically designed to demonstrate security requirement compliance.

Security testing will involve performing tests to demonstrate that Secure IAS:

- Complies with each security requirement in the Secure IAS A requirements Specification,
- Enforces each security policy axiom in the Abstract Implementation Model,
- Preserves security in each kernel call, system call, trusted task operation, and specified sequences of system calls and kernel calls.

Many of the problems in computer operating systems have been related to the handling of tables. Accordingly, the security testing will include tests to demonstrate that each table containing data affecting security is handled satisfactorily, including response to overloading.

A final part of the security testing will demonstrate that classical generic security flaws causing security problems in conventional operating systems are not present in Secure IAS.

#### 6.5.6.4 EUCLID Compiler Testing

TRW will test the EUCLID compiler upon its delivery and installation on the PDP-11/70 at TRW. Although this testing will not be exhaustive, it will conform to practical standards of compiler testing at TRW, which have been shown to be adequate for the production compilers used in TRW's scientific computing and business data processing. This testing plus the testing performed by I. P. Sharp Associates and the University of Toronto Computer Research Group should provide a high degree of confidence in the compiler. Additionally, use of the compiler during software development will provide further evidence of compiler performance. The results of the compiler testing and any problems encountered in the use of the EUCLID compiler during Secure IAS development will be documented.

### 6.5.7 Verification Trace

In the verification, data will be collected at each verification level and related to verification data at the next higher level:

- Security requirements and security model components to security regulations and security concepts.
- Top level formal specification functions to security requirements and security model components.
- Lower level formal specification functions to top level formal specification functions.
- HOL code routines to formal specification functions.
- Routine test case execution to test verification condition for each logic path.
- Security requirement test case execution to security requirement demonstrated.

This data allowing trace of each verification result back to the security requirements and security model should do much to promote confidence that the Secure IAS implemented is in fact "secure" and should provide a data base useful to the AF in any security certification or accreditation activities.

## 6.6 Verification Tasks

The specific tasks to be performed in carrying out Secure IAS verification, in accordance with the verification methodology defined in Section 6.5, are listed below.

### 6.6.1 Validation of the Security Model and Security Requirements

The security model and computer system security requirements have been traced to the specific paragraphs of Government security regulations they formally represent, and to concepts of security policy enforcement. Section 2.4 of this report is the result of this effort.

### 6.6.2 Development of Top Level Formal Specifications Tools

This task will specify and develop the top-level verification tools specified in Section 6.4.4. These will be developed in Phase 1.

### 6.6.3 Verification of the Top Level Formal Specifications

This task will establish the correspondence to the security model and security requirements. This will be a Phase 2 activity.

### 6.6.4 Verification of the Lower Level Formal Specifications

During Phase 2, the lower level specifications will be verified to correspond to the top level specifications in accordance with the mapping described in Section 6.5.3.

### 6.6.5 Establishing the Verifiability of the HOL Implementation

The establishment of the verifiability will be performed following the completion of coding for selected routines in Phase 2.

### 6.6.6 Verification of the Executable Machine Code

The code for the trusted software will be tested in both Phase 1 and Phase 2.

# 7. DEVELOPMENT PLAN

## 7.1 Scope

This section contains a development plan for development of Secure IAS. This development plan defines the tasks to be accomplished and method to be used to develop and deliver Computer Product Configuration Items (CPCIs) and appropriate support computer resources. As a management plan, this development plan defines the project objectives. It analyzes and breaks down the work to manageable tasks and provides a schedule for the performance of those tasks. It describes the organizational structure; allocates resources; selects programming standards and guidelines; defines and ensures product development, test, integration, and acceptance; and provides for management monitoring and control of the work effort.

## 7.2 Project Objectives

This paragraph states the objectives of the development plan. The Secure IAS development program involves the integration of a security kernel with the IAS operating system to achieve an operational data processing system having verifiable security, comparable performance, and compatibility with current IAS services.

The specific objectives of Secure IAS are as follows:

- Enforce the Department of Defense (DOD) security policy through a security kernel based on an abstract model of security, allowing processing of DOD classified information in a multi-level security mode.
- Provide users and system services compatible with corresponding services of the current IAS operating system, version 6 of RSX-11D and version 1 of IAS.
- Provide adequate performance, compared with current IAS, and
- Be verifiably secure with respect to DOD security policy.

## 7.3 Development Phases

It is recommended that the development phase be divided into two phases. The first phase will develop a prototype system; while the second phase will develop a full productional system based on the prototype.

The specific tasks of the first phase will be:

- Continued detailing of the design of the system
- Formally specifying the security related components in a formal specification language for the high-level specification.
- Implementing a prototype security kernel based IAS.
- Testing the prototype and assessing both performance and compatibility with existing IAS.
- Developing a detailed Phase 2 implementation plan.

The specific tasks of the second phase will be:

- Detailing of the design of those services not provided by the prototype.
- Modifications of the high-level formal specifications of the security related components.
- Developing formal low-level specifications for the full production system.
- Implementing a full productional system based on the prototype.
- Verifying both the high-level and low-level specifications.
- Testing the implemented system for compliance against the security and service requirements.

## 7.4 Technical Management of Software Development

Technical management of the software development process is supported by a variety of management tools. These tools provide timely visibility into software status and allow for prompt and effective technical direction. These tools and techniques include normal project progress reporting and reviews, techniques for surfacing development problems, formal reviews and reviews in

accordance with MIL-STD-1521A, security management plans, configuration management plans, and quality assurance plans.

### 7.4.1 Project Progress Reporting and Reviews

Progress reporting will consist of the monthly progress reports as called for in the contract. The monthly progress reports will report status, open issues, and plans.

Work package managers will prepare weekly status reports relating to each of their major areas of responsibility. These reports will deal with performance regarding schedule and cost, accomplishments, plans and problems requiring action at a higher level.

The project manager will hold weekly status meetings with each of the work package managers. These will be working sessions where problems are discussed, actions assigned, and status reviewed. Each month the project manager will meet with upper management to discuss overall contractual performance specifically in the areas of cost, schedule, status of deliverables, and problem areas.

### 7.4.2 Software Development Problem Surfacing

The major tool which will be utilized for identifying software development problems will be the Unit Development Folder (UDF's). Work package managers will meet with responsible programmers and review status against planned schedules on a weekly basis.

In addition to the weekly work package manager's review, UDF's are open at all times for independent quality assurance audit. These audits will be performed by the quality assurance organization and their findings reported directly to the project manager.

In addition to the inherent control provided by the Unit Development Folder, there will be an informal design walk-through early in the development cycle. These design walk-throughs will provide an independent assessment of any potential software development problems.

A final technique which will be utilized for problem identification is the discrepancy report. After the responsible programmer has completed code and unit testing, software is baselined and placed under configuration control. Any subsequent changes will require preparation of a formal discrepancy report.

### 7.4.3 Formal Reviews

TRW will produce software according to MIL-STD-490 standards. These standards form a natural baseline for the standard reviews in accordance with MIL-STD-1521A. The formal reviews conducted in Phase 1 will be the Preliminary Design Review and the Critical Design Review. During Phase 2, all formal reviews will be conducted.

#### 7.4.3.1 Preliminary Design Review

The Preliminary Design Review is the formal review of the basic design concept for Secure IAS, which establishes a preliminary design approach and the implementation and test plans necessary to proceed into detailed design and development.

The following items will be covered:

a. Demonstrate that every requirement in the B5 specification has been properly accounted for (and is traceable) in the design.

b. Ensure that the design is complete, consistent, feasible, maintainable, and testable.

c. Each software module must be identified as being trusted, untrusted, or part of the Security Kernel.

d. Evaluate the adequancy of the design tradeoff studies and preliminary performance estimates substantiating basic design approach.

e. Review implementation planning.

f. Identify any open issues/problems.

#### 7.4.3.2 Critical Design Review

The Critical Design Review is the formal review conducted on each Secure IAS software CPCI. The design will be described in sufficient detail to allow in-depth review by all parties.

The following items will be covered:

a. Demonstrate that every requirement in the B5 specification has been properly accounted for, and is traceable to the detailed design.

b. Ensure that the detailed design is complete, consistent, feasible, maintainable, and testable.

c. Provide a complete tree structure (top down) of the detailed modules showing their adherence to the initial system security design. This includes a historical trace that the design furnished at CDR implements, and does not violate that provided at PDR. This includes the identification of each module belonging to a trusted software, the Security Kernel, or untrusted software.

d. Evaluate the design evaluation and tradeoff studies and performance estimates substantiating the detailed design.

e. Review current, detailed implementation planning and acceptance testing planning for completeness.

f. Identify and discuss any critical issues.

#### 7.4.3.3 Test Readiness Review

The Test Readiness Review will be conducted in Phase 2 and is an informal review established by TRW and not required by MIL-STD-1521A. The TRR is an internal review to review development test results and evaluate preparations for qualification testing, including the CPCI configuration control approach/procedures, prior to performing qualification testing.

The specific purpose of the meeting is to:

- Review/evaluate, for completeness and adequacy, the testing accomplished to date.
- Evaluate the preparedness to commence qualification testing.
- Determine if schedules are realistic and necessary tools, facilities, and personnel are available.

7.4.3.4 Functional Configuration Audit

The Functional Configuration Audit (FCA) will be conducted in Phase 2 and will be used to verify the CPCI's actual (test) performance compliance with the B5 Development Specifications' requirements. Test data will be reviewed to verify that the CPCI's met all of the requirements associated with its allocated baseline.

7.4.3.5 Physical Configuration Audit

The Physical Configuration Audit (PCA) will be conducted in Phase 2 and is the formal examination of the coded version of the CPCI against its technical documentation and of the configuration management records pertinent to the CPCI's in order to establish the Product Baseline. TRW will provide to the government the final drafts of the C5 Product Specifications 30 days prior to PCA.

7.4.4 Security Management/Configuration Management

The integrity of the products to be produced under this contract will be a major input to the security certification of Secure IAS. They must be correct not only in the technical sense but they must be protected so that:

a. The security model, to which the correspondence of the formal specifications is verified, is not changed from that approved by the government;

b. The formal specifications to which the correspondence of the HOL code is verified are not changed from the formal specifications approved by the government;

c. The HOL code verified to correspond to the formal specifications is not changed from the approved HOL code;

d. The products delivered are not changed from those produced.

Accordingly, as part of its project and configuration management, TRW will provide an effective security management of these items and the verification activities to protect against unauthorized modifications. This involve

a. All personnel working on the project will have at least DOD Secret clearances.

b. The work will take place in a cleared facility having controlled access.

c. The configurations of the products will be carefully controlled, involving:

- The master copy of the model.
- The master copy of the formal specifications and approved HOL code will be kept in a secure place. Working copies will be duplicated from "master" copies.
- During the verifications, careful control procedures will be used.
- As each portion of the code is verified, the formal specifications and approved HOL code of that portion as they appear in the verification results will be checked against the master copies and the verification results placed under security control.

Further details of the security management effort will be specified in a Security Plan. In addition, a clandestine vulnerabilities analysis wil be performed after the completion of the design. This analysis will identi potential vulnerabilities of Secure IAS.

### 7.4.5 Configuration Management

During Phase 1, TRW will prepare a Configuration Management Plan which establishes the series of Secure IAS baselines and methods of controlling these baselines. A preliminary draft will be prepared in Phase 1 within one month of the Phase 1 startup.

### 7.4.6 Quality Assurance

During Phase 1, TRW will prepare an informal Quality Assurance Plan. This plan will be prepared within one month of Phase 1 startup. This plan will be provided and concurrence with RADC will be requested.

### 7.4.7 Software Standards and Procedures

An important feature of the Secure software is the use of detailed, enforceable standards and procedures. The standards and procedures will be prepared well in advance of any actual coding. The coding standards and procedures are divided into those with which compliance is required (waivers of compliance can be granted in exceptional circumstances), and those regarde as simply recommendations to the programmers. During Phase 1, a preliminary set of standards will be developed for the prototype version and submitted fo RADC approval before coding begins. In Phase 2, these will be updated as necessary.

## 7.5 Development Methodology

The program development methodology to be employed on this project is ar adaption of TRW's basic program development methodology to the special requir ments of developing a Secure IAS. This methodology generally follows the sequence in the software development process illustrated in Figure 7.5-1. It is based on six techniques that TRW has found by experience to be necessary in software development:

- plan the entire software project,
- establish a firm development baseline,

- develop software incrementally to minimize risk,
- keep product documentation current, complete, and correct,
- implement disciplined test planning and execution, and
- integrate formal verification with the development activit

### 7.5.1 Plan the Total Software Project

Planning is essential to the success of any software project a results in an orderly software development. Proper in-depth planni provides a clear definition of the work to be performed, enabling a firm commitment by performers to complete the work task on schedule, within cost, and with the required performance. Detailed work plans also provide the means for measuring progress through budgets, schec and earned value. The Project Work Authorization is the formal vehi within TRW for obtaining the firm commitment of the performers. A d implementation plan for Phase 1 is contained in Section 7.6.

### 7.5.2 Establish a Firm Baseline

Secure IAS will follow the software development process illustr in Figure 7.5-1. These steps, completed sequentially, are fundament the successful and efficient production of complex software. Each s serves as the baseline for the succeeding step and the baseline is u at the time the next step begins. When a problem is encountered, it rarely necessary to go back more than one step to correct it. In th way, the disruptive effects of redesign are kept to a minimum.

Each step is responsible for producing a set of baseline docume The baseline documents include the following:

- requirements specifications,
- design specification including formal specification,
- software code,
- implementation plan,
- verification plan,
- configuration and quality assurance plans, and
- test plan and procedures.

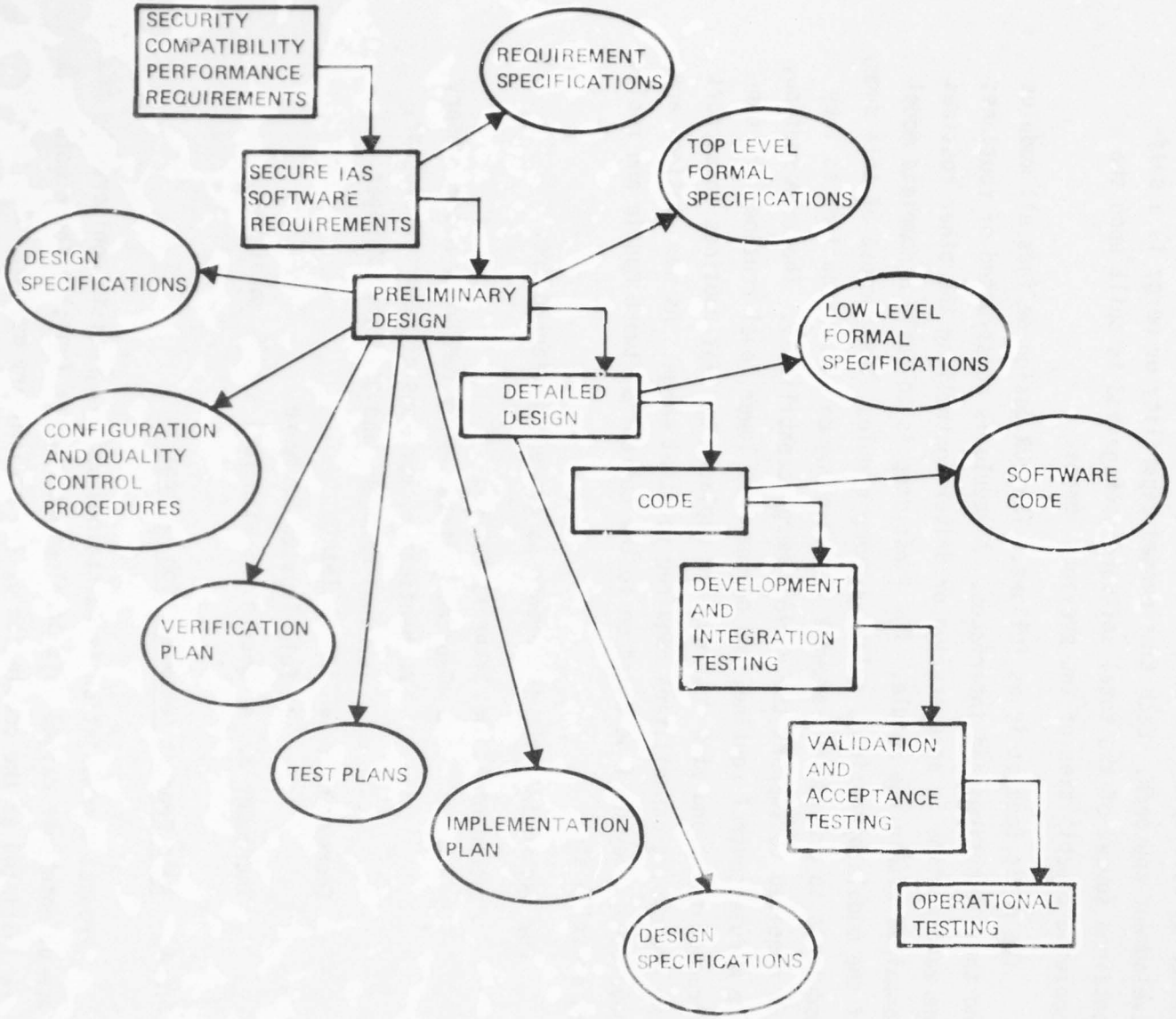



Figure 7.5-1. Software Development Process

AD-A070 141 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
MULTILEVEL SECURITY FOR INTELLIGENCE DATA PROCESSING SYSTEMS.(U)
APR 79 J COTRELL, C SHU, G SHORT F30602-77-C-0119
UNCLASSIFIED RADC-TR-79-88 NL

3 OF 3
AD
A070141



END
DATE
FILMED
7-79
DDC

### 7.5.3 Develop Secure IAS Incrementally

This software development technique is concerned directly with the minimization of risk at the integration and test phase. This is accomplished by developing the software units and modules in a time-phased development approach. Each time-phased capability or merge is a self-contained subset of the total software package and is built upon the completed capabilities of the previous merges.

The basic concept is as follows. The PDR design defines all modules and their functions and interfaces. A module is constructed of routines; one main routine is an executive or driver controlling the other routines contained within the module. The first step is to build a complete model of the module operations, called a dummy module. The purpose of this dummy module is to provide a framework for "hanging on" other routines as they are required to support the various merge capabilities. The main routine, the module control routine, and one or more lower level routines are constructed first and stub routines are replaced by real routines to support the processing capabilities required for that merge. As new routines are developed, they will be added on to the current software module and replace an existing stub.

For Secure IAS, the increments as currently planned are:

Increment 1:
- Security Kernel
- Trusted Software -- System Boot, Discretionary Authenticator, portions of Secure Dialoguer, Terminal Manager, and Directory Manager.

Increment 2:
- IAS Emulator
- Full Trusted Software

Increment 3:
- Secure IAS with full system maintenance.

### 7.5.4 Keep Product Documentation Current

Product documentation is maintained and progress is monitored by the use of notebooks called Unit Development Folders (UDFs). These notebooks are initiated at the completion of the preliminary design. They are the vehicles for managing the design, coding, and development test phases,

recording current status, and documenting the detailed design, code, and results of development tests. These notebooks are maintained throughout all phases of software development to ensure documentation is current, complete, and correct.

A UDF will be set up for each logical programming subdivision (e.g., one or more routines). At the routine level, notebooks will be maintained by the programmer; at higher levels, they are maintained by the assigned work package manager.

The content, current status, and schedule are reflected on the cover sheet of each notebook. Appropriate entries are made on this cover sheet by the responsible programmer or work package manager. At each stage of the detailed design, the technical content is reviewed by an independent technical reviewer as well as by the WPM. The notebooks are audited by the product assurance organization and are available for review by RADC.

During detailed design, the notebook includes the following information:

a. Description of the requirements satisfied by this unit of code, including a mapping of the detailed design to the B5 specification requirements.

b. A description of the design.

c. Interfaces with other routines.

d. Unique assumptions and constraints governing the detailed design.

When completed, the contents of the notebook are transferred to paragraph 3.2 of the C5 specification. Any change which affects the applicable task must be processed through the responsible work package manager. On approval, the data supporting the change are entered into the appropriate notebook.

During the coding phase, each notebook is again expanded to include a listing of the code. This listing is updated throughout the development period. During this phase, change control is exercised against the notebook content by the work package manager.

At completion of coding, unit level testing is conducted in accordance with the provisions of the applicable test plans and procedures.

During unit level testing, the notebooks are again expanded by including the following information for each routine.

a. Description of the unit level test cases. Testing is based on the B5 specifications requirements.

b. An independent engineering review (by someone other than the developer) to determine that the code will perform as specified and that the proposed test cases satisfactorily demonstrate the requirements.

c. Test case results which demonstrate that the routine is debugged and unit level testing is complete. An updated listing is included.

d. An updated description of the design for inclusion in the C5 specification.

On successful completion of unit level testing, product assurance certifies test case acceptance and the code is incorporated into the master routine library.

### 7.5.5 Perform Disciplined Test Planning and Execution

Disciplined test planning and execution consists of the following activities:

a. Allocating requirements to test levels,

b. Developing test plans,

c. Developing test procedures,

d. Performing development level testing,

e. Performing system level testing, and

f. Reporting testing results.

These activities are described in the subsequent paragraphs.

#### 7.5.5.1 Allocation of Requirements to Test Levels

The following criteria will be used to allocate requirements to the development or system test levels:

a. If the requirement is totally contained in a single routine or module, it is allocated (in most cases) to the development level for testing.

b. If the requirement is significantly affected by interfaces with other routines (even though it satisfies (a) above), it cannot be tested sufficiently at the development level.

c. If the requirement is satisfied by more than one routine, it is allocated to the system level.

Each requirement is analyzed in the context of the above allocation criteria to select the best suited level of testing. Documentation of the allocation of requirements to test levels is accomplished via the Test Evaluation Matrices (TEMs).

TEM is a data base program used for maintaining requirements testing status. It utilizes TRW's GIM I data management system. It provides detail requirement traceability from the B5 specifications, the test level at which the requirement will be tested, and the current status of testing of the requirements.

#### 7.5.5.2 Test Plans

The Test Plan document will provide for each component a definition of the component, the scope of testing activities, and the relationship to the other components in Secure IAS. Additional documentation which is used in conjunction with the document will be referenced by document number and title for ease of identification. All terms, mnemonics, and abbreviations peculiar to the document will be defined in a glossary.

The test program management will be discussed. A definition of the verification test team, its duties, and responsibilities will be referenced. Software resource requirements prerequisite to the verification test effort will be referenced. The hardware configuration and any auxiliary hardware devices which are required to support the test effort will be specified.

The verification test approach will be included to provide the test philosophy, test structure, test identification and descriptions, as well as the test objectives. A requirements traceability matrix will be provided to identify all the requirements, which are imposed on Secure IAS, and the test procedures in which they will be demonstrated. The schedules included in the plan will address the time frame in which the verification effort is to take place as well as the specific test dates.

7.5.5.3 Test Procedures

Test procedures describe how the test is to be conducted and what test drivers and supporting software are required.

In providing the method by which the Secure IAS software is to be demonstrated, the following will be included in each test procedure:

a. The test name and identification.

b. A general statement of the test objective.

c. The verification method to be used in demonstrating the requirements specific to the test.

d. A list of requirements.

e. The prerequisite for test execution.

f. Identification of input/output items.

g. Details of any physical operations that must be performed by the verification and test personnel.

h. Acceptance criteria, i.e., the pre-established data and events that must occur to constitute acceptance of Secure IAS software's ability to satisfy the requirements imposed upon it.

i. Appendices to the test procedures. The contents of these appendices shall include:

1) detailed input data, and
2) detailed predicted output data.

### 7.5.5.4 Development Level Testing

Development Testing is the first level of testing to be performed. It consists of unit testing and integration testing. The function of these tests are as follows:

a. Unit Test. Unit testing is the lowest level of testing performed and its purpose is to assure that the module or routine executes correctly in accordance with the design.

b. Integration Testing. The development integration testing goal is to integrate all the unit tested modules into a full executable program. Of primary importance is the assurance that the interface characteristics between modules are handled correctly.

Individual tests at the unit level are initiated as soon as each routine is coded and an error-free compilation is made. Routines are tested to verify the accuracy and adequacy of the software logic branches, computations, data handling, and internal interfaces. When a routine has been completely tested, interfacing routines are integrated and tested as a single unit.

For those requirements being verified at the development level, documentation within the Unit Development Folders (UDFs) explicitly relates requirement verification to test cases and includes a detailed description of constraints, procedures employed, and expected/actual results.

The requirements allocated via the TEMs for each routine are used as a checklist to ensure that testing for all required capabilities has been accomplished. The software test tool NODAL will be used to evaluate the comprehensiveness of the tests and to indicate whether or not further tests are required.

NODAL is a tool which will be adapted to the approved HOL to aid testing at the unit level source programs. NODAL analyzes the structure of a program into code segments and transfers, and instruments the program to record, during test execution, the exercising of these structural elements.

Product Assurance group audits the development testing with primary emphasis on adherence to established programming standards and procedures and test plan.

The development phase of testing is completed when the software is turned over to the system test group.

7.5.5.5 System Level Testing

The goals of system level testing are to complete integration of the software components, fully exercise the interfaces of the Secure IAS system, and verify the system against the allocated requirements.

The specific system level test activities are outlined below:

a. Integration Testing. Integration testing is the initial verification that the software design satisfies the requirements of the B5 specifications.

b. Qualification Testing. Qualification testing is the formal verification testing, executed under a controlled environment to assure that the software satisfies the requirements of the B5 specifications.

c. Performance Benchmark Testing. This testing is a special demonstration test to verify that the performance of Secure IAS is comparable to current IAS.

d. Security Verification Testing. This testing is designed to specifically demonstrate compliance with security requirements.

Qualification testing will be according to the approved test plans/procedures and be aided by the use of TEMs. Each requirement from the approved B5 Specifications will be entered into the data base. A test procedure will correspond to each test requirement and an indicator will specify at what testing level the requirement will be verified.

Performance testing will use IBM's Teleprocessing Network Simulator (TPNS) to generate SCRIPTs. The SCRIPTs will be those approved by the RADC. TPNS, an IBM program product, is a network test driver that simulates terminals. Through the SCRIPT language provided, test scenarios will be generated and statistical data will be collected to be used in performance, component, and system level testing.

Security testing will be in accordance with that specified in the Verification Plan. Test data will be selected by associating a test case having a functional capability of Secure IAS with the corresponding functional requirement from the formal specifications. This provides a basis for asserting, following correct execution of a test case, that the functional capability of the program meets the formal specification.

System level testing is conducted under formal conditions with the software and related documentation formally controlled. Included in the documentation requirements are formal system level test plans and system level test procedures.

Product Assurance takes an active part in system testing activities and will monitor ongoing tests for conformity with the test plans and test procedures documents.

#### 7.5.5.6 Test Reporting

System test activities will be reported daily and summarized weekly. This is possible by merely summarizing the TEM's data base. This process has been completely automated. The final summarized listing of this data base will serve as the final test report.

### 7.5.6 Formal Verification

The formal verification activities are detailed in the Verification Plan (Section 6). The personnel responsible for the verification tasks will be members of the development team to ensure that the development activities are supportive of verification principles. Formal verification will occur on the top level formal specification, the low level formal specifications, and selected HOL routines.

## 7.6 Work Breakdown Structure

The following paragraphs describe the organization and the precise work breakdown structure for the Phase 1 development. The project has been broken into three work packages. They are Project Management Support, Development, and Testing.

### 7.6.1 Work Package 1 - Project Management Support

This work package consists of those activities which are global to all the other work packages in support of the project manager.

Task 1: Project Management

- Direct development effort
- Provide detailed technical status to customer
- Coordinate interface with all organizational elements (internal and external) to assure satisfaction of all Secure IAS requirements
- Update plan
- Prepare and conduct reviews
- Review and approve all deliverable items
- Prepare minutes of all required meetings
- Prepare monthly status reports

Task 2: Systems Engineering Support

- Support all areas of the project
- Support PDR, CDR meetings
- Review, comment, and provide inputs to all deliverables
- Close out PDR and CDR meeting action items

Task 3: Project Control

- Review the costs and financial control of the project
- Prepare monthly financial statement

Task 4: Conduct Formal Reviews

- Prepare PDR and CDR data packages
- Satisfy requirements of MIL-STD-1521A
- Conduct formal reviews
- Generate and coordinate meeting minutes

Task 5: Conduct Project Review Meetings

- Provide technical communication
- Review action items from previous meetings
- Identify problems
- Levy action items for problem resolution
- Produce minutes

Task 6: Prepare Programming and Specification Standards

- Preparation of the programming and specification standards to be used in the preparation of Secure IAS products

Task 7: Prepare Quality Assurance Plan

- Preparation of Preliminary Quality Assurance Plan
- Review of Quality Assurance Compliance

Task 8: Prepare Configuration Management Plan

- Preparation of Preliminary Configuration Management Plan according to MIL-STD-483
- Review products for compliance

Task 9: Develop Implementation Plan

- Prepare a Phase II Implementation Plan according to the methodology discussed in Section 7.5. This will include

discussion of the integrity of the implementation activities and the expected impact of the planned techniques on performance and verification.

### 7.6.2 Work Package 2 - Development

This work package is responsible for the prototype development and will consist of the following tasks:

Task 1: Complete Preliminary Design Specifications

- Update Preliminary Design Specifications completed in Feasibility Study as the result of comments received from reviews, etc.
- Publish B5 specifications in MIL-STD-490 format

Task 2: Prepare Detailed Prototype Specifications

- Develop detailed prototype specifications for the functions to be supported by the prototype
- Publish C5 specification in MIL-STD-490 format

Task 3: Prepare Formal Specifications of Security Related Software

- Update the formal specifications
- Prepare formal specifications of the trusted software
- Prepare lower level formal specifications

Task 4: Code, Debug, Unit Test Secure IAS

- Code, debug, and unit test the Secure IAS prototype

### 7.6.3 Work Package 3 - Testing

This work package covers the integration and testing of the prototype. This includes functional testing and performance testing of the prototype.

Task 1: Update Preliminary Design Specification Requirements Allocation

- Allocate all Preliminary Design Specification requirements to the various levels of testing
- Update allocation based on inputs from Design/Implementation work package
- Publish Test Evaluation Matrices (TEM)

Task 2: Prepare Test Plan

- Develop test plan for prototype testing
- Publish test plan

Task 3: Prepare Test Procedures

- Prepare test procedures for prototype testing
- Develop/adapt test tools
- Publish test procedures

Task 4: Conduct Prototype Testing

- Perform integration testing
- Document problems encountered
- Revise TEM as required
- Conduct functional test
- Perform performance tests
- Publish results

# 8. CONCLUSIONS AND RECOMMENDATIONS

## 8.1 Conclusions

The Secure IAS design is based upon security kernel technology. The concept of the reference monitor, the heart of the security kernel technology, requires that all accesses be mediated in order to preserve the secure state of the system. The model as defined in Appendix A provides a formal (abstract) representation of the information-handling requirements found in DOD directives and regulations in terms of abstract entities that access information and the entities that contain information. As such, the model formalizes only those requirements dealing with information access. Section 2 provides a set of computer security requirements which define the necessary protection capabilities in addition to those formalized by the model. The security definition provides the statement of the problem by which Secure IAS can be designed.

From these requirements, Secure IAS was designed to consist of three functional components: the Security Kernel, trusted software, and IAS Emulator. The Security Kernel interfaces directly with the IAS Emulator, trusted software, and PDP-11/70 computers and enforces DOD security policy in accordance with the Security model. The Security Kernel supports task, memory, file, I/O, and trap management.

The rest of the Secure IAS software is encapsulated within user processes, running under the control of the Security Kernel and making calls to it. In addition to the Security Kernel there are other security-related functions that are not part of the Security Kernel, but must be verified to perform these functions correctly. This trusted software has the responsibility for handling available terminals, logging-on and authenticating users, and starting up the appropriate software. It is protected by the Kernel from modification during execution.

The IAS Emulator simulates the standard IAS system directives by mapping each of the system directives into Security Kernel calls. It is this interface at which compatibility is sought.

Other design parameters of Secure IAS included performance and compatibility. Compatibility with current IAS will be affected because the incorporation of the security-related software will result in a great deal of internal reorganization and modification. Further, the elimination of executive privileged tasks will result in the replacement of over half of the IAS system software with Kernel call privileged software. Additionally, significant changes are required to the internals of the Files-11 supporting software.

At the user level, however, compatibility will be maintained with most interfaces; except the executive privileged task building capability is not provided. The user will perceive Secure IAS as a more restrictive system with standard IAS system facilities augmented with security related services such as login/logoff, entering protection data, and Kernel call privileges.

The issue of performance at this stage is very difficult to assess. One could expect performance for tasks with high input/output activity to decrease as much as 5-10% due to the fact that Secure IAS will be implemented in three hardware domains of the PDP-11/70 instead of two, as in current IAS. Further degradation may result due to the fact that Secure IAS will be implemented in a higher order language instead of assembly language as the current IAS is.

The performance of real-time tasks will be markedly affected since access to system resources will be through the Security Kernel and not directly as in current IAS. Since performance is such a key issue to the success of such an undertaking, a two phased development approach (section 7) has been described in which performance would be a key factor in the development of the prototype in the first phase.

The program development methodology presented is based upon generally practiced concepts and includes:

- establishment of firm development baselines,
- incremental development to minimize risk,
- use of unit develop folders to provide visibility and keeping documentation current,
- disciplined test planning and execution, and
- continuous product control.

These concepts are augmented by automated tools throughout to provide a higher quality product.

The verification methodology presented consists of a complementary set of known verification techniques in order to discover known kinds of errors efficiently. The proposed verification language is a synthesis of the GYPSY work of Don Good at the University of Texas, the EUCLID committee, SRI International developments, and TRW's own in-house developments. Tools to support top level verification will need to be developed. They are planned to be developed in the first phase to minimize any risk to the government.

## 8.2 Recommendations

The fact that the DARPA sponsored development of a secure version of Western Electric's UNIX*tm operating system called "Kernelized Secure Operating System (KSOS)," is scheduled to be completed in late 1979 raises the question of whether KSOS should replace or supplement the currently used DEC-developed RSX-11D operating system or the planned use of its interactive extension, IAS. The consequences of the availability of KSOS allow for many options for AF intelligence data processing. For example, AF intelligence can replace all current and planned systems with the KSOS system when it becomes available; time phase into KSOS; phase into KSOS on selected systems only; or remain with the RSX family of operating systems. Each choice has a different impact on capability and costs. Some installations may migrate to KSOS relatively easily; while for others it could be a major undertaking. While moving to a new system such as KSOS may open up new services to a limited spectrum of users, it often eliminates the possibility of old services.

The other alternative is that the AF intelligence community support the development and maintenance of Secure IAS. Similar options as stated above for KSOS exist for Secure IAS. Before any course of action is pursued, RADC needs to support several study efforts: They are:

- The development of the AF intelligence community's data processing user profile based upon the services of IAS. This involves not only

the identification of IAS services but also determining the frequency that these services are used by current AF intelligence applications. Particular emphasis should be placed on the real time services and executive privileged tasks.

- The identification of the development and maintenance costs for Secure IAS. This involves determining both the development and maintenance costs associated with Secure IAS.
- Identification of the migration considerations and costs of transition to Secure IAS and KSOS. This involves identifying the capability and service impacts, assessing the cost impact, and specifying any migration aids.

These RADC sponsored study efforts will provide adequate data to determine the AF intelligence community requirements. It will also determine if KSOS is a feasible option for the AF intelligence community. In particular, KSOS does not provide real-time facilities and hence migration to KSOS could result in major modifications to the current AF software. Further, both prototype systems developed at Mitre and UCLA showed much slower performance than current UNIX. There is some reason to believe that KSOS may have similar performance problems primarily because the major emphasis is placed on verifiable security rather than performance.

RADC should therefore monitor the status of the KSOS development very closely. This monitorship should not only attempt to assess the performance of the system, but also to determine any problems in the technical performance that could be applicable to a Secure IAS development. Particular attention should be paid to the choice of the programming language and the effectiveness of the formal verification methodology as demonstrated by qualification and security verification testing.

Based upon the above recommended studies and monitorship of KSOS, RADC will be in a position to develop a Secure IAS. The studies will provide data necessary to assess the cost impact of transitioning to a Secure IAS, and the monitorship will provide data as to the technical risk of developing a Secure IAS. Any technical problems can be identified and eliminated in the Secure IAS development.

# APPENDIX A
# AN ABSTRACT IMPLEMENTATION MODEL (AIM)

The abstract implementation model (AIM) interprets the security policy in security kernel implementation terms - kernel inputs, outputs, and states. The elements of AIM have relatively simple correspondences to V-functions and O-functions in the top level formal specifications. AIM represents discretionary security in terms of the need-to-know group types and access modes supported by IAS. The AIM also incorporates activity, tranquility, domain, and erasure policies.

## A.1 BASIC CONSTRUCTS OF AN AIM

In [Bell-la Padula 76], [Popek-Farber 77], and [Feiertag et all 77], different versions of abstract machine models of security kernels are presented. To varying degrees of detail, these models provide security constructs in terms of which certain security "axioms" are formulated. A security axiom expresses a relation between security attributes of information and information users. The conjunction of the axioms is intended to be a demonstrably sufficient condition which precludes any unauthorized information flow as specified by DOD security regulations. The extent to which this intended objective is achieved and demonstrated is the extent to which the model conforms, and is proved to conform, to DOD security policy.

Besides providing the primitive constructs on which the axioms can be based, an abstract implementation model should also serve as a useful general schema for the specification of a particular implementation of a security kernel. Therefore, an abstract implementation model must provide enough detail, in the form of machine constructs, to permit it to be a source reference for the design of practical implementations, while not obscuring or complicating unnecessarily the demonstration that the model conforms to DOD security policy. The AIM is such a model for security kernels (SK).

The AIM was developed by combining some of the concepts in [MITRE 77], [Bell-La Padula 76], [Popek-Farber 77], [Feiertag et all 77] and [Millen 76] with basic semantic concepts drawn from recent research in semantics of programming languages described in [Anderson et all 77].

In automata theory, it is customary to conceive of an abstract machine as a system, $M = (S,I,O,T,R,s^{(o)})$, where S is an abstract set of states, I is a set of inputs, O is a set of outputs, $s^{(o)} \in S$ is an initial state, $T: S \times I \rightarrow S$ is a state-transition function and $R: S \times I \rightarrow O$ is a result (or output) function. In some versions, the set O and the function R are omitted, since O can be regarded as a special part of the state to be retrieved later. In our version, we allow for the possibility that certain inputs may produce no externally visible output by introducing a distinguished NULL output element. Thus, R(s,i) = NULL means that input i produces no output when the machine is in state s. Furthermore, we limit our attention to finite-state machines, by which we mean machines in which S, I, and O are finite sets. This is justified by the finiteness of real implementations and it is useful restriction which eliminates many undecidability problems. This will be explained further in the following discussion.

In the theory of program schemata, programs with abstract uninterpreted operations are studied. This is a useful approach for analyzing general program properties for wide classes of programming languages. We adopt a similar approach in setting up a model of security kernels (SK). We define an abstract implementation model (AIM) to be an abstract machine schema. An AIM is a class of abstract machines, M, specified by giving general properties of I, T and R. Thus, it does not prescribe a particular set I and particular functions T and R. The particular sets and functions for IAS are specified in what we call the top level specification.

An AIM can be viewed as a class of machines, $(S,I,O,T,R,s^{(o)})$, as above, with the following properties.

(1) Each S is a finite set. A state $s \in S$, is a mapping from a fixed set, X, of state variables $\{x_0,x_1, \ldots,\}$ to an abstract finite set, V, of values. We write $s(x) = v$ to indicate that a variable $x \in X$ has a value $v \in V$, when the machine is in state s. We assume further that the domain of each s is finite; i.e. only a finite subset of the variables is involved.

(2) Each I is a finite set. An input, $i \in I$, is a triple (f,a,p), where f is a function name, a is a list of arguments and p is a process name. f is chosen from some abstract set of function names. p is chosen from some abstract set, P, of process names. The intended interpretation of (f,a,p) is a particular function call by a specific process to a SK, where f denotes the function, a is a list of arguments of the function to be used in its evaluation and p denotes the process making the call. The arguments in the list, a, will usually be variables and values. However, they may also be process names denoting other processes which are involved in the call. Note that we treat X, P and V as abstract sets. Hence, they need not be disjoint. Note also that in the intended interpretation a device is a special kind of process. In terms of security, a kernel input i is an access request, with p corresponding to the user, f corresponding to an access mode, and the argument list a corresponding to information objects. However, the functions f do not correspond to "pure" access modes, but rather, a kernel function performs the process management, memory management, etc. tasks involved in an access request.

(3) We introduce a partially ordered set of security levels and a function L which maps S x X U P into the set of security levels. Thus, L(s,x) and L(s,p) are the security levels assigned to x and p, respectively in state s. Similarly, we introduce a partially ordered set of integrity levels and a function J, which maps S x X U P into the set of integrity levels. The partial ordering in both sets is denoted by $\leq$. Thus, $L(s,x) \leq L(s,p)$ means that the security level of x is less than or equal to the security level of p and similarly for $J(s,x) \leq J(s,p)$. We assume that the set of security levels has a top element (TOP) and a bottom element and similarly for integrity. The value of the security level assigned to a process p is the security level at which process p assigns values to state variables or output variables, rather than the security clearance of the user on whose behalf the process p is acting. AIM assumes that the security level is assigned to p when p is initiated and is checked at that time to be less than or equal to the user's security clearance. Similarly, the integrity level J(s,p) of process p is the integrity level of the data output by p.

Remark. We shall explain later the correspondence of AIM constructs to the constructs in the top level specification of an implementation. In this correspondence, such attributes as security level and integrity level will be incorporated into the state of the top level specification, which may be conceived of as a specification of a particular machine in the AIM class. In setting up the AIM for SK, it seems preferable to keep the state structure as simple as possible and prescribe properties of a state by means of sets and functions external to the AIM machines themselves. The set of security levels and the function L is an example of such a set and function. This construct will be used to formulate the axioms of mandatory security policy. Other such constructs involving

elements external to the machines in the AIM class, but serving to define the class, are constructs to be used in formulating discretionary security policy and domain policy. We describe them in paragraphs 4 and 7 below. As with all abstract constructs in the AIM, we try to use names which suggest the intended interpretation.

(4) For the purpose of formulating discretionary security policy, we introduce a finite set, U, of user-identifiers and its power set, G, of user-groups. A subset $G_S \subset G$ is assigned to the IAS system. The equality relation (=) is defined on U and on G; i.e. we may determine if u = u' is true for any two user-identifiers in U and, similarly, if g = g' for any two user-groups. Next, we introduce two "owner" functions, $F_U$ and $F_G$, which for each state, assign user-identifiers and user-groups to certain state variables and to all processes. Thus, $F_U$: $Sx(X_1 UP) \rightarrow U$, where $X_1$ is a subset of X, and $F_U(s,x)$ is a user-identifier which specifies a "user" who owns $x \in X_1$ in state s. Not every $x \in X$ has a user owner. In SK, only those x corresponding to segments have owners. However, $F_U(s,p)$ is defined for every process $p \in P$; i.e. every process has a user owner. Similar considerations apply to $F_G$: $Sx(X_1 \cup P) \rightarrow G$ and group ownership. Finally, we introduce a set of discretionary access rights, D = {read, write, extend, delete}, and four access functions, $F_{UD}$: $SxX_1 \rightarrow$ Power-set-of D, $F_{GD}$: $SxX_1 \rightarrow$ Power-set-of D, $F_{SD}$: $SxX_1 \rightarrow$ Power-set-of D, and $F_{OD}$: $SxX_1 \rightarrow$ Power-set-of D. The value, $F_{UD}(s,x)$, determines the subset of access rights for the user owner, $F_U(s,x)$, of x in state s. Similarly, $F_{GD}(s,x)$ determines the group owner's access rights. $F_{SD}(s,x)$ determines the subset of access rights for the IAS system tasks. The fourth value $F_{OD}(s,x)$, is the subset of access rights accorded to other categories. Since user access rights depend on the state s, a change of access rights to a variable, x, resulting from some kernel call can be modeled as an input causing a state transition from s to a state s' such that $F_{UD}(s,x) \neq F_{UD}(s',x)$. Similar considerations apply to the other access functions and to the ownership functions. These functions represent the "need-to-know" types supported by IAS and the function values represent the access modes supported by IAS.

(5) To model privileged (or trusted) processes, we introduce a finite set of privileges, PR = {PR1, ..., PRn}, and a privilege function $F_{PR}$: P → Power-set-of (PR). Thus, $F_{PR}(p)$ is a subset of privileges which are accorded to each process $p \in P$, granting it exemption from

certain security axioms; i.e. the relations defined by certain axioms need not be applied to a process having certain privileges.

(6) In the operation of most computer systems, certain components of the state (e.g. disk segments) may undergo a change of status from "active" (i.e. created, allocated etc.) to "inactive" (i.e. destroyed, deallocated etc.). Likewise, processes become active (i.e. are spawned) and inactive (i.e. killed, dead, etc.). Again, this status information will be part of the state structure in the top level specification, but in an AIM it is more convenient to define status by means of a set and function independent of the state structure. Hence, we introduce an abstract set, A = {active, inactive}, and a function, $F_A$: $S \times (X_1 \cup P) \rightarrow A$, to represent activity status. Thus, $F_A(s,x)$ is the activity status of x in a state s. If an input causes the activity status of x to change, this is modeled by a transition of a new state, s', such that $F_A(s',x) \neq F_A(s,x)$. It can be seen that activity status is implicitly part of an AIM state. However, by using $F_A$ and A in the manner just explained, we can avoid giving any explicit description of the structure of the state to include this component. In the top level specification, a more explicit commitment to the placement of this component in the state structure is made. This, in turn, suggests a particular data structure for activity status in SK.

(7) As mentioned earlier, an AIM should include enough machine constructs to serve as a starting point for the design of SK. Most implementations will run on computers which provide hardware-protected <u>domains</u>. Domains provide hardware protection for those components of a state which contain system information relevant to the management of security information flow. To model domain policy we introduce an abstract totally ordered set of domains, DOM = {user, supervisor, kernel} and a function, $F_{DOM}$: $S \times (X_1 \cup P) \rightarrow DOM$. Thus, $F_{DOM}(s,p)$ is the domain of process p in state s. As with the other functions described above, the inclusion of s as an argument of $F_{DOM}$ allows us to model changes in the domain by effecting a state transition.

This completes the definition of the AIM. It is an abstract machine class endowed with the attributes described in (1) - (7).

## A.2 RELATIONS BETWEEN PROCESSES AND STATE VARIABLES

In an SK implementation, a person who wishes to be a user of the information in the system must do so by logging in on some terminal device. The log-in procedure, if successfully completed, causes a process to be initiated for the user at that terminal. From the system viewpoint, this process is

the user's surrogate. All interaction between the user and the information stored in the system is accomplished by interaction between the surrogate process and the rest of the system. Such interaction may require that the surrogate process initiate other processes which then perform appropriate accesses to the desired information. Therefore, in an AIM, relations between users and information are represented by relations between processes and state variables. DOD regulations which apply to a person who wishes to use classified information give rise to axioms expressed in terms of the processes which that person creates to interact with the state variables containing the desired information as values. To specify the permissible interactions, it suffices to consider two AIM relations between processes and variables. To suggest the intended interpretation of these relations, we call them the observe and modify relations. They are defined as follows:

Let $M = (S,I,O,T,R,s^{(o)})$ be an AIM machine. Consider each input $i = (f,a,p.) \in I$. It is convenient to define a corresponding output function, $R_i$: $S \to O$, in the obvious way by the equation, $R_i(s) = R(s,i)$. Let $x \in X$. We say that i depends on x in state s if there exists a state $s_1 \in S$ such that $s_1(x) \neq s(x)$, $s_1(y) = s(y)$ for all $y \neq x$ and $R_i(s_1) \neq R_i(s)$. We denote this dependency relation by Depend(i,x,s). Thus, Depend(i,x,s) is true if and only if the value of the output function $R_i$ can be changed by changing only the value of x. We remark that Depend(i,x,s) is decidable, since the domain, $S \times I$, of R is finite and S involves a specified finite subset of X.

Another kind of functional dependence is important in our modeling of the flow of information. Let x and y be two state variables. We say that y depends strictly on x in state s if there exists a state, $s_1$, and an input i such that $s_1(x) \neq s(x)$, $s_1(z) = s(z)$ for all $z \neq x$, and $s_1'(y) \neq s'(y)$, where $s_1' = T(s_1,i)$ and $s' = T(s,i)$. We denote this relation by Dep-str(y,x,s). Thus, Dep-str(y,x,s) holds if and only if a change in the value of x alone can affect how i changes the value of y. Again, this is a decidable relation for any AIM machine. Any such input i which causes Dep-str(y,x,s) to hold will be called a (y,x,s)-dependency input. If i is a (y,x,s)-dependency input for some s, then we call i a (y,x)-dependency input.

In an implementation of SK, an input, i, is realized as a kernel call. Each call is issued by a currently executing process, p. This situation is modeled in our AIM by representing i as a triple, (f,a,p), where p denotes the current process. For i = (f,a,p) we define proc(i) = p. Thus, proc(i) is the process which issues the call. For each $p \in P$, we define the set, $I_p = \{i \in I: \text{ proc}(i) = p\}$. $I_p$ is just the set of all inputs i having p as the third component. In SK, it is the set of all calls which p can issue. Clearly, $I = \bigcup_{p \in P} I_p$.

Definition 1. Let $p \in P$ and $x \in X$. We say that p can observe x in state s if there exists an input $i \in I_p$ such that Depend(i,x,s) holds. We denote this relation by Observe(p,x,s).

Thus, in the intended interpretation, Observe(p,x,s) holds if and only if p can make a kernel call i which can produce an output which depends on x. Observing implies the transmission to the external world of information based on x. For the reasons remarked above, Observe(p,x,s) is decidable for any AIM machine.

Definition 2. Let $p \in P$ and $x \in X$. We say that p can modify x in state s (written Modify (p,x,s)) if there exists an $i \in I_p$ such that $s'(x) \neq s(x)$, where s' = T(s,i).

Thus, in the intended interpretation, Modify(p,x,s) holds if and only if p can make a kernel call which changes the value of x.

## A.3 MANDATORY SECURITY AXIOMS

Using the Observe and Modify relations, we can now formulate three axioms which are meant to embody DOD mandatory security policy. As with the Observe and Modify relations, the axioms are framed with respect to an arbitrary but fixed AIM machine.

Notation. If $L(u) \leq L(v)$ does not hold, we write $L(u) \not\leq L(v)$. This means that $L(u) > L(v)$ or L(u) and L(v) are incomparable.

$A_1$ (Simple Security Axiom). For all $s \in S$, $x \in X$ and $p \in P$, if $L(s,x) \not\leq L(s,p)$ and $PR_1 \notin F_{PR}(p)$, then Observe(p,x,s) does not hold.

There are several logically equivalent formulations of $A_1$. One states that if $PR_1 \notin F_{PR}(p)$, then Observe(p,x,s) implies $L(s,x) \leq L(s,p)$; i.e. if p does not have privilege $PR_1$, then the security level of any variable which p can observe must be less than or equal to the security level of p. We prefer the statement $A_1$, since it is directly translatable into exception conditions in the top level specification of V-functions and OV-functions corresponding to abstract inputs. Let i = (f,a,p) be any input in $I_p$. As already explained, in an SK implementation, f(a) would be a particular instance of a call of a kernel function f and p would be the current process making the call in current state s. Suppose that "normal" execution of f(a) would produce an output dependent on the value of some segment x. If $L(s,x) \not\leq L(s,p)$ and p does not have a special privilege, then an exception is made to the normal execution, that is, a NO-ACCESS output which does not depend on x is produced instead of the normal output. Since this applies to any $i \in I_p$, the process p cannot observe this x in state s. Normal execution presupposes that there is some process, $p_1$ say, which can issue the call f(a) and cause the x-dependent output to be produced. In AIM terms, the input $(f,a,p_1)$ depends on x and $p_1$ observes x in some state, s. By axiom $A_1$, either $p_1$ is privileged or $L(s,x) \leq L(s,p_1)$. The effect of $A_1$ on an AIM machine is to impose a constraint on the output functions $R_i$ and, therefore, on the output function R.

In the top level specification, this situation can be described by defining f to be a V-function or OV-function which returns a result provided that an exception condition does not hold. The exception condition is derivable immediately from $A_1$, namely, $L(s,x) \not\leq L(s,p)$ and $PR_1 \notin F_{PR}(b)$, where p is the current process. Here, x need not be an arbitrary state variable, since the parts of an actual state which can possibly be observed by a call f(a) can be determined by examining the description of f and the formal parameters corresponding to the argument list a.

Because the security level L(s,p) of p in state s is the security level at which p is allowed to output information, the *-property relation of other security models decomposes into two axioms, one concerned with p modifying a state variable and one concerned with information flow between two state variables.

$A_2$ (Security *-Property Axiom). For all $s \in S$, $x \in X$ and $p \in P$, if $L(s,p) \not\leq L(s,x)$ and $PR_2 \notin F_{PR}(p)$, then Modify(p,x,s) does not hold.

Equivalently if $PR_2 \notin F_{PR}(p)$, then Modify(p,x,s) implies $L(s,p) \leq L(s,x)$; i.e. if p does not have privilege $PR_2$, then the security level of p must be less than or equal to the security level of any variable which p can modify. Again, the statement in $A_2$ is directly translatable into an exception condition in the top level specification of O-functions and OV-functions corresponding to inputs i = (f,a,p) which change the state. Suppose that normal execution of f(a) would have the effect of changing the value of state variable x for some state s. (This can be determined by examining the list of effects in the definition of the O-function corresponding to f.) If the current process, p, is such that $L(s,p) \not\leq L(s,x)$ and p does not have a special privilege, then an exception is made to the normal execution, that is, f(a) produces no change of state. Since this applies to any $i \in I_p$ which could change x under normal execution, the process p cannot modify this x. Normal execution presupposes that there exists some process, $p_1$ say, which can modify x by issuing the call f(a) in some state s. In AIM terms, input $(f,a,p_1)$ causes the value of x to change when the machine is in state s. By axiom $A_2$, either $p_1$ is privileged or $L(s,p_1) \leq L(s,x)$. The effect of $A_2$ on an AIM machine is to impose a constraint on the state-transition function T.

$A_3$ (Information Flow Security Axiom). Let x,y be state variables. If $L(s,x) \not\leq L(s,y)$ and $p_1$ is such that $PR_3 \notin F_{PR}(p_1)$, then no input of the form $(f,a,p_1)$ is a (y,x,s)-dependency input.

Equivalently, if there is an input i = (f,a,p) which can cause a change in the value of y which depends on the value of x (i.e. if i is a (y,x,s)-dependency input), then p has privilege $PR_3$ or $L(s,x) \leq L(s,y)$. Axiom $A_3$ imposes a further constraint on the transition function of an AIM machine. In the top level specification, the exception $L(s,x) \not\leq L(s,y)$ must be included in the definition of every O-function or OV-function f, corresponding to an input (f,a,p) which can change the value of y in a manner which depends on the value of x. This prevents information from "flowing down" security levels. The list of effects in the definition of f allows us to determine the pairs (x,y) for which this may occur.

## A.4 MANDATORY INTEGRITY POLICY

Since integrity can be viewed as a "dual" of security, the three integrity axioms can be derived from the three security axioms by substituting J(s,P) and J(s,x) for L(s,p) and L(s,x) respectively and by inverting the $\not\leq$ relation.

$A_4$ (Integrity *-Property Axiom). For all $s \in S$, $x \in X$ and $p \in P$, if $J(s,p) \not\leq J(s,x)$ and $PR_4 \notin F_{PR}(p)$, then Observe(p,x,s) does not hold.

$A_5$ (Simple Integrity Axiom). For all $s \in S$, $x \in X$ and $p \in P$, if $J(s,x) \not\leq J(s,p)$ and $PR_5 \notin F_{PR}(p)$, then Modify(p,x,s) does not hold.

$A_6$ (Information Flow Integrity Axiom). Let x,y be state variables. Let $s \in S$. If $J(s,y) \not\leq J(s,x)$ and $p_1$ is such that $PR_6 \notin F_{PR}(p_1)$, then no input of the form $(f,a,p_1)$ is a (y,x,s)-dependency input.

## A.5 DISCRETIONARY SECURITY POLICY

$A_7$ (Discretionary Observe Axiom). Let $s \in S$, $x \in X$ and $p \in P$. Suppose that all of the following five conditions hold:

(a) $F_U(s,p) \neq F_U(s,x)$ or read $\notin F_{UD}(s,x)$;

(b) $F_G(s,p) \neq F_G(s,x)$ or read $\notin F_{GD}(s,x)$;

(c) read $\notin F_{OD}(s,p)$ or $F_U(s,p) = F_U(s,x)$ or $F_G(s,p) \in G_S$ or $F_G(s,p) = F_G(s,x)$;

(d) read $\notin F_{SD}(s,p)$ or $F_G(s,p) \notin G_S$;

(e) $PR_7 \notin F_{PR}(p)$.

The Observe(p,x,s) does not hold.

A logically equivalent statement of the discretionary observe relation is obtained by taking the contrapositive of $A_7$. It says that a process p can observe a variable x in state s only if at least one of the following conditions is true in s: (not a) x and p have the same user-owner and the user-owner has the read access right to x; (not b) x and p have the same group-owner which has read access to x; (not c) p has some other user-owner and

group-owner and other users (meeting mandatory security) have read access to x; (not d) p is in some system group which has read access to x; (not e) p has special privilege $PR_7$ which exempts it from (not a), (not b), (not c) and (not d). In the top-level specification, the conjunction of (a), (b), (c), (d) and (e) constitutes an exception which must be included in the definition of any O-function, f, for which a call, f(a), can cause the issuing process to observe a variable x. As mentioned earlier, for any O-function (or OV-function), f, the candidate variables x can be determined from the formal argument list corresponding to a and the body of the definition of f. If the definition of f contains references to other V-functions, then determination of all possible x's may require tracing through finite chains of such syntactic references starting with the specified output result. Each x in the set of variables determined by this syntactic search is a candidate for the relation Observe(p,x,s). However, by Definition 1, whether Observe(p,x,s) holds depends on whether there are two states which differ only in the value of x and for which some call, f(a), produces different outputs. In principle, this could be determined by exhaustive trials of all such pairs of states and all possible calls, f(a), since these are finite in number. Of course, in practice, whether Observe(p,x,s) holds can usually be decided by an analysis of the definition of the O-function f.

It is understood that the implementation will check exception conditions whenever a kernel call is received. Thus, the exception conditon must hold for all time (i.e. all possible states) to prevent observation of x.

$A_8$ (Discretionary Modify Axiom). Let $s \in S$, $x \in X$ and $p \in P$. Suppose that all of the five following conditions hold:

(a) $F_U(s,p) \neq F_U(s,x)$ or $F_{UD}(s,x) \not\subset \{write, delete, extend\}$;

(b) $F_G(s,p) \neq F_G(s,x)$ or $F_{GD}(s,x) \not\subset \{write, delete, extend\}$;

(c) $F_{OD}(s,x) \not\subset \{write, delete, extend\}$ or $F_U(s,p) = F_U(s,x)$ or $F_G(s,p) \in G_S$ or $F_G(s,p) = F_G(s,x)$;

(d) $F_{SD}(s,p) \not\subset \{write, delete, extend\}$ or $F_G(s,p) \notin G_S$;

(e) $PR_8 \notin F_{PR}(p)$.

Then Modify(p,x,s) does not hold.

As with $A_7$, we formulate the contrapositive of $A_8$. It says that p can modify x only if at least one of the following conditions holds in state s:

(not a) p and x have the same user-owner, who has write, delete or extend access to x;

(not b) p and x have the same group-owner, who has write, delete or extend access to x;

(not c) p has a different user-owner and group-owner than x and other users (meeting mandatory security) have write, delete or extend access to x;

(not d) p is in some system group which has write, delete or extend access to x;

(not e) p has special privilege $PR_8$.

The implications of $A_8$ for the top level specification are analogous to those for $A_7$.

## A.6 ACTIVITY, TRANQUILITY AND ERASURE

The activity and erasure axioms $A_9$ and $A_{11}$ represent the security requirement that information no longer needed within the system (e.g., the residue in a segment after completion of processing) should not be recoverable by a system action. The tranquility axiom $A_{10}$ represents a requirement that security levels of active objects not be changed.

$A_9$ (The Activity Axiom). Let $s \in S$, $x \in X$ and $p \in P$. Suppose that the condition $F_A(s,x)$ = inactive and $PR_9 \notin F_{PR}(p)$ holds. Then Observe(p,x,s) does not hold and for all $y \in X$, Dep-str(y,x,s) does not hold.

$A_9$ prescribes another exception condition which must be included in all O-function, OV-function and V-function definitions which have a potential to observe some variable x.

The security level function, L, depends on s. This permits security levels to be changed by certain inputs. We restrict such changes by imposing the following axiom.

$A_{10}$ (Tranquility Axiom). Let $x \in X \cup P$. Suppose there exists $s \in S$ and $i \in I$ such that $L(s,x) \neq L(s',x)$ where $s' = T(s,i)$. Then $F_A(s,x) =$ inactive or $F_A(s',x) =$ inactive.

To model erasure, we introduce a special value, called UNDEFINED. The interpretation of $s(x) =$ UNDEFINED is that the value of x in state s is not specified explicitly in the top level specification or in an SK implementation. Likewise, $s(x) \neq$ UNDEFINED means that the value of x in state s is specified explicitly in the top level specification and in an SK implementation.

$A_{11}$ (Erasure Axiom). Let $x \in X$. Suppose there exists $s \in S$ and $i \in I$ such that $F_A(s,x) =$ inactive and $F_A(s',x) =$ active, where $s' = T(S,i)$. Then $s'(x) \neq$ UNDEFINED. Suppose there exists $s \in S$ and $i \in I$ such that $F_A(s,x) =$ active and $F_A(s',x) =$ inactive, where $s' = T(s,i)$. Then $s'(x) =$ UNDEFINED. Finally, for all $x \in X$ such that $F_A(s^{(o)},x) =$ inactive, $s^{(o)}(x) =$ UNDEFINED.

# APPENDIX B
# GLOSSARY OF ACRONYMS

| | |
|---|---|
| ACP | Ancillary Control Processor |
| AF | Air Force |
| AF/ESD | Air Force/Electronic Systems Division |
| AIM | Abstract Implementation Model |
| ARPANET | Advanced Research Projects Agency Network |
| AST | Asynchronous System Trap |
| ATL | Active Task List |
| BAD | Bad Block Locator Utility |
| CDA | Core Dump Analyzer |
| CDR | Critical Design Review |
| CLI | Command Language Interpreter |
| CMP | File Compare Utility |
| CPCI | Computer Product Configuration Item |
| DAA | Designated Approving Authority |
| DARPA | Defense Advanced Research Projects Agency |
| DEC | Digital Equipment Corporation |
| DMP | Dump Utility |
| DOD | Department of Defense |
| DSC | Disk Save and Compress Utility |
| DSW | Directive Status Word |
| EDI | Text Editor Utility |
| EUCLID | Programming Language Chosen for Implementation of Secure IAS |
| FCA | Functional Configuration Audit |
| FCS | File Control Services |
| FLX | File Transform Program |
| F11ACP | Files-11 Ancillary Control Processor |
| GYPSY | Candidate Programming Language for Implementation of Secure IAS |
| HOL | High Order Language |
| IAS | Interactive Applications System |
| IASCOM | Time-sharing Executive Data Base |
| IMP | Interface Message Processor |
| IRQ | I/O Request Queue |

| | |
|---|---|
| ISI | University of Southern California Information Sciences Institute |
| ITC | Inter-task Communication |
| KSOS | Kernelized Secure Operating System |
| LBR | Librarian Utility |
| LUN | Logical Unit Number |
| LUT | Logical Unit Table |
| MCR | Monitor Control Routine |
| MFD | Master File Directory |
| MTAACP | Magtape Ancillary Control Processor |
| MULTICS | Operating System for HIS/GE 600 Series Computers |
| NODAL | Automated Test Tool to Analyze Structure of Each HOL Routine |
| ODT | On-line Debugging Technique |
| OFUN | O-function |
| OSTEST | Automated Test Tool Which Provides a Measure of Test Effectiveness |
| OVFUN | OV-function |
| PAR | Page Address Register |
| PASCAL | Candidate Programming Language for Implementation of Secure IAS |
| PAT | Object Module Patch Utility |
| PCA | Physical Configuration Audit |
| PDP-11 | Family of Computers Manufactured by DEC |
| PDR | Preliminary Design Review |
| PDS | Program Development System |
| PIP | Peripheral Interchange Program |
| PRESRV | Preservation Utility |
| QIO | Queue I/O |
| RSX-11O | Comprehensive Multi-programming System |
| SCI | System Control Interface |
| SCOM | System Communication Area (Kernel Communication Area) |
| SGA | System Global Area |
| SLP | Source Source Language Input Program |
| SPECIAL | Specification and Assertion Language |
| SP-EUCLID | Specification Language Developed from the EUCLID Programming Language |
| SQZ | Squeeze Utility |

| | |
|---|---|
| SST | Synchronous System Trap |
| TCB | Transmission Control Block |
| TCP | Time-sharing Control Primitive, Transmission Control Program |
| TCS | Time-sharing Control Services |
| TEM | Test Evaluation Matrix |
| TKB | Task Builder |
| TPNS | IBM's Teleprocessing Network Simulator |
| TRR | Test Readiness Review |
| UDF | Unit Development Folder |
| UFD | User File Directory |
| UIC | User Identification Code |
| UNIX | Trademark of the Bell System for Their Proprietary Product - UNIX Operating System for DEC PDP-11 |
| UPF | User Profile File |
| VFUN | V-function |
| VFY | Verification Utility |
| WPM | Work Package Manager |
| XIVUS | Interactive Verification System Developed by ISI |
| ZAP | ZAP Utility |

## APPENDIX C

## REFERENCES


Ames, S.R., Jr., and J.K. Millen, Interface Verification for a Security Kernel (DRAFT), Mitre Corp., Bedford, Mass. (September 1, 1977).

Anderson, E.R., F.C. Belz, and E.K. Blum, Issues in the Formal Specification of Programming Languages, Proc. IFIP WG 2.2 Working Conference, St. Andrews (August 1977).

Bell, D.E., Secure Computer Systems: A Refinement of the Mathematical Model, Mitre Technical Report MTR-2547, Vol. III (28 December 1973).

Bell, D.E., and L.J. La Padula, Secure Computer Systems: Mathematical Foundations, Mitre Technical Report MTR-2547, Vol. I (1 March 1973).

Bell, D.E., and L.J. La Padula, Secure Computer Systems: Unified Exposition and Multics Interpretation, ESD-TR-75-306, Mitre Corp. (March 1976).

Biba, K.J., Integrity Considerations for Secure Computer Systems, Mitre Technical Report MTR-3153, Rev. 1 (April 1977).

Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1976).

Cerf, V., Specification of Internet Transmission Control Program, Version 3, December 1977 (Available from DARPA).

Feiertag, R.J., K.N. Levitt, and L. Robinson, Proving Multilevel Security of a System Design, Operating Systems Review 11, 5 (Special Issue 16-18 November 1977) 57-65.

Floyd, R.W., Assigning Meanings to Programs, in Proc. Symposium Applied Math. 19, American Matehmatical Society, Providence, R.I. (1967) 19-32.

Gerhart, S.L. and L. Yelowitz, Observations of Fallibility in Applications of Modern Programming Methodologies, IEEE Trans. Software Engineering SE-2 (3 September 1976) 195-207.

Goodenough, J.B. and S.L. Gerhart, Toward a Theory of Test Data Selection, IEEE Trans. Software Engineering SE-1 (2 June 1975) 156-173.

Hantler, S.L. and J.C. King, An Introduction to Proving the Correctness of Programs, Computing Surveys 8 (3 September 1976) 331-353.

Hoare, C.A.R., An Axiomatic Basis for Computer Programming, CACM 12 (10 October 1969) 576-583.

Hoare, C.A.R., Proof of Correctness of Data Representations, Acta Information 1 (1972) 271-281.

Howard, J.H., Proving Monitors, CACM 19 (5 May 1976) 273-279.

Kemmerer, R.A., Verification of the UCLA Security Kernel: Abstract Model, Mapping, Theorem Generation, and Proof, UCLA Computer Science Department Dissertation Prospectus (1978).

Kernelized Secure Operating System Verification Plan, TRW Report prepared for Defense Advanced Research Projects Agency (April 1978).

La Padula, L.J., and D.E. Bell, Secure Computer Systems: A Mathematical Model, Mitre Technical Report MTR-2547, Vol. II (31 May 1973).

Lampson, B.W., J.J. Horning, R.L. London, J.G. Mitchell, and G.L. Popek, Report on the Programming Language EUCLID, SIGPLAN Notices 12 (2 February 1977) 1-79.

Millen, J.K., Security Kernel Validation in Practice, CACM 19 (5 May 1976) 243-250.

Millen, J.K., Causal System Security (DRAFT), Mitre Corp., Bedford, Mass. (September 6, 1977a).

Millen, J.K., Example of a Formal Flow Violation, Mitre Corp., Bedford, Mass. (1977b).

Millen, J.K., Formal Specifications for Security, in Trends and Applications 1977: Computer Security and Integrity, IEEE Computer Society (1977c).

MITRE Corp., Draft, B-Level Specification of Secure UNIX (October 1977).

Neumann, P.G., R.J. Feiertag, K.N. Levitt, and L. Robinson, Software Development and Proofs of Multi-Level Security, Proc. 2nd Int. Conf. on Software Engineering (1976).

Owicki, S., Axiomatic Proof Techniques for Parallel Programs, Cornell University Computer Science Department TR 75-251, Ithaca, N.Y. (July 1975).

Owicki, S. and D. Gries, Verifying Properties of Parallel Programs: An Axiomatic Approach, CACM 19, 5 (May 1976) 279-285.

Parnas, D.L., A Technique for Software Module Specification with Examples, CACM 15 (5 May 1972) 330-336.

Popek, G.J., and D.A. Farber, A Model for Verification of Data Security in Operating Systems, UCLA Report (1977).

Price, W.R., Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems, Ph.D. Dissertation, Carnegie-Mellon University (June 1973).

Robinson, L., and K.N. Levitt, Proof Techniques for Hierarchically Structured Programs, CACM 20 (4 April 1977) 271-283.

Robinson, L., and O. Roubine, SPECIAL - A Specification and Assertion Language, Technical Report CSL-46, SRI International, Menlo Park (January 1977).

Wulf, W.A., R.L. London and M. Shaw, An Introduction to the Construction and Verification of Alphard Programs, IEEE Software Engineering SE-2 (4 December 1976) 253-265.

☆U.S. GOVERNMENT PRINTING OFFICE: 1979-614-023/142