Improving Security and Performance for Capability Systems

Paul Ashley Karger

Wolfson College

Dissertation submitted for the degree of Doctor of Philosophy
in the University of Cambridge, 30 March 1988.

Copyright ©1988 by Paul Ashley Karger.

Contents

Acknowledgements **12**

Summary **14**

I Background **15**

1 Introduction **17**
1.1 History of the Project 17
1.2 What is SCAP? 18
1.3 Overview of Experimental Environment 18
1.4 Plan of the Dissertation 19

2 The Need for Security **21**
2.1 Classes of Vulnerabilities 21
2.1.1 Browsing 21
2.1.2 Unauthorized Acts of Authorized Individuals 22
2.1.3 Direct Penetration 22
2.1.4 Trap-Door and Trojan-Horse Attacks 23
2.2 Development of Secure Systems 25
2.2.1 Access-Control-List Systems 26
2.2.2 Capability systems 26
2.2.3 Security Kernels 27
2.3 Assumptions 28

3 Models of Security **31**
3.1 Preventing Information Disclosure 31
3.1.1 Discretionary Access Controls 32
3.1.2 Non-Discretionary Access Controls 32
3.1.2.1 Elements of the Lattice Model 33
3.1.2.2 Defeating Trojan Horses 34
3.1.3 Retrofitting Non-discretionary Security 34
3.1.3.1 Incremental Addition of Features 35
3.1.3.2 Kernel/Emulator Approach 35
3.2 Preventing Tampering and Sabotage 36
3.2.1 Biba Integrity Model 37
3.2.2 Lipner Commercial Integrity Model 37

3.2.3 Clark and Wilson Commercial Integrity Model 37
3.3 Preventing Denial of Service 39

4 Principles of Capability Systems **41**
4.1 What is a Capability? 41
4.2 Capability Storage 42
4.3 Need for Protected Subsystems 43
4.4 Type Managers and Sealing 45

II SCAP Architecture **47**

5 Overview of the SCAP Architecture **49**
5.1 SCAP Processor Architecture 49
5.2 SCAP Operating System 50

6 SCAP Domain Model **53**
6.1 Scheduling Entities 53
6.1.1 Jobs 53
6.1.2 Processes 53
6.2 Address Space Entities 54
6.2.1 Protected Subsystems 54
6.2.2 Domains 55
6.3 A Simple Example 55
6.4 Cross-Domain Calls 57
6.5 Comparison with Reed's Scheduler 57
6.6 Non-Discretionary Controls for Processes and Domains . . . 58
6.7 Creation and Initialization 59
6.7.1 Jobs 59
6.7.2 Processes 61
6.7.3 Protected Subsystems 61
6.7.4 Domains 62
6.8 Message Passing and Procedure Calls 63

III Improving Security **65**

7 Solving the Confinement Problem **67**
7.1 Attempts with Traditional Capabilities 68
7.1.1 HYDRA 68
7.1.2 PSOS 69
7.2 The Secure Capability Architecture 69
7.3 Non-Discretionary Security 70
7.4 Comparison With Other Systems 72
7.4.1 System/38 72
7.4.2 SWARD 73
7.4.3 Monash Password-Capability System 73

7.4.4 Honeywell Secure Ada Target (SAT) 74
7.4.5 KeyKOS 75
7.4.6 Flex 75
7.5 Kain and Landwehr's Taxonomy 76

8 Traceability of Access Problems **79**
8.1 Asymmetric Views of Security 79
8.2 Discretionary Security with SCAP 80

9 Discretionary Trojan Horses **83**
9.1 Directory Management 83
9.2 Name-Checking Protected Subsystem 85
9.3 Name Translation in Batch Jobs 88
9.3.1 Special Directory Trees 88
9.3.2 Pre-Compiled Batch Jobs 88
9.3.3 Additional Approaches 89
9.3.3.1 Wildcard Authorization 89
9.3.3.2 Post-Authorization 90
9.4 Access-Control-List Systems 90
9.5 Alternate Strategies 90
9.5.1 Strict Need-to-Know Policy 91
9.5.2 Enhanced Linker 91
9.5.3 Flex Cartouches 91
9.6 Limitations of the Technique 92

10 Implementing Commercial Integrity **95**
10.1 Implementation 95
10.1.1 Certification Difficulties 95
10.1.2 Crucial Role of the Audit Trail 96
10.1.3 Implementing with Secure Capabilities 97
10.1.4 Handling Groups of Users 99
10.1.5 Security Policy, Auditing, and Recovery Management 100
10.2 Related Work 100
10.2.1 AAS 101
10.2.2 RSS 101
10.2.3 Cascaded Network Connections 101
10.2.4 Program-Integrity Policy 102
10.2.5 Secure Committees 103
10.2.6 Assured Pipelines 103
10.2.7 Enforcing Clark and Wilson with Integrity Categories 103
10.3 Trojan-Horse Problems 104
10.4 Performance Problems 105
10.5 Retrospective 105

11 Improved Revocation Algorithms **107**
11.1 Need for Revocation 107
11.2 Revocation Difficulties 109
11.2.1 Multics Revocation with Back Pointers 109
11.2.2 Redell's Revocation with Indirection 111
11.3 Revocation with Eventcounts 113
11.4 Revocation by Chaining 115

12 Secure Garbage Collection **117**
12.1 Source of the Problems 117
12.2 Solutions that Do Not Work 118
12.3 Quota Management 118
12.3.1 Multics Quota Problem 119
12.3.2 Quota Cells 120
12.4 Payment Systems 120

IV Improving Performance **123**

13 Performance Overview **125**
13.1 Performance Problems 125
13.2 Applying RISC Technology 126

14 Programming Generality **129**
14.1 Costs of Programming Generality 129
14.1.1 Argument Validation 129
14.1.2 Procedure Calls in the Intel 432 131
14.2 Costs of Capability Refinements 132
14.2.1 Uses of Refinements 133
14.2.2 Restricting Refinements 134

15 Translation Buffers **137**
15.1 Hardware-Visible Segmentation 137
15.1.1 Segment Descriptors in the Translation Buffer 138
15.1.2 Segment Descriptors in Software Only 138
15.2 Context Switching 139
15.2.1 Flushing 139
15.2.2 Translation-Buffer Swapping 140
15.2.3 Address Space Numbers (ASNs) 140
15.2.4 Unique-ID Addressing 141
15.3 TB Fill in Software or Hardware 142
15.4 Shared-Memory Multiprocessors 143
15.4.1 Flush with Interprocessor Interrupts 143
15.4.2 Snoopy Translation Buffers 144

16 Hashed Page Tables **147**
16.1 Implementing a Hashed Page Table 147
16.2 Probability of Collisions 149
16.3 Hash Function Costs 149
16.4 Number of Memory References 150
16.5 Revocation by Chaining 152
16.6 Hashing Experiment 153
16.6.1 Structure of the Hashed Page Table 153
16.6.2 Hashing Algorithm 154
16.6.3 Address-Space-Number Management 156
16.6.4 Invalidating PTEs in the Hash Table 157
16.6.5 Hashing Results 158

17 Cross-Domain Call Optimization **161**
17.1 Performance of Cross-Domain Calls 161
17.2 Multiple Register Sets 162
17.3 Argument Passing 163
17.4 Categories of Trust 164
17.5 Register Optimization Based on Trust 165
17.6 Capability-Argument Optimization 166
17.6.1 Avoiding Clearing 168
17.6.2 Avoiding Probing 169
17.7 Implementing the Optimizations 169
17.7.1 Trusted Linker 169
17.7.2 Microcoded Cross-Domain Call 170
17.7.3 RISC Cross-Domain Call 175
17.8 Minimizing Argument Clearing 177
17.9 Optimizing With Long Returns 180
17.10 Benefits 181

18 Cross-Domain-Call Performance Experiments **183**
18.1 Experimental Results 184
18.2 Comments on the Performance Results 187

19 Real-Time Issues **189**
19.1 Idealized Interrupt Handling 189
19.2 SCAP Hardware Interrupt Handling 190
19.3 Cross-Domain Calls at Elevated IPL 190
19.4 Software Interrupts and ASTs 191
19.5 Operation of the Scheduler 192
19.5.1 Scheduling Due to Interrupts 192
19.5.2 Scheduling Due to Explicit Calls 193
19.6 Summary of Stack Usage 193
19.7 Real-Time Processing 194

20 Conclusions **195**
20.1 Major Research Results 195
20.2 Problem Resolution 196
20.3 The Next Step 196

References **197**

Appendices

A Computer Security Evaluation Criteria **217**

B Tutorial on Paging **219**
B.1 Atlas 219
B.2 Multics 220
B.3 VAX 220
B.4 Additional Levels of Page Tables 221

C Translation Buffer Associativity **223**
C.1 Fully Associative 223
C.2 Direct Mapped 224
C.3 Set Associative 226

D VAX Processor Architecture **229**
D.1 Data Types 229
D.2 Registers 229
D.2.1 Programmer Visible Registers 229
D.2.2 Internal-Processor Registers 232
D.3 Instruction Formats 233
D.4 Memory Management 233
D.5 Protection 233
D.6 Interrupts and Exceptions 234

E Microarchitecture of the VAX-11/730 **237**
E.1 System Overview 237
E.2 CPU Description 239
E.2.1 Memory Controller (MCT) 239
E.2.2 Writable-Control Store (WCS) 239
E.2.3 CPU Data Path (DAP) 241
E.2.4 Micro Instruction Set 241
E.3 Microprogram Organization 242
E.4 Microprogramming Tools 243

F Interrupt Handling in Capability Systems **245**
F.1 CAP 245
F.2 Intel 432 246
F.3 IBM System/38 246
F.4 Honeywell DPS 88 246
F.5 Plessey System 250 247

G Possible Kernel Design **249**
G.1 The Major Type Managers 249
G.2 Performance Considerations 253

H SCAP Software Compatibility **255**
H.1 General Issues 255
H.2 Replacing Setuid 256
H.2.1 Setuid Protected Subsystems 256
H.2.2 SCAP Protected Subsystems in UNIX 257
H.3 A Virtual Machine Monitor for SCAP 257

I Annotated Code Sequences **259**
I.1 Call with JSB 259
I.2 Cross-Domain Call with SVPCTX 260
I.3 Microcode Invoker 263
I.4 Cross-Domain Call Microcode 263

Index **277**

List of Figures

2.1 Lampson's Access Matrix 26

3.1 Kernel/Emulator Approach 36

4.1 DBMS Protected Subsystem Example 44

6.1 Protected Subsystems, Domains, and Processes 56
6.2 Domains and Non-Discretionary Controls 60

7.1 Non-discretionary Limited Capabilities 70

8.1 Access-Control-List Limited Capabilities 80

9.1 Trojan Horse in Action 84
9.2 Trojan Horse Blocked 87
9.3 Example of Cartouches in a Flex edfile 92

11.1 Quota Causing Storage Channel Problem 108
11.2 Multics Revocation Scheme 110
11.3 Redell's Revocation Scheme 112
11.4 Recursive Redell Revocation 113
11.5 Revocation with Eventcounts 114
11.6 Chained Page-Table Entries 116

12.1 Multics Quota Example 119

14.1 Simple Refinement Example 132

16.1 IBM System/38 Hashed Address Translation 148
16.2 Hashed Address Translation with Open Addressing 151
16.3 Hashed Address Translation with Shared Page Chains 152
16.4 Hashed-Page-Table Entry Format 154
16.5 Hashing Algorithm 155
16.6 Microcode to Compute Hash Function 156

17.1 Format of Cross-Domain Linkage Table Entry 171
17.2 Format of C-stack Frame 171
17.3 Format of Domain Control Block (DCB) 172
17.4 Microcode to Save and Clear Registers Unconditionally . . . 173
17.5 Microcode to Save and Clear Registers Using Masks 174

17.6 RISC Optimized Cross-Domain Call 175
17.7 C-stack Frame Usage 178
17.8 Format of Revised C-stack Frame 179

B.1 Atlas Page Table 219
B.2 Multics Address Translation 220
B.3 VAX Address Space Mapping 221

C.1 Fields of a Virtual Address 223
C.2 Fully-Associative Translation Buffer 224
C.3 Direct-Mapped Translation Buffer 225
C.4 Two-Way Set-Associative Translation Buffer 226

E.1 VAX-11/730 System 238
E.2 KA730 Block Diagram 240
E.3 Data Path Block Diagram 241

G.1 Lower-Level Type Managers for SCAP Security Kernel . . . 251

I.1 JSB Measurement Code 259
I.2 SVPCTX/LDPCTX Measurement Code, Part 1 261
I.3 SVPCTX/LDPCTX Measurement Code, Part 2 262
I.4 Microcode Invoker 263

List of Tables

2.1 Published Penetration Successes 22

17.1 Register Usage on Cross-Domain Call 167
17.2 Register Usage on Cross-Domain Return 168

18.1 Cross-Domain-Call/Return Performance Results 184

D.1 VAX Data Types 230
D.2 Processor Status Longword Fields 231
D.3 Architecturally Defined Internal Processor Registers (IPRs) . . 232
D.4 PTE Protection Codes 234

E.1 VAX-11/730 Microcode Shifts and Rotates 242
E.2 VAX-11/730 Microprogram Modules 243

ACKNOWLEDGEMENTS

There are many people that I must thank for help during the long and sometimes difficult process of Ph.D. research. Foremost are Andrew J. Herbert, my thesis supervisor and Steven B. Lipner, my supervisor at Digital. They have spent many hours discussing my work and guiding it in proper directions.

The research might not have started at all, without the crucial questions that Prof. Maurice Wilkes and Dr. William Strecker asked concerning secure capabilities and performance. Prof. Roger Needham and Prof. David Wheeler reviewed many of my interim reports and provided much useful guidance. Tim Leonard taught me about the VAX-11/730 microcode, and we had many useful discussions on processor architecture and on process structuring.

A number of people must be thanked for technical discussions and/or reviewing some of my early design notes and drafts of this dissertation. They include Jean Bacon, Mike Burrows, Martyn Johnson, John Line, Mark Lomas, Nick Maclaren, and Julian Pardoe at Cambridge, Sape Mullender at Cambridge and the Centre for Mathematics and Computer Science at Amsterdam, and Dileep Bhandarkar, Morrie Gasser, Judy Hall, B. J. Herbison, Marty Hurley, Cliff Kahn, Alan Kotok, Drew Mason, Simon Steeley, and Joe Tardo at Digital. Richard Jordan particularly helped in the presentation of the cross-domain call optimization. Beverly Elliott and Martha Moore, librarians at Digital, and Ann Clocksin, librarian at the Computer Laboratory, deserve particular thanks for help with literature searching and ordering very obscure technical reports and articles from all over the world.

The Digital Equipment Corporation was most generous in funding my research under its Graduate Engineering Education Program (GEEP). I must thank Shirley Stahl, the manager of GEEP, for making it possible for me to come to Cambridge, and Dolores Miller and Terry Sarandrea for all their help in handling the various crises that happen to someone working several thousand miles away. Digital's office in Newmarket was most generous to provide computing and technical support for network communications. Particular thanks must go to Martin Black, Kim Burgess, Pete Darby, and Terry Young.

Finally, I must thank my wife, Carol-Lynn, for her constant support throughout my research and for enduring several transatlantic moves. She also provided editorial and technical assistance throughout my research. In particular, she made extensive modifications to the TEXindex program that produced the index to this dissertation, designed a METAFONT version of the Digital logo, drew several of the figures in the dissertation, and extensively reviewed and commented on the text.

The dissertation was prepared using Leslie Lamport's document preparation system, LATEX. Several of the figures were drawn with TEXdraw, a line drawing package written by Tom Taylor, a student of Hank Levy's, at the University of Washington. The index was generated with TEXindex, originally implemented by Terry Winograd, Bill Paxton, Skip Montanaro, and Charles Karney, and significantly improved by Carol-Lynn Covitt Karger.

DECLARATION

I hereby declare that this dissertation is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other University. I further state that no part of this dissertation has already been or is being concurrently submitted for any such degree, diploma or other qualification. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

DISCLAIMER

This dissertation presents the opinions of its author, which are not necessarily those of the Digital Equipment Corporation. Opinions expressed in this dissertation must not be construed to imply any product commitment on the part of the Digital Equipment Corporation.

TRADEMARKS

METAFONT is a trademark of Addison Wesley Publishing Company.
TEX is a trademark of the American Mathematical Society.
UNIX is a registered trademark of the American Telephone and Telegraph Company.
AT&T is a trademark of the American Telephone and Telegraph Company.
CRAY is a registered trademark of CRAY Research, Inc.
CLIPPER is a trademark of Fairchild Semiconductor Corporation.
Gould and UTX/32S are trademarks of Gould, Inc.
IBM is a registered trademark of the International Business Machines Corporation.
Intel is a registered trademark of the Intel Corporation.
KeyKOS is a trademark of Key Logic, Inc.
Microsoft is a registered trademark of the Microsoft Corporation.
XENIX is a trademark of the Microsoft Corporation.
UNIVAC is a registered trademark of the Sperry Corporation.
SunOS is a registered trademark of Sun Microsystems, Inc.
Ada is a registered trademark of the U.S. Government Ada Joint Program Office.
Zilog is a registered trademark of Zilog, Inc.
Z8000 is a trademark of Zilog, Inc.

The following are trademarks of the Digital Equipment Corporation:

DATATRIEVE	MicroVAX	PDP-11	VAX	VAX-11/785
DEC	MicroVAX-II	PDP-11/45	VMS	VAX 8550
DEC/MMS	PDP	ULTRIX	VAX/VMS	VAX 8600
DECtape	PDP-1	ULTRIX-32	VAX-11/730	VAX 8800
DECtape II	PDP-10	UNIBUS	VAX-11/780	digital

SUMMARY

This dissertation examines two major limitations of capability systems: an inability to support security policies that enforce confinement and a reputation for relatively poor performance when compared with non-capability systems.

The dissertation examines why conventional capability systems cannot enforce confinement and proposes a new secure capability architecture, called SCAP, in which confinement can be enforced. SCAP is based on the earlier Cambridge Capability System, CAP. The dissertation shows how a non-discretionary security policy can be implemented on the new architecture, and how the new architecture can also be used to improve traceability of access and revocation of access.

The dissertation also examines how capability systems are vulnerable to discretionary Trojan horse attacks and proposes a defence based on rules built into the command-language interpreter. System-wide garbage collection, commonly used in most capability systems, is examined in the light of the non-discretionary security policies and found to be fundamentally insecure. The dissertation proposes alternative approaches to storage management to provide at least some of the benefits of system-wide garbage collection, but without the accompanying security problems.

Performance of capability systems is improved by two major techniques. First, the doctrine of programming generality is addressed as one major cause of poor performance. Protection domains should be allocated only for genuine security reasons, rather than at every subroutine boundary. Compilers can better enforce modularity and good programming style without adding the expense of security enforcement to every subroutine call. Second, the ideas of reduced instruction set computers (RISC) can be applied to capability systems to simplify the operations required. The dissertation identifies a minimum set of hardware functions needed to obtain good performance for a capability system. This set is much smaller than previous research had indicated necessary.

A prototype implementation of some of the capability features is described. The prototype was implemented on a re-microprogrammed VAX-11/730 computer. The dissertation examines the performance and software compatibility implications of the new capability architecture, both in the context of conventional computers, such as the VAX, and in the context of RISC processors.

Part I

Background

Chapter 1

Introduction

1.1 History of the Project

There has been a long-standing debate in the computer security community over the relative merits of capability-based systems and access-control-list systems. This dissertation has evolved from a particular incident that occurred in late 1981 when Prof. Maurice V. Wilkes arranged a meeting between Dr. Andrew Herbert and myself to discuss whether one could make capability-based systems adequately secure.[1] At the time, I was squarely in the access-control-list camp, based on my experiences designing security kernels and non-discretionary controls. (See Section 3.1.2 for a definition of non-discretionary controls.) Prof. Wilkes and Dr. Herbert were in the capability camp, based on their work in the design of the Cambridge Capability System (CAP). I entered the meeting concerned that capabilities could not adequately solve the confinement problem, described in Section 7, while Dr. Herbert entered the meeting convinced that capabilities were sufficiently powerful to model whatever was needed. We left the meeting agreeing that conventional capability systems did have serious weaknesses, but that a new secure capability architecture was possible that would solve not only the confinement problem, but also many other problems of auditability and performance. The immediate result of the meeting was a paper that we jointly published [119] in 1984, and the long term result was that I came to the University of Cambridge to do research on the detailed design of a system to implement the new capability scheme, with Dr. Herbert as my supervisor. This dissertation is the result of that research.

The principal goal of the dissertation is to describe the implementation of a modified capability architecture that can achieve much higher levels of security than had been possible with conventional capability systems. The resulting

[1] The meeting was held in Digital Equipment Corporation's Corporate Research Group, where Prof. Wilkes and I worked, he having joined Digital upon retiring from the University of Cambridge.

system should be capable of achieving at least an **A1** security rating from the National Computer Security Center [59]. Indeed, the use of capabilities and protection domains should help in achieving ratings beyond the current maximum **A1**. (Appendix A contains a brief summary of the computer security evaluation criteria.)

The second goal of the dissertation is to show how the new capability architecture achieves an acceptable level of speed. This goal was also inspired by Prof. Wilkes, who asked me why I expected a secure capability architecture to perform any better than previous capability systems that had a reputation for poor performance. This dissertation will show how the use of reduced instruction set computer (RISC) technology will aid in not only making the system more secure, but also in providing a much higher level of performance than was possible in previous capability architectures. Related to the issue of performance improvement is the question of how much hardware support is needed for a capability-based system. The dissertation argues that much less hardware support is needed than was previously assumed. The principal need is for hardware support for fast context switching and little else.

1.2 What is SCAP?

The new secure capability architecture is called SCAP, because it is a secure derivative from the Cambridge CAP system. The basic difference between SCAP and a conventional capability system is that in a conventional system, a capability is necessary and sufficient to gain access to an object, while in SCAP, a capability is necessary but not sufficient to gain access to an object. By this simple change in the definition of a capability, SCAP can achieve much higher levels of security and, in particular, solve the confinement problem, as defined in Section 3.1.1, that has always been a major difficulty for capability systems.

The dissertation examines SCAP in two contexts, a description of the SCAP architecture, and an implementation of SCAP on a VAX computer. The architecture of SCAP is based on the current theories of reduced instruction set computers (RISC), while the VAX implementation is particularly aimed at preserving software compatibility with existing operating systems.

1.3 Overview of Experimental Environment

I have used the VAX-11/730 processor as a tool to measure the performance of some of the enhancements described in this dissertation. The VAX-11/730 is attractive for this purpose, because many of the enhancements must be implemented in microcode, and the VAX-11/730 is particularly easy to microcode. As described in more detail in Appendix E, the VAX-11/730 CPU (called the

KA730) is vertically microprogrammed and the microcode executes from a writable-control store (WCS) that can be easily reloaded from tape. Thus, it was relatively easy to microcode the two major experiments described in this dissertation: hashed page tables and cross-domain-call optimizations.

Two different VAX-11/730 systems were actually used for the experiments: one in the University of Cambridge Computer Laboratory, and the other in Digital's Secure Systems Development Group. The hashing experiments of Section 16.6 were run at Digital's facility, and the cross-domain-call experiments of Chapter 18 were run at the University of Cambridge Computer Laboratory. Certain of the cross-domain-call experiments were also run on a VAX 8550 at Digital's facility.

1.4 Plan of the Dissertation

The dissertation is broken into four parts. Part I is background material, covering the need for computer security, a survey of computer security models, and the principles of capability systems. Part II is a summary of the major features of the SCAP architecture and detailed definition of how SCAP deals with protected subsystems, domains, and processes. Part III addresses improving security in capability systems. It describes how the SCAP architecture solves the confinement problem and then looks in general at traceability of access discretionary Trojan horses, commercial data-integrity models, and revocation, secure garbage collection. Part IV focuses on performance and addresses the problems of programming generality, translation-buffer design, page-table structures, optimization of cross-domain calls, and fault and interrupt handling. Finally, the dissertation concludes with a number of appendices that contain tutorial and supplemental material on a number of topics covered in the main body of the dissertation. The last three appendices present design sketches for future research on SCAP and annotated code listings.

Chapter 2

The Need for Security

At this stage in the development of the computer industry, there is little disagreement that security and protection of information are essential to many computer systems. The popular press is filled with reports of computer break-ins, and, with the publication of beginners' handbooks [46, 133], the techniques of system penetration are widely accessible. Breaches of computer security have also been a popular topic in fiction, the earliest example being a 1953 short story by Poul Anderson [8].

2.1 Classes of Vulnerabilities

Computer security systems have many forms of vulnerabilities. This section summarizes the major types of vulnerabilities that will be discussed in the dissertation.

2.1.1 Browsing

The simplest and most common threat is *browsing*. Browsing is an unauthorized user experimenting with a computer system to see what information can be retrieved by simply asking. For example, a user may attempt to read other users' files to see what they contain. An unauthorized individual might try to guess passwords by connecting over a dial-up line or through a computer network. A wiretapper might attempt to listen to on-going communications either to terminals or other computers in a local or wide area network.

Most of the published "hacker" incidents have been based on browsing. Browsing attacks can be countered with techniques such as file system controls, good password management, and encryption of communications lines. Consequently, this dissertation will not address such attacks, but will assume that good password management is in place, that all communications lines are properly encrypted, and that all users have been properly identified to the system.

2.1.2 Unauthorized Acts of Authorized Individuals

The largest computer security losses today occur due to *unauthorized actions of properly authorized individuals*, that is, when insiders exploit a computer system for their own gain. The insider will have access to some set of computer functions as part of his or her job. However, the insider may enter false information into the computer in such a way that he or she can profit from the results. Poul Anderson's [8] earliest example of computer security penetration in fiction described an insider penetration.

It is difficult to distinguish between authorized and unauthorized actions of legitimate users, but there are some attempts at developing commercial data-integrity models to deal with this problem. This area will be discussed in detail in Chapter 10.

2.1.3 Direct Penetration

Direct penetration exploits flaws that may be present in either the hardware or software implementation of a computer-security system. If a penetrator can locate a flaw in the implementation, then the penetrator can exploit that flaw to gain access to sensitive information. Jim Anderson developed one of the earliest taxonomies of direct penetrations in the now-classic 1972 Air Force computer-security-panel report [6]. Direct penetrations of most major computer systems have been reported in the literature, as shown in Table 2.1.[1]

Computer	Operating System	Citation
Burroughs B6700	MCP	[232]
DEC PDP-10	BBN Tenex	[1]
DEC VAX	VAX/VMS	[47, pp. 113–114]
Honeywell H6000	GCOS	[6, 186]
Honeywell H645/H6180	Multics	[121, 38]
IBM System/360	OS/360	[1]
IBM System/370	MVS	[170]
IBM System/370	VM/370	[10]
IBM System/370	MTS	[90]
IBM System/38	System/38 OS	[148]
Sperry 1100	1100 Series OS	[1]
various	AT&T UNIX	[82]

Table 2.1: Published Penetration Successes

[1]Systems not mentioned in the table should not be assumed to be invulnerable to direct penetration. Rather, reports of their penetrations have simply not been published in the open literature.

It is interesting to note that the extensive literature on reported computer crimes includes almost no cases in which the penetrator used a direct penetration of the operating system.[2] This is not because direct penetrations do not exist or direct penetrations are terribly difficult to find, but rather because the existing procedural security controls in most systems are so ineffective that simple browsing attacks will suffice in most cases. Likewise, application controls are generally so weak that authorized individuals can easily take unauthorized actions with legitimate programs, rather than resorting to a technically sophisticated direct penetration. As the simpler attack paths are closed, the direct penetration of the operating system will be more commonly exploited.

The primary defence against direct penetrations is the use of security kernel technology to give some assurance that the operating system security controls are in fact implemented correctly. Security kernels will be discussed in Section 2.2.3. A principal goal of this dissertation is the use of capability-based systems to support the implementation of security kernels.

It is frequently argued that direct penetrations can be avoided by preventing most on-line programming and restricting users to a simplified higher level language or to a simple query-based transaction system. However, Jim Anderson [6] showed the vulnerability of even highly restricted languages in a penetration of the Honeywell H635/GCOS III Time Sharing System. He was restricted to a minimal subset of FORTRAN IV that barred the use of subroutines and all I/O statements. However, Anderson gained illegal access to the system password file, using only FORTRAN IV arithmetic-assignment statements and ASSIGNED GOTO statements.

A query-only transaction system would not be vulnerable to Anderson's use of ASSIGNED GOTO, but it could be attacked through the clandestine software modification attacks described in the next section.

2.1.4 Trap-Door and Trojan-Horse Attacks

Much more insidious than the direct penetrations are the *trap-door* and *Trojan-horse* attacks that involve clandestine software (or hardware) modifications. In a direct penetration, an attacker must discover a flaw in the security system and then must find a way to exploit the flaw. With a trap door or a Trojan horse, the attacker uses a custom-designed flaw that was previously installed in the system. The distinction between traps doors and Trojan horses is subtle and not of great importance—a trap door is a modification to system supplied software, perhaps introduced by the computer system developers. A Trojan horse is a modification to software that the user of a system borrows or purchases to add on to his existing system.

[2] Probably the only exception is a very recent incident involving VAX/VMS systems [47, pp. 113–114].

The term *Trojan horse* was first used in a computer security context by Dan Edwards [6]. Schell and I [121] demonstrated the feasibility of trap-door insertion by surreptitiously introducing a benign trap door into the standard Multics operating system. The normal Multics distribution system then delivered the trap door to actual customer sites. We also hypothesized the existence of compiler trap doors that could install operating system trap doors and preserve their existence across re-compilations of the operating system or the compilers themselves. In his Turing Award Lecture, Thompson [210] discussed an actual trap door in the UNIX C compiler that would preserve its own existence and would plant a trap door in the UNIX operating system.[3]

An even more dangerous form of Trojan horse is the self-replicating *virus*. The virus [43] is a clandestine software modification that installs copies of itself on other machines and thus spreads itself, much as real viruses do. The virus lies dormant in a computer system, hidden in an applications program or a utility. Whenever the infected program is run, the virus inspects its environment to see if it can spread itself. For example, if the virus notices a new disk mounted on the system, it might copy itself there. If the virus is run by a privileged user on a time-sharing system, the virus might copy itself from an unprivileged applications program into a privileged portion of the operating system.

Desmedt [60] has hypothesized the existence of viruses in hardware that are propagated by software viruses in the CAD programs used to design chips. While interesting, such CAD-based viruses seem less feasible than Thompson's C-compiler trap door, because unlike an operating system, a particular chip is only run through a CAD program a relatively few times. Then, the masks are created and the chips are produced. Planting a Trojan horse requires advance knowledge of the design of the target, be it a chip or an operating system. Because the chip is run through the CAD system less frequently than an operating system is recompiled, the window of opportunity to plant the Trojan horse is much larger for the operating system.

Because they are custom-designed security penetrations, Trojan horses can easily defeat the security of even restricted query-only systems. For example, a Trojan horse or a trap door located in the terminal-input routine of an operating system could wait for a particularly unlikely sequence of characters to be typed and then accept arbitrary commands from the terminal. Similarly, a network-driver routine could scan incoming packets for a particular bit pattern. When the pattern was located, the network driver (which runs in the most privileged part of most operating systems) would treat the remainder of the packet as binary machine code and begin executing it. Thus, the Trojan horse would function as a password-protected bootstrap loader that would allow a penetrator arbitrary access to the target system, no matter how restricted the applications

[3]The "unknown Air Force document" that Thompson cites is by Karger and Schell [121].

were supposed to be. Schell and I [121, Appendix C] demonstrated that such a bootstrapping Trojan horse would require no more than a ten-word modification to the object code of an operating system.

Until recently, the threat of Trojan-horse attack had remained a hypothesis of security researchers and authors of thriller novels [30], but no real attacks had occurred. Recently, however, there have been numerous incidents of actual Trojan-horse and viral attacks on public-domain microcomputer software [152], and articles in computer journals intended for the general reader [235] have contained extremely detailed instructions on how to build self-replicating Trojan horses and viruses. Very recently, a commercial software package was found to have been infected by a virus [175]. Thus, the hypothetical threat of Trojan horses in commercial software that Schell and I demonstrated in 1973 [121] has become the reality of 1988. The Trojan-horse threat can no longer be dismissed as a concern of only the most sensitive system managers.

One of the primary goals of this dissertation is to demonstrate how to deal with the Trojan horse threat in a capability-based system. I will show how traditional capability-based systems are vulnerable to Trojan horse attacks, and how a modified capability architecture can deal with both non-discretionary (in Chapter 7) and discretionary (in Chapter 9) Trojan horses.

2.2 Development of Secure Systems

Most security models are based on Lampson's access matrix [132] in which the rows of the matrix represent the active entities or *subjects*[4] of the system, and in which the columns of the matrix represent the passive information-containing *objects* of the system. Figure 2.1 shows a typical access matrix. The rights that a subject holds to an object can be found at the intersection of the row and column belonging to the subject and object. Thus in the example, Subject 1 has read and write permission to Object 2, but Subject 2 has only read permission, and Subject N has no permission at all to Object 2.

Lampson's matrix appears quite simple, but in practice its interpretation can be quite complex, because the categories of objects must often include the subjects and the access-matrix entries, themselves. This complexity comes from the desire to control not just access to files, but also access rights to control jobs and processes and access rights to change other access rights. Furthermore, in any real system, the access matrix would be very sparse, as most objects fall into two classes: private and accessible by only one or a small number of subjects, or public and accessible to everyone. Thus in practice, computer systems do not

[4]Lampson actually used the term *domain* rather than *subject*. The term *subject* was first used in this context by Graham and Denning [81].

	Object 1	Object 2	···	Object M
Subject 1	R	RW	···	null
Subject 2	null	R	···	RW
⋮				
Subject N	RW	null	···	R

Figure 2.1: Lampson's Access Matrix

store their access rights in matrix form. Instead, they use some form of access control list or capability representation.

2.2.1 Access-Control-List Systems

Access-control-list (ACL) systems view Lampson's access matrix by columns. An access control list is attached to each object and consists of a list of subjects and their associated access rights. When a subject wishes to gain access to an object, the system must look up the subject in the access control list to determine whether or not to grant access. Most systems compress the ACL by assuming that a subject who is not mentioned gets no access rights, and by providing a mechanism to name groups of users in a single entry. Group names also make it easy to add or remove a user from the ACL's of many objects in a single operation, avoiding the need to scan many objects. Full access control lists were first implemented for Multics [49] and for the Titan multi-access system [12].

In one way or another, an access-control-list system must check the ACL on every reference to an object. Obviously, such continuous checking could be extremely slow, unless some form of caching were provided. The Multics approach is to cache the results of an ACL check in the protection bits of a segment descriptor word (SDW). The hardware then performs the required protection check on every memory reference. By contrast, the UNIX approach is to check the ACL at the time a file is opened and to record the permissions in a software table of open files. That table is then checked on every disk I/O operation.

2.2.2 Capability systems

Capability systems view Lampson's access matrix by rows. Each subject possesses a list of objects to which the subject has access. The entries in the list are called *capabilities*. Each capability names the object and describes where the object is stored and what access rights are to be granted to the possessor of this capability. In a conventional capability system, possession of a capability is both necessary and sufficient to gain access to an object.

Capabilities closely resemble tickets or keys in the physical world. A capability can be passed from one subject to another, and copies may be made of the capability. However, the system must ensure that capabilities cannot be forged or tampered with. Dennis and Van Horn [58] originally proposed the concept of capabilities in 1966. Since that time, a number of software capability systems have been implemented, including the MIT PDP-1 time-sharing system [3], CAL for the CDC 6400 [130], and HYDRA for the C.mmp multiprocessor [238]. The early software implementations of capabilities had serious performance problems, principally because of either a total lack of virtual address translation hardware or too small a virtual address space. A number of hardware capability systems also have been implemented, including the Plessey System 250 [48], the Cambridge CAP computer [231], the IBM System/38 [104], the Intel 432 [169], and Flex [72]. See Chapter 4 for a brief tutorial on capability systems.

2.2.3 Security Kernels

The history of direct penetrations and Trojan-horse attacks, described in Sections 2.1.3 and 2.1.4, makes it evident that simply implementing an access-control-list system or a capability system is not sufficient to support a high level of security. The U.S. Air Force sponsored a Computer Security Technology Planning Study in 1972 to examine the construction of highly secure computer systems. The principal result [6] was the recommendation of a new style of secure operating system, called a *security kernel*.

The panel identified three requirements for a security kernel:

- The kernel must mediate all references to data.
- The kernel must protect itself against tampering.
- The kernel must be simple enough that independent evaluators can assess whether it will operate correctly.

The first requirement can be met by completely virtualizing all references to objects, that is, by making all references to objects indirect.[5] Virtual memory particularly helps by making protection checks possible on all memory references at relatively low performance cost. Resistance to tampering can be achieved by isolating the kernel in its own protection domain. Smith [200] and Tangney [207] have suggested that the CPU support a minimum of three protection domains, one for the kernel, one for the rest of the operating system, and one for user code. Finally, assurance of correct operation can be aided by making the security kernel much smaller and simpler than conventional operating systems and by applying

[5]Prof. David Wheeler has suggested that most problems in computing can be solved by adding a level of indirection.

the best techniques of modular design and formal specification and verification to assist in the design and implementation assurance.

The final requirement, that the security kernel be simple enough to assess correctness, is the hardest to achieve. Although security kernels are supposed to be small and simple, in practice they have been quite large [185]. Implementing the security kernel as a collection of small protection domains would provide better structuring and provide additional confidence in its effectiveness. Each of those domains could be analyzed independently, making the overall analysis of the reference monitor easier. Janson [108] originally suggested implementing a security kernel as a set of type managers in separate protection domains. At least two kernels have been developed using levels of abstraction within a single protection domain [187, 120]. These kernels have not had adequate protection between the levels of abstraction. A particular level that malfunctioned could arbitrarily destroy or subvert other levels. Placing the different levels of the security kernel in separate protection domains [166, 28, 119] can achieve two things. First, the separation of levels can limit the spread of damage caused by a kernel malfunction. Second and more important, separating the levels should make assurance of correct implementation easier by limiting the amount of code that must be proven or manually inspected at any one time.

This dissertation examines how a capability-based architecture that supports protected subsystems can be used to assist in the construction of security kernels. Section G examines how a security kernel could be subdivided into a collection of protected subsystems, to limit damage propagation if a part of the kernel should fail and to assist in the formal verification of the kernel's implementation.

2.3 Assumptions

The remainder of this dissertation is based on a number of security assumptions stated here. These assumptions limit the scope of the dissertation to certain interesting problems. However, they are not unreasonable for practical secure systems in use today.

1. Users connect to systems via terminals or networks, and there is adequate physical security for CPUs and disk drives, etc. Office workstations introduce serious physical security problems that are not addressed in this dissertation.

2. Computer hardware is sufficiently reliable that the chance of random failures leading to security problems is negligible.

3. Communications lines are either physically protected or encrypted to prevent any kind of tapping or tampering.

4. User authentication techniques, whether based on passwords or physical characteristics, are adequate to correctly identify users.

5. Systems management and operations staff are fully trustworthy.[6]

6. Each user can be assumed trustworthy to some specified level.

7. Security kernel software can be developed and distributed in a tamper-resistant manner.

8. Other software may be subject to unauthorized tampering.

Thus, the focus of the dissertation is on internal software security of systems, rather than on physical, communications, or procedural security.

[6]The trustworthiness of systems management and operations staff is the least credible of the assumptions. The Clark and Wilson commercial-integrity model, discussed in Section 3.2.3 provides a little help, but this area remains a fundamentally unsolved problem.

Chapter 3

Models of Security

A formal model of security is essential when reasoning about the security of a system. Without an unambiguous definition of what security means, it is impossible to say whether a system is secure. Security models can be broken down into three major categories, listed in order of complexity:

- models that protect against unauthorized disclosure of information,
- models that protect against unauthorized tampering or sabotage, and
- models that protect against denial of service.

Protection against disclosure of information has been understood the longest and has the simplest models. Protection against tampering or sabotage has been less well understood and appropriate models are only now under development. Protection against denial of service is not well understood today, and there may be Turing-machine halting arguments against ever solving denial of service. This chapter briefly summarizes the security models that are used throughout the remainder of the dissertation and discusses why the particular models have been chosen.

3.1 Preventing Information Disclosure

The first requirement of most security systems is preventing unauthorized disclosure of information. Indeed, the basic point of Lampson's access matrix and the various access-control-list and capability-based systems that have been implemented is to control who may have access to which objects. This section examines two classes of mechanisms: discretionary access controls and non-discretionary access controls.

3.1.1 Discretionary Access Controls

Discretionary access controls are the commonly available security controls based on the fully general Lampson access matrix. They are called discretionary, because the access rights to an object may be determined at the discretion of the owner or controller of the object. Both access-control-list and capability systems are examples of discretionary access controls. The presence of Trojan horses in the system can cause great difficulties with discretionary controls, because a Trojan horse could surreptitiously change the access rights on an object or could make a copy of protected information and give that copy to some unauthorized user. All forms of discretionary controls are vulnerable to this type of Trojan-horse attack. A Trojan horse in an access-control-list system could surreptitiously change the ACL of an object. A Trojan horse in a capability system could make a copy of a capability for a protected object and then store that capability in some other object to which a penetrator would have read access. In both cases, the information is disclosed to an unauthorized recipient.

Lampson [131] has defined the *confinement problem* as determining whether there exists a series of operations in a security system that will ultimately leak some information to some unauthorized individual. Harrison, Ruzzo, and Ullman [89] have shown that there is no solution to the confinement problem for fully-general, discretionary access controls, such as either a general access-control-list or capability system. Their argument is based on modelling the state transitions of the access matrix as the state transitions of a Turing machine. They show that solving the confinement problem is equivalent to solving the Turing-machine halting problem.

The paths over which a Trojan horse leaks information are called *covert channels*. Covert channels can be divided into two major categories: *storage channels* and *timing channels*. Information can be leaked through a storage channel by changing the values of any of the state variables of the system. Thus, contents of files, names of files, and amount of disk space used are all examples of potential storage channels. A Trojan horse can leak information through a storage channel in a purely asynchronous fashion. There are no timing dependencies.

By contrast, information can be leaked through a timing channel by modifying the length of time that system functions take to complete. For example, a Trojan horse could encode information into deliberate modifications of the system page-fault rate. Timing channels all use synchronous communication and require some form of external clocking.

3.1.2 Non-Discretionary Access Controls

Non-discretionary access controls have been developed to deal with the Trojan horse problems of discretionary access controls. The distinguishing feature of non-discretionary access controls is that the system manager or security officer

may constrain the owner of an object in determining who may have access rights to that object.

Lipner [143] and Denning [56] have shown that for *lattice security models*, unlike for fully general access matrices, it is possible to solve the confinement problem. All non-discretionary controls, to date, have been based on lattice security models.

3.1.2.1 Elements of the Lattice Model

A lattice security model consists of a set of *access classes* that form a partial ordering. Any two access classes may be less than, greater than, equal to, or not ordered with respect to one another. Two access classes that are not ordered are called *disjoint*. Furthermore, there exists a lowest access class, called *system low*, such that system low is less than or equal to all other access classes, and there exists a highest access class, called *system high*, such that all other access classes are less than or equal to system high.

A very simple lattice might consist of two access classes: LOW and HIGH. LOW is less than HIGH. LOW is system low, and HIGH is system high. A slightly more complex example might be a list of sensitivity levels, such as UNCLASSIFIED, CONFIDENTIAL, SECRET, and TOP SECRET. Each level in the list represents data of increasing sensitivity.

There is no requirement for a strict hierarchical relationship between access classes. The U.S. military services use a set of access classes that have two parts: a sensitivity level and a set of categories. Categories represent compartments of information for which an individual must be specially cleared. To gain access to information in a category, an individual must be cleared, not only for the sensitivity level of the information, but also for the specific category. For example, if there were a category NUCLEAR, and some information classified SECRET–NUCLEAR, then an individual with a TOP SECRET clearance would not be allowed to see that information, unless the individual were specifically authorized for the NUCLEAR category.

Information can belong to more than one category, and category comparison is done using subsets. Thus, in the military lattice model, for access class A to be less than or equal to access class B, the sensitivity level of A must be less than or equal to the sensitivity level of B, and the category set of A must be an improper subset of the category set of B. Since two category sets may be disjoint, the complete set of access classes has only a partial ordering. There is a lowest access class, {UNCLASSIFIED–no categories}, and a highest access class, {TOP SECRET–all categories}. The access classes made up of levels and category sets form a lattice.

Many other security policies also form lattices. One could define other lattices that model commercial security requirements. Section 3.2.2 describes one such commercial security lattice.

The remainder of this dissertation assumes the use of the military lattice, whenever non-discretionary controls are mentioned, unless the use of other lattices is explicitly noted.

3.1.2.2 Defeating Trojan Horses

Lattice models were first developed at the MITRE Corporation by Bell and LaPadula [15, 16, 17] and at Case Western Reserve University by Walter [225] to formalize the military security model and to develop techniques for dealing with Trojan horses that attempt to leak information.[1] At the time, no one knew how to deal with Trojan horses at all, and it came as quite a surprise that two quite simple properties could prevent a Trojan horse from compromising sensitive information.

First, the *simple security property* says that if a subject wishes to gain read access to an object, the access class of the object must be less than or equal to the access of the subject. This is just a formalization of military-security-clearance procedures that one may not read a document unless one is properly cleared.

Second, the *confinement property*[2] requires that if a subject wishes to gain write access to an object, the access class of the subject must be less than or equal to the access class of the object. The net effect of enforcing the confinement property is that any Trojan horse that attempts to steal information from a particular access class cannot store that information anywhere except in objects that are classified at an access class at least as high as the source of the information. Thus, the Trojan horse could tamper with the information, but it could not disclose the information to any unauthorized individual. A more detailed discussion of the confinement property and its interpretation in the context of a practical time-sharing system can be found in [14]. A survey on formal security models in general can be found in [134].

3.1.3 Retrofitting Non-discretionary Security

Adding non-discretionary controls to existing operating systems has been the topic of much research and development since 1973. Over the years, two basic

[1] The non-discretionary models were based on earlier work on the ADEPT–50 operating system [227] and on the security enhancements to the WWMCCS-GCOS operating system [146, pages 147–148]. Neither of those systems could solve the Trojan-horse problem, although ADEPT–50 contained a partial solution.

[2] The confinement property was called the *∗–property* in [16]. It was so named as a place holder until a better name could be found. No better name was found prior to publication, so ∗–property was used, and much of the literature on non-discretionary controls continues to use the name ∗–property.

approaches have been taken that result in relatively secure systems that retain a high level of compatibility with existing code. These approaches are the incremental addition of features and the kernel/emulator approach.

3.1.3.1 Incremental Addition of Features

Incremental addition of features is simply evolving the existing operating system to add non-discretionary security controls. Such modifications may be relatively easy or difficult, depending on how well the operating system virtualizes the real resources. The extent to which resources are not virtualized is usually a good predictor of where a system might have significant storage-channel problems.

The drawback of incremental addition of features is that one can never achieve a high level of assurance that the features are correctly implemented, because existing operating systems are so large and complex (typically millions of lines of code). As a result, incremental addition of features can only get an operating system up to a B2 security rating [59].

The classic example of incremental addition of security features is the Multics Access Isolation Mechanism (AIM) [229] that has received a B2 rating. More recent efforts at incremental addition of security features include the VAX/VMS security enhancements [23] targeted for a possible B1 rating. There have been several efforts to incrementally add security features to the UNIX operating system, including Linus IV [129], the IBM Secure XENIX system [79], and Sun's Secure SunOS [4]. The incremental improvements to UNIX are typically aimed at a B1 or B2 rating, at most.

3.1.3.2 Kernel/Emulator Approach

The *kernel/emulator approach* to adding non-discretionary security is aimed at a much higher level of security, typically a B3 or A1 security rating from the National Computer Security Center. The kernel/emulator approach depends on there being a large quantity of code that is not security relevant in the most privileged domain of most operating systems. The operating system is replaced by a security kernel that runs in the most privileged domain and an operating-system emulator that runs in a less-privileged domain than the security kernel, but which is still protected from user programs. Figure 3.1 shows a typical kernel/emulator system.

Just as the Multics AIM system is the classic example of incremental addition of security, the Multics Guardian project is the classic example of a kernel/emulator design [190]. Funding limitations never permitted completion of the Multics kernel design. The kernel/emulator approach has been used with varying degrees of success for several secure-UNIX systems, including the MITRE Secure UNIX [236], UCLA Secure UNIX [174], and KSOS [153]. Note that the

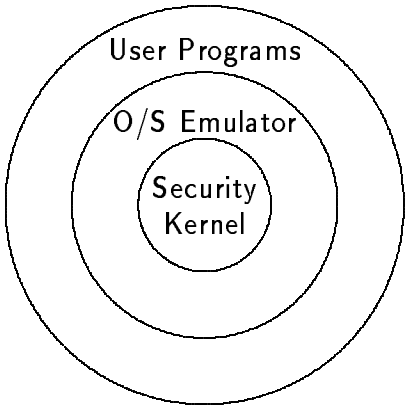

Figure 3.1: Kernel/Emulator Approach

kernel/emulator secure-UNIX systems were able to achieve a significantly higher security rating than the incremental-addition systems, listed in Section 3.1.3.1.

All of the kernel/emulator designs have suffered from two problems. First, the introduction of a security boundary between the emulator and the kernel can badly affect performance. Second, and perhaps more crucial, the design and implementation of an operating-system emulator is as large and complex a task as building the operating system itself. If the non-secure version of the operating system continues to evolve, the emulator must track new versions of the operating system. The software-development resources required to build and maintain such an emulator can be prohibitively large.

3.2 Preventing Tampering and Sabotage

The military's emphasis on the lattice security models, as typified by the requirements of the National Computer Security Center's evaluation criteria [59], has often been criticized as neglecting the issues of information tampering. However, the history of the development of the lattice security model shows that the military services have always been concerned both with unauthorized release and tampering. When the original work on the lattice security model was done at the MITRE Corporation [15] and at Case Western Reserve University [225], no one knew how to make formal statements about security policy. Indeed, protection against Trojan horses was considered an unsolvable problem at the time, and security researchers (including this author) were all quite surprised when the $*$–property made it possible to formalize Trojan horse protection. The later emphasis on protecting against unauthorized release was because no one knew how to protect against tampering but protection against unauthorized release was understood.

3.2.1 Biba Integrity Model

Biba [18] later developed a model of non-discretionary integrity that is a mathematical dual of the Bell and LaPadula non-discretionary-security model. Biba defines a set of *integrity access classes* that are analogous to security access classes and defines *simple-integrity* and *integrity-confinement properties* that are analogous to the simple-security and confinement properties. The difference between integrity and security is that the direction of the less-than signs are all reversed, so that a program of high integrity is prevented from reading or executing low integrity objects that could be the source of tampering or sabotage. The principal difficulty with the Biba integrity model is that it does not model any practical system. Unlike the security models that developed from existing military security systems, the Biba integrity model developed from a mathematical analysis of the security models.

3.2.2 Lipner Commercial Integrity Model

Lipner developed a commercial integrity model [144] that uses both the non-discretionary security and non-discretionary integrity models to represent a software development environment in a bank. It tied the integrity modeling much closer to reality than the Biba model did, but it was still quite complex. No effort has been made to actually implement the Lipner commercial integrity model.

3.2.3 Clark and Wilson Commercial Integrity Model

The most recent development in preventing tampering and sabotage is the Clark and Wilson commercial integrity model [40]. They have proposed a model of data integrity that they assert more accurately describes the needs of a commercial application than the Bell and LaPadula lattice security model [14]. Clark and Wilson's model focuses on two notions: *well-formed transactions* and *separation of duties*. Separation of duties is commonly used in commercial organizations to protect against fraud.[3] Clark and Wilson contrasted their work with Lipner's commercial security interpretation of the lattice security and integrity models [144] and concluded that Lipner's commercial model does not adequately deal with limiting data manipulation to specific programs to implement the well-formed transactions.

The Clark and Wilson model consists of a set of certification and enforcement rules to be applied to a computer system. Several entities in the system must be defined.

[3]Separation of duties is also a familiar concept to the military. Launch of nuclear weapons is done under a concept of *two-person control* in which no one individual can ever launch a nuclear weapon. Two separate individuals must turn separate keys simultaneously. The keys are placed such that it is physically impossible for one person to perform the necessary actions.

Data is stored in two classes of objects: *constrained data items (CDIs)* and *unconstrained data items (UDIs).* CDIs are the objects to be protected by the Clark and Wilson model. UDIs are conventional objects whose integrity is not assured by the model. Simple data input is a good example of UDIs.

Operations on CDIs are performed by two classes of procedures: *integrity verification procedures (IVPs)* and *transformation procedures (TPs).* The purpose of IVPs is to assure that all CDIs conform to some application-specific model of integrity and consistency. The purpose of the TPs is to change the set of CDIs from one consistent state to another. The TPs must implement the notion of *well-formed transactions.*

The certification and enforcement rules that follow are directly quoted from Clark and Wilson's paper [40]. (Clark and Wilson's paper gives a detailed justification for the rules.)

C1: All IVPs must properly ensure that all CDIs are in a valid state at the time the IVP is run.

C2: All TPs must be certified to be valid. That is, they must take a CDI to a valid final state, given that it is in a valid state to begin with. For each TP, and each set of CDIs that it may manipulate, the security officer must specify a "relation," which defines that execution. A relation is thus of the form: (TPi, (CDIa, CDIb, CDIc, ...)), where the list of CDIs defines a particular set of arguments for which the TP has been certified.

E1: The system must maintain the list of relations specified in rule **C2**, and must ensure that the only manipulation of any CDI is by a TP, where the TP is operating on the CDI as specified in some relation.

E2: The system must maintain a list of relations of the form: (UserID, TPi, (CDIa, CDIb, CDIc, ...)), which relates a user, a TP, and the data objects that TP may reference on behalf of that user. It must ensure that only executions described in one of the relations are performed.

C3: The list of relations in **E2** must be certified to meet the separation of duty requirement.

E3: The system must authenticate the identity of each user attempting to execute a TP.

C4: All TPs must be certified to write to an append-only CDI (the log) all information necessary to permit the nature of the operation to be reconstructed.

C5: Any TP that takes a UDI as an input value must be certified to perform only valid transformations, or else no transformations, for any possible value of the UDI. The transformation should take the input from a UDI to a CDI, or the UDI is rejected. Typically, this is an edit program.

E4: Only the agent permitted to certify entities may change the list of such entities associated with other entities: specifically, the list of TPs associated with a CDI and the list of users associated with a TP. An agent that can certify an entity may not have any execute rights with respect to that entity.

Section 10.1 proposes an implementation of Clark and Wilson's commercial security model using the SCAP architecture and shows how the restricted capability model combined with the lattice security model can aid in that implementation. It also discusses why Clark and Wilson's security model may present much more difficult problems than the relatively simple lattice security models. In the implementation, audit trails take a much more active role in security enforcement than in previous systems. In particular, access-control decisions are based on historical information retrieved from the audit trail, as well as on descriptive rules of who may have access to what.

3.3 Preventing Denial of Service

The most difficult problem of security enforcement is preventing denial-of-service attacks. This is because there is no good definition of what denial of service actually means. Furthermore, it can be argued informally that detecting and preventing a malicious denial-of-service attack may be equivalent to solving the Turing-machine halting problem.[4]

Various systems have been devised for allocating quotas and limiting resource expenditures in computer systems, but none of these have dealt with malicious denial-of-service attacks that might be implemented in the form of Trojan horses

[4]The argument is based on a very broad definition of denial-of-service attack that a Trojan horse could deny service by entering an infinite loop at a critical point. Since the Trojan horse could be hidden anywhere, detecting and preventing such a denial-of-service attack might be equivalent to the halting problem. Certainly, more restrictive definitions of the denial-of-service problem would not be equivalent to the halting problem.

or trap doors. While the integrity models, described in Chapter 10, could provide some assistance, denial of service remains a major unsolved problem in computer security. This dissertation does not address denial-of-service problems, except as they relate to security or integrity solutions.

Chapter 4

Principles of Capability Systems

This chapter provides a brief tutorial on capability systems and highlights their most attractive features. More complete tutorials can be found in Saltzer and Schroeder's [182] and Fabry's [67] papers and in the books by Levy [139] and Gehringer [76]. In this chapter and throughout the remainder of the dissertation, the focus is on capabilities within a single computer or a tightly-coupled multiprocessor. Networked capability systems are not addressed, except where explicitly mentioned.

4.1 What is a Capability?

A *capability* is both the name for an object and a ticket granting access to the object. It defines the type of the object, what access rights the holder may exercise, and in some systems, the object's location and size. Possession of the capability (at least in traditional capability systems) is both necessary and sufficient to gain access to the object. Furthermore, the holder of a capability may make copies of the capability and store them in the file system or pass them to others as subroutine parameters. When copying a capability, the holder may restrict the access rights or may shrink the size (from either end) of the object. A capability that either restricts the rights or shrinks the size (or both) is called a *refinement*.

It is essential that capabilities be protected against unauthorized modification or forgery. If a user could change the contents of a capability, then the user could gain arbitrary access to any object in the system. (The password-capability systems, described below, are an exception to this rule. They depend on probabilistic detection of modification, rather than outright prohibition.)

4.2 Capability Storage

Capabilities can be stored and used in many ways, depending on the system implementation. The simplest capability implementations use special capability registers. To use a capability, the programmer must explicitly load it into a capability register. The Plessey System 250 [48] is an example of a capability-register machine.

Capabilities can be stored in special capability segments and mapped automatically into the address space on use. Instead of capability registers, some kind of slave store or capability cache can automatically load the capabilities. Such a slave store makes management of a large, virtual-address space much simpler for the programmer. The capability cache is the direct analogue of the translation buffer in a conventional, virtual-memory machine. The CAP system [231] is a good example of an early capability machine with such a slave store.

Capabilities in both the Plessey System 250 and the CAP are stored in capability segments. To prevent unauthorized tampering, the capability segments are protected against modification. Only special capability loading, storing, and moving instructions are allowed to operate on capability segments.

Capability segments have the drawback that the capabilities must be kept separate from the rest of the programmer's data structures. Such separation could be inconvenient. Tagged capability machines solve this restriction by associating one or more *tag* bits with each location of memory. The tags identify the data type of the location. In the simplest case of a 1-bit tag, the type is either capability or data. In more complex machines, such as the Burroughs B6700 [167, 62], the tag could be several bits long and actually identify the type of the data as integer or floating point, etc. The IBM System/38 is an example of a tagged capability machine with a 1-bit tag for each 32 bits of memory. The Burroughs B6700 is not a capability machine, although it does use a 3-bit tag for each 48 bits of memory. Obviously, tagging can significantly increase memory usage. Gehringer [76, Section 6.2.2] has developed an alternative to tagged memory that he calls *typed memory*. Essentially, typed memory allows one to specify a type field for a larger block of memory than a single machine word, yet still maintain security.

Finally, there are new, distributed, capability systems that protect their capabilities with passwords. The capabilities may be stored anywhere in memory, and the user programs are allowed to arbitrarily modify them. Each capability includes a large (typically 64-bit) password. When a capability is presented for use, the contents of the capability and the password are checked against a master copy. If the password does not match or if the access rights or other, critical fields have been modified, then the operation is rejected. The Cambridge File Server [165, chapter 4] was the first to use password capabilities, and

Amoeba [162] and the Monash University password-capability system [7] have continued and extended the concept.

4.3 Need for Protected Subsystems

One of the principal reasons for having a capability-based system is to support *protected subsystems*. A protected subsystem is a piece of software that, together with its associated data, is encapsulated within a *protection domain* to prevent tampering. A protection domain is a separate and distinct address space in which the code of the protected subsystem may execute. The address space is defined by the union of the set of capabilities that make up the protected subsystem and the set of capabilities passed as arguments when the domain is called. The users of a protected subsystem are restricted to calling specified entry points and are prevented from direct access to the encapsulated data. The restrictions are implemented by granting only a so-called *enter capability* to the user of the protected subsystem. The enter capability defines the type of calls that may be made on the domain. An enter capability is invoked by executing a cross-domain call instruction, also called an ENTER instruction. The cross-domain call switches the address space of the CPU to that of the called domain, and transfers control to the address starting address specified by the enter capability. A caller of a domain cannot transfer to any address, except that specified in the enter capability.

Protected subsystems are useful for securing applications packages, such as electronic mail, database management, and transaction processing. Protected subsystem mechanisms enable the applications package to enforce some application-specific security policy that the operating system does not provide. For example, the electronic-mail system may wish to allow a user to write a message into another user's mailbox, but not to allow the sender to overwrite other messages that are already in the mailbox. A database management system may wish to enforce a security policy on individual records or fields of records.

An example of a database management system (DBMS), implemented as a protected subsystem, is shown in Figure 4.1. The user has an enter capability for the DBMS and can invoke the DBMS by executing a cross-domain call instruction, providing the enter capability as an argument. The cross-domain call transfers control into the DBMS, taking on the capabilities of the protected subsystem, while simultaneously discarding the capabilities of the caller. Thus, the caller and the DBMS protected subsystem are prevented from tampering with each other's data, meeting the requirements for mutual suspicion, defined by Schroeder [189].

An implementation of protected subsystems using just a protection-ring architecture [192] could not provide adequate support. Protection rings define a set of hierarchically ordered domains of protection, in which more privileged domains

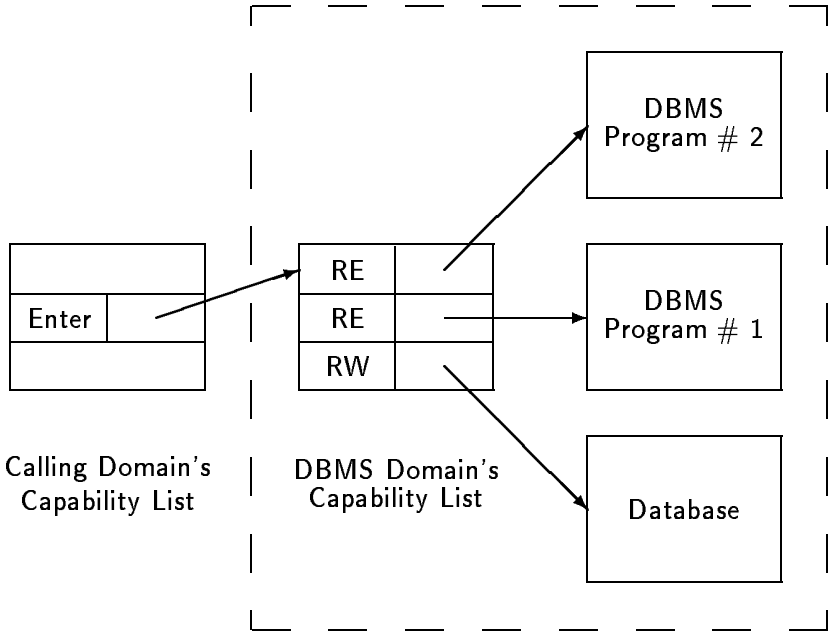

Figure 4.1: DBMS Protected Subsystem Example

take on all the rights of the less privileged domains. Rings are a generalization of the user/supervisor mode protections schemes, developed in the early 1960s. The strict hierarchical nature of rings forces one to make decisions as to whether, for example, an electronic-mail subsystem is more or less privileged than a database management subsystem. Since such subsystems are generally autonomous, a hierarchical ordering is not suitable. Furthermore, typical protection-ring machines have only a small number of protection rings. For example, the VAX architecture [138] supports only four protection rings; the Multics architecture [192] supports eight protection rings.[1] With only a small number of protection rings, some subsystems necessarily will become over-privileged. For example, the VAX DBMS product [214] runs in executive mode, because it must be protected from supervisor mode and user mode code. However, since executive mode in the VAX/VMS operating system has unlimited access rights within the operating system, the VAX DBMS product is over-privileged, and malfunctions in VAX DBMS could cause operating-system failures, rather than just DBMS failures.

Pure, access-control-list systems cannot be discounted in the support of protected subsystems. Data General has built a machine [52, 29] that supports protected subsystems using a pure, access-control-list approach that may have overcome some of the difficulties described above. The Data General scheme is based in part on Schroeder's work on mutually-suspicious subsystems. The sig-

[1]The original Multics design supported sixty-four rings in software, but the later hardware implementations only supported 8 rings.

nificant issue that must be resolved is how to represent a software package as a principal in an access control list. The little, published literature on the machine implies that they have worked on this problem beyond the simple proposal in Schroeder's dissertation [189, p. 149], but Data General's results are not made clear, as the machine has never been marketed.

Other, more restrictive protected-subsystem mechanisms have been built including the Titan Multiple-Access user-control system [73] and the UNIX setuid mechanism. Setuid is discussed in detail in Section H.2.

4.4 Type Managers and Sealing

Capabilities and protected subsystems can also be used to provide abstract data types and object-oriented programming by encapsulating abstract type managers within protected subsystems and passing *sealed* capabilities for abstract objects. A user can pass the sealed capabilities around, but cannot use them to examine the contents of the abstract objects. Only the type manager can *unseal* the capability and get access to the internal structure of the abstract object. This type of capability sealing was first proposed by Redell [179] and was first implemented in the CAP-III system by Herbert [95].

Sealing can be implemented by the addition of a type-id field to the capability. Each type manager is assigned a unique type-id. When a capability is sealed, the type-id of the requesting type manager is copied into the capability. Once sealed, the capability can only be copied or passed as a parameter. The unsealing operation is restricted to the type manager that possesses a type-id that matches the value stored in the capability. If the type-id values do not match, then the attempt to unseal would fail.

Part II

SCAP Architecture

Chapter 5

Overview of the SCAP Architecture

This chapter contains an overview of the SCAP secure capability architecture, including both hardware and software components. The description highlights the major features of SCAP and indicates which portions have been designed in detail, which have experimental implementations, and which have yet to be designed. Little rationale or supporting information is included here. Instead, each major feature is cross-referenced to the appropriate portion of the dissertation where the issues are covered in detail.

5.1 SCAP Processor Architecture

The SCAP processor architecture has three major variants. The preferred variant is based on reduced instruction set computer (RISC) design principles and makes no concessions to compatibility with existing systems. The second variant is based on abstract extensions to the VAX architecture and represents hardware on which SCAP could run and also preserve VAX-compatibility. The third variant is based on a VAX-11/730 CPU, with microcode changes but no hardware changes permitted. Only the VAX-11/730 variant has been implemented, as the other variants would have required extensive hardware work beyond the scope of this dissertation.

- SCAP on a RISC

  The RISC variant of SCAP uses a very simple processor with a pure load/store architecture, to minimize complexity both in the memory management hardware and in the page-fault handlers in the software. (See Section 13.2.) The processor has a large register file and dedicates portions of the register file as distinct register sets for different address spaces. (See Section 17.2.)

The processor has a large, paged, linear virtual-address space. (See Section 15.1.) The translation buffer supports address space numbers (ASNs), so that entries from more than one address space can be stored simultaneously. (See Section 15.2.3.) Furthermore, translation-buffer misses are handled in software, rather than by the processor hardware or microcode. (See Section 15.3.)

- SCAP on a Modified VAX Processor

  The preferred VAX processor for SCAP is similar to the RISC processor in some respects. The principal difference is that the VAX processor would support the VAX instruction set rather than a simple load/store architecture. Supporting the VAX instruction set would permit software compatibility with existing VAX operating systems, as discussed in Appendix H.

  The memory management of the preferred VAX processor for SCAP is very similar to the RISC variant, described above. The process has a 32-bit linear address space, and a translation buffer that supports ASNs and software handling of translation buffer misses. Such a memory-management unit is quite different from that of any existing VAX processor.

  The preferred VAX processor for SCAP also supports special microcode to optimize the performance of cross-domain calls and returns, as described in Section 17.7.2.

- SCAP on a Re-Microprogrammed VAX-11/730

  A VAX-11/730 with microcode changes only cannot support all of the features of the preferred VAX-architecture for SCAP. Specifically, the translation buffer hardware cannot support ASNs, and translation buffer misses are most easily handled in microcode, rather than software. The modified microcode does implement the hashed page tables that would otherwise be implemented in software. (See Section 16.6.) The modified microcode also supports the optimized cross-domain calls and returns, described in Section 17.7.2.

5.2 SCAP Operating System

Because the SCAP processor hardware was kept deliberately simple, most of the capability and security features are implemented in the SCAP operating system. Except where otherwise indicated, the design of the features is shown in this dissertation, but full implementation has not been done, because the work would be well beyond the scope of a single Ph.D. dissertation.

- The SCAP operating system is capability-based, with capabilities handled entirely in software. To achieve adequate performance, however, the effective access rights from capabilities for primary-memory segments are recorded in page tables and enforced by the translation-buffer hardware of the processor. (See Sections 14.2 and 15.1.) Abstract data types are implemented with seal and unseal operations, similarly to CAP-III. (See Section 4.4.)

- Secure capabilities are the central feature of SCAP. A secure capability, as defined in Chapter 7, is designed to solve the confinement problem. Secure capabilities, as distinct from conventional capabilities, are necessary to gain access to an object, but are not sufficient in themselves. Before a secure capability can be used, the SCAP operating system also checks the non-discretionary access controls and the access control list associated with the object in question. With secure capabilities, the SCAP operating system can enforce both the Bell and LaPadula security model (defined in Section 3.1.2) and the Clark and Wilson commercial-integrity model. (See Section 3.2.3 and Chapter 10.) Furthermore, the SCAP operating system can meet traceability-of-access requirements using the access control lists described in Chapter 8.

- The SCAP model of domains and processes is designed to support tightly-coupled multiprocessors, with large numbers of CPUs. SCAP processes are light-weight processes, so that it should be relatively easy to distribute a computation across many processors. (See Chapter 6.)

- The SCAP operating system will be implemented as a layered security kernel. Each layer will reside in a separate domain of protection to limit error propagation and aid in formal verification. Appendix G contains a preliminary sketch of the lower layers of the SCAP security kernel. Detailed design of the kernel is beyond the scope of this dissertation.

- SCAP protection domains will primarily be used to implement major layered products, such as database-management systems or electronic-mail systems. Protection domains will only be used to enforce security requirements. Compilers will be expected to enforce data-type safety by generating proper code, rather than by relying on operating system or hardware constructs. Chapter 14 discusses why the use of protection domains will be limited and what impact excessive programming generality has on overall system performance.

- SCAP provides immediate revocation of access rights on all objects using one of two new revocation algorithms. If implemented on a processor with

shared page tables, such as Multics, SCAP uses revocation with event-counts, as described in Section 11.3. All of the preferred SCAP processors, described above in Section 5.1, use unshared page tables. For these processors SCAP uses revocation by chaining, as described in Sections 11.4 and 16.5.

- The SCAP operating system does not support a system-wide garbage collector, because of the storage channels inherent in such a scheme. Instead, it supports storage quotas, combined with a rent collection mechanism to provide automatic, storage-channel-free management of resources. (See Chapter 12.)

- To deal with discretionary-Trojan-horse attacks, the SCAP command-language interpreter incorporates a name-checking protected subsystem. The name-checking subsystem ensures that programs do not reference or modify objects that are not defined in advance in a database of expected references. Potential security violations are referred to a human being for ultimate resolution. (See Chapter 9.)

- The SCAP architecture includes several microcode and software techniques to optimize the performance of cross-domain calls, described in Chapter 17. An experimental implementation, described in Chapter 18, shows that the SCAP optimizations lead to cross-domain calls that, when compared to simple instructions, are significantly faster than in previous capability-based systems.

- The SCAP architecture provides a mechanism for fast handling of hardware interrupts. As described in Chapter 19, interrupts are initially handled in a system-wide trusted domain that is part of the security kernel. SCAP, however, provides a mechanism so that the interrupt handler can make cross-domain calls at elevated interrupt priority levels (IPLs). This mechanism is intended to improve response times to hardware interrupts, while still using small domains of protection to isolate portions of the security kernel.

- The SCAP architecture, when implemented on a VAX processor, can provide some levels of software compatibility with both the VAX/VMS and ULTRIX-32 operating systems. Appendix H contains sketches of how such compatibility might be achieved.

Chapter 6

SCAP Domain Model

This chapter defines the basic notions of *processes*, *domains*, and *protected subsystems* as they will be used in SCAP. These definitions are different from those used in the literature, and the differences are highlighted. The definitions are based on the assumption that the underlying configuration could be a single processor in either a workstation or a large time-shared system, or a multiprocessor implementation with at least some memory shared among processors. The chapter also discusses performance optimizations in light of the duality of message-oriented and procedure-oriented systems which was proposed by Lauer and Needham [135].

6.1 Scheduling Entities

Two classes of scheduling entities are defined: *jobs* and *processes*. Each has its own protection implications, as outlined below.

6.1.1 Jobs

When a user logs into the system, a *job* is created that will exist until the user ultimately logs out. A job may contain many points of execution, but each of the points of execution is operating on behalf of the particular human being, called the *principal*, who owns the job. Jobs may be initiated either from interactive terminals or from batch queues. (There may also be network-initiated jobs and jobs that function on behalf of the system itself.) Jobs are intended to be similar to VAX/VMS jobs or CAP sessions.

6.1.2 Processes

A *process* is an execution point with an associated set of machine registers and a cross-domain call stack, called the *C-stack*. The C-stack is the activation record for cross-domain calls. It behaves much like an ordinary subroutine-call stack,

but is protected against access from any of the domains. Each C-stack frame contains any registers that may have been saved and any capability arguments that have been passed as part of the current cross-domain call. The CAP-I C-stack is described in [231, p. 11], and Section 17.7.2 shows the detailed design of the C-stack for a microcoded implementation of SCAP.

Every job must have at least one process. SCAP processes are very different from the processes in other operating systems, such as the VAX/VMS or UNIX operating systems, because SCAP processes do not have a single associated address space. At any particular point in time, a SCAP process must be executing in some address space, called a *domain* (defined in Section 6.2.2). Many SCAP processes may share a single address space, and there may be address spaces with no processes currently executing in them at all. Unlike processes in VAX/VMS, SCAP processes do not carry fixed discretionary access rights. The discretionary access rights of a SCAP process can vary as execution moves from one domain to another. However, SCAP processes do carry non-discretionary access rights, as discussed in Section 6.6.

SCAP processes are similar to the light-weight processes that are found in systems such as Mach [2]. SCAP processes do not have address spaces, so the costs of creating and scheduling them should be significantly lower than that of VAX/VMS or UNIX processes.[1]

6.2 Address Space Entities

SCAP supports two types of address space entities: *protected subsystems* and *domains*.

6.2.1 Protected Subsystems

A *protected subsystem* is a way of encapsulating a set of objects and ensuring that those objects can be manipulated only by particular programs. A protected subsystem is a collection of programs and capabilities for protected objects. A protected subsystem is not an entity that can execute on a CPU. Examples of

[1]SCAP processes are potentially still more expensive than one might need, because every SCAP process has a C-stack for making cross-domain calls. If one is writing an application for a system with a large number of processors, even the cost of the C-stacks could prove excessive. Therefore, SCAP could also support an even less expensive scheduling construct, called a *thread*. A thread is simply a process with no C-stack. As a result, a thread would be barred from making cross-domain calls or even calls on the security kernel itself. A thread communicates with other threads or processes in the same domain through shared memory constructs. The concept of thread is completely optional to the rest of the SCAP architecture, and its implementation is *not* recommended. Threads are included solely as a possible performance optimization if the cost of C-stacks proves too high. Depending on the precise implementation of the security kernel, certain kernel calls or scheduling primitives may have to be provided by some ad hoc mechanism, for use even without the presence of a C-stack.

protected subsystems are a database and its associated manager or an electronic-mail system. The CAP-I counterpart of a SCAP protected subsystem is a *procedure control block* [91, Section 3.2.3].

6.2.2 Domains

A *domain* is an invocation of a protected subsystem. Figure 4.1 on page 44 in Chapter 4 showed a domain invocation of a typical protected subsystem. A SCAP process is like the execution point of a VAX/VMS process, and a SCAP domain is like the address space of a VAX/VMS process. There may be several, distinct domains simultaneously invoked from a single protected subsystem. The code of the programs that run in those domains must synchronize with the other invocations, using locking primitives and shared memory. The actual techniques for invocation are discussed in Sections 6.4 and 6.7.4.

Several different jobs may wish to use the services of a particular protected subsystem. Depending on the application's design, there might be a single domain serving all requests, or each job (or process) might create its own domain to provide the services.

All domains invoked from a particular protected subsystem share a common set of protected objects. Different domains might have different access rights to those objects, but the objects remain shared. Thus, a particular database would be encapsulated in a particular protected subsystem. Various domain invocations from that protected subsystem would receive capabilities with varying access rights to that database. A different database would be encapsulated in a different protected subsystem. However, both protected subsystems might share the code of the common database manager programs.

6.3 A Simple Example

Figure 6.1 contains a simple example of a job with several processes, protected subsystems, and domains. User pak has a job active in the system, and that job has two processes. Two domains have been activated from protected subsystems A and B shown in the file system on disk. Note that process 1 has started executing in domain pak.A, and then issued a cross-domain call to domain pak.B. Process 2 is shown executing simultaneously in domain pak.B. While executing in pak.B, both processes share the same address space, so the code of the domain must include synchronization to prevent conflicts on shared variables. (Of course, a different example could have shown two distinct invocations of B with distinct address spaces, both belonging to the job owned by pak. Section 6.7.4 shows how to create distinct invocations.)

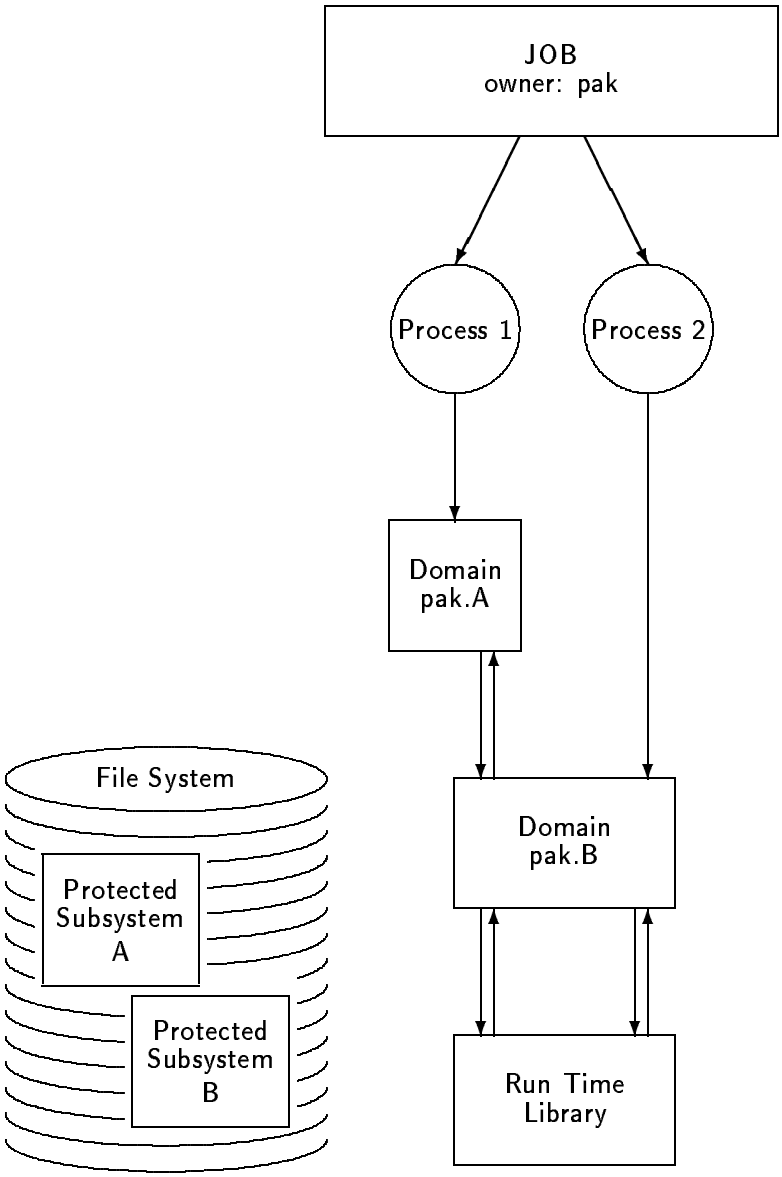

Figure 6.1: Protected Subsystems, Domains, and Processes

The figure also shows that domain pak.B makes calls on a run-time-library domain. Other domains, such as pak.A, would probably make similar calls, but those calls are omitted for clarity.

6.4 Cross-Domain Calls

A process transfers control from one domain to another by making a cross-domain call, possibly passing capabilities as arguments. Cross-domain calls are quite similar to remote procedure calls [21] and could easily be extended to transfer control to other domains running on other processors in a network. The right to make a cross-domain call is determined by the possession of an enter capability for the domain in question. Access to the capability will be constrained, of course, as defined in Chapter 7.

Arguments are passed in a cross-domain call in one of two ways. A small number of small arguments may be passed by value in the machine registers. For arguments that cannot fit in the registers or that must be passed by reference, capabilities may be passed as arguments on the C-stack. The receiving domain will then map those capabilities into its address space.

When a capability for a memory segment is passed to a new domain, that memory segment must be mapped into the called domain's address space. By contrast, capabilities for abstract data types need not be mapped and can therefore be passed at lower cost. Register parameters can be passed at still lower cost, as no security checking is required. As discussed in Chapter 17, the vast majority of cross-domain-call parameters can be passed in registers, so the protocol will be optimized for register parameters only. Thus, the cost of capability parameters will be paid only for those calls that require them.

When a process enters a domain, it takes on the entire address space of that domain, including any argument capabilities that may have been passed to the domain by simultaneous cross-domain calls from other processes. Similarly, if a process maps in some argument capabilities, then other processes that are simultaneously executing in that domain will gain access to those arguments. Of course, two processes executing in a single domain will have to synchronize their references to shared objects, including the argument capabilities. Section 6.7.4 shows how two processes can execute in the same protected subsystem, but in different domain invocations and therefore different address spaces.

6.5 Comparison with Reed's Scheduler

It is useful to compare SCAP processes with the virtual processors of Reed's two-level scheduler [180], because Reed's scheduler design has been used in other security kernel implementations, such as Schell's multiprocessor kernel [187].

Reed's scheduler defines two kinds of virtual processors: *level one virtual processors (vp1s)* that provide primitive processes for use in the kernel and *level two virtual processors (vp2s)* that are user-level processes with address spaces. Reed implements the scheduler as two abstract type managers: the lower level scheduler that binds vp1s to actual CPUs and the higher level scheduler that binds vp2s to vp1s. In this way, Reed decouples the issues of micro-scheduling to keep physical CPUs busy, from macro-scheduling to implement swapping policies.

The SCAP scheduler entities have loose analogues in Reed's model. A SCAP process is somewhat like Reed's level one virtual processor (vp1), in that it is a primitive execution point without an address space. However, SCAP processes are user visible, unlike Reed's vp1s. A SCAP domain is somewhat like Reed's level two virtual processor (vp2) in that it is an address space and SCAP processes can enter or leave it. Thus, a cross-domain call is analogous to the operation of binding a vp2 and a vp1. Many different SCAP processes may be present in a single domain simultaneously, while Reed makes the vp2–vp1 binding one-to-one. The essence of the difference is that Reed's design reserved the low-cost processes for use by only the kernel, whereas the SCAP design allows low-cost processes to be used by untrusted user code as well.[2]

6.6 Non-Discretionary Controls for Processes and Domains

Although different domains called by the same process may have different access rights, all domains called by a process must have the same non-discretionary access class. This is because the cross-domain call constitutes a write operation from the calling domain to the called domain, and the cross-domain return constitutes a write operation from the called domain back to the calling domain. Because the domains can each write information to the other, the Bell and LaPadula security model [14] requires them to be at the same non-discretionary access class. Therefore, each process and each domain will have a non-discretionary access class assigned to it, and a process will only be able to call domains that are at a matching access class.

Although a domain must be at a fixed access class, different domains invoked from the same protected subsystem may be at different access classes. Of course, the non-discretionary access rights of those domains would be different, even though they might inherit the same capabilities from the protected subsystem.

There is also no requirement that all processes within a single job be at the same access class. A user could have processes at many different access classes

[2]For completeness, note that SCAP threads are analogous to vp1s permanently bound to a single vp2.

operating simultaneously on his or her behalf during a single login session. When the user first logs in, the system would create a secure-server process as the first process in the job. From the secure-server process (which runs trusted code, considered part of the security kernel), the user could then control multiple processes at multiple access classes, switching among them at will.

Figure 6.2 shows how jobs, processes, protected subsystems, and domains interact in the presence of non-discretionary controls. A user named pak has a job running on the system. That job has a secure-server process that is executing trusted code in a secure-server domain. From the secure server, pak has created four processes. Processes 1 and 2 are running at the non-discretionary access-class Public; processes 3 and 4 are running at the access-class Secret. The file system contains two protected subsystems, A and B, that consist of program code, data files, and capabilities for data. In the figure, each of the protected subsystems has been activated twice, once at each of the two access classes. Process 1 is executing in the domain pak.A at access class Public, and issues cross-domain calls to domain pak.B, also public. Process 2, by contrast, started execution in domain pak.B, so the code of protected subsystem B must implement appropriate mutual-exclusion algorithms to synchronize access to its data. Domain pak.B can be seen making cross-domain calls to a domain containing a run time library.

Processes 3 and 4 are running at access class Secret, but they also wish to invoke the protected subsystems A and B. Therefore, they get distinct domains with different access rights to data stored within those domains, all in accordance with the non-discretionary security model. The figure also shows an invocation of the run-time library in a domain at access class Secret. Although not shown in the figure, other users would have their own jobs and processes and, ordinarily, their own instances of the domains. It is also possible to set up a job and its domains as a server to handle cross-domain calls from processes belonging to many different jobs. Such a server might implement a line-printer service or an electronic-mail queuing facility. Servers of this kind would require verification, if they handled material from different non-discretionary access classes.

6.7 Creation and Initialization

This section describes how jobs, processes, protected subsystems, and domains are created and initialized.

6.7.1 Jobs

Jobs are created in one of two ways. If the user logs into the system from a local terminal or from a network connection, a secure server process will be started, as described above in Section 6.6 and shown in Figure 6.2.

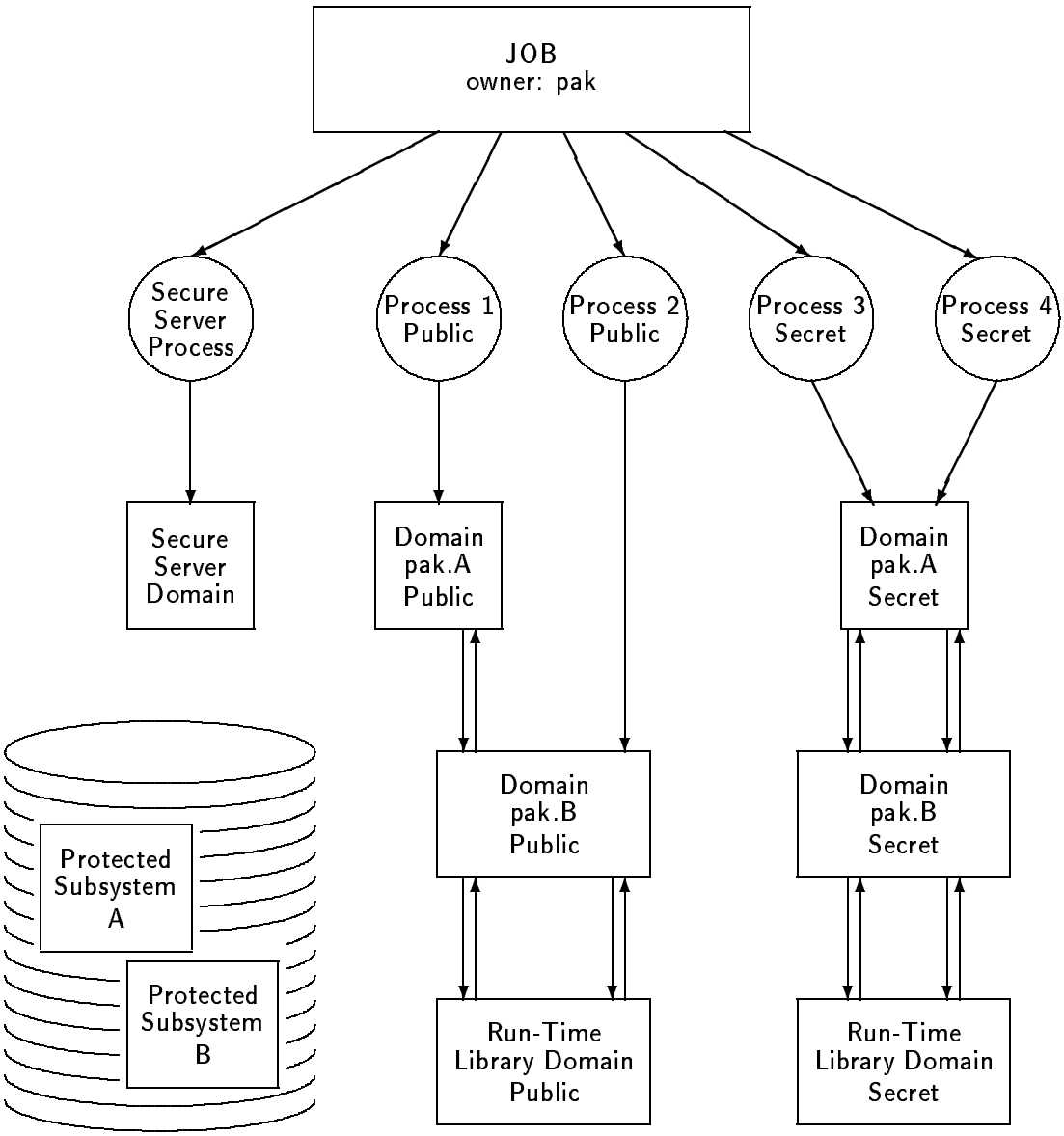

Figure 6.2: Domains and Non-Discretionary Controls

Jobs can also be created from the batch queues. A batch job does not need a secure-server process, since there is, by definition, no human being with whom to communicate. Therefore, a batch job can be started with a process running in a command-language-interpreter domain at the access class at which the job was submitted.

6.7.2 Processes

Processes are created using a mechanism similar to the *fork*(2) system call [213, p. 2-45] of the UNIX operating system [211]. Unlike *fork*(2), the SCAP process creation call does not copy the address space of the parent process. Instead, both the parent and child processes find themselves running in the same, shared address space. The only difference between the parent and child is that the parent process will have the same C-stack as it had immediately prior to the process-creation call, while the child process will have a new C-stack with no prior domains to which the child could return. The new C-stack will be allocated from space reserved by the security kernel specifically for C-stacks. Each job will have a quota of processes that can be created at any access class, to prevent malicious process forking either from running the kernel out of C-stack space or from using C-stack allocation as a storage channel. The process-creation call should always be encapsulated in a language run-time-library routine that allocates a new local-variable stack for the child process. The child process, of course, could be scheduled to run on a different processor in a tightly-coupled multiprocessor system.[3]

Once created, the child process runs in the same domain as the parent and at the same non-discretionary access class. The secure-server domain, as part of the security kernel, will have the privilege to create processes at different access classes, so that job creation can be performed as described in Section 6.7.1.[4]

6.7.3 Protected Subsystems

Protected subsystems are static entities that exist in the file system. They contain programs, data, and capabilities for the stored data. Once a protected subsystem has been created by the trusted linker, a user's program can only copy, move, or delete it in the file system, or invoke it with a protected-subsystem

[3]The process created has been described as a child of the creating process. There is no inherent security reason, however, to force processes into a hierarchy. All processes of a job could be treated as co-equal, or they could be treated as a hierarchy, as is done in UNIX. I have left that choice to the ultimate implementors of a SCAP security kernel.

[4]For completeness, thread creation would be exactly the same as process creation, except that the child thread would not have a C-stack. The C-stack pointer register of the child thread would be set to an invalid value, so that any attempt to perform a cross-domain call or return would fail. The C-stack is described in detail in Chapter 17.

enter capability. Invoking a protected subsystem is actually the act of creating a domain, and is discussed in Section 6.7.4.

6.7.4 Domains

A domain is created by executing a cross-domain call with a protected-subsystem enter capability. An address space for the domain is created and the domain is started at its initialization entry point.[5] The domain invocation is at the access class of the process that executed the cross-domain call on the protected-subsystem enter capability. In a similar way to the CAP-I LINKER [91, Section 3.2.3], the SCAP operating system replaces the protected-subsystem enter capability with a domain enter capability at the time of the first call.[6] The replacement occurs in the capability list of the calling domain, so that future calls will execute more rapidly. This provides transparent dynamic linking to the majority of programs that merely wish to invoke a single domain as an instance of a single protected subsystem.

Just as in Birrell's design for cross-domain calls for CAP-I [20, Section 3.1], the first cross-domain call into a domain executes both language-provided and user-written initialization code. Thereafter, any further cross-domain calls into the domain will be implemented as co-routines.

Calls with domain enter capabilities are much more efficient than those with protected-subsystem enter capabilities, because the work of address-space creation and initialization is done only once. Domain enter capabilities can be copied and used by many different processes, as long as the code for the domain is properly synchronized. This is important to make effective use of tightly-coupled multiprocessor systems. While domain enter capabilities can be preserved indefinitely, the domains to which they point may be deactivated. Such a deactivation would be equivalent to a revocation of the domain enter capability, and the caller would have to again use the protected-subsystem enter capability to create a new invocation.

A small number of callers may wish to activate multiple domains from a single protected subsystem. Those callers should preserve a copy of the protected-subsystem enter capability elsewhere, and then perform another cross-domain call on the protected subsystem capability, rather than on the newly created domain enter capability.

The secure-server domain will have the privilege to invoke domains at any non-discretionary access class, so that job creation can be performed. Specifically,

[5] A domain may have multiple entry points, but each entry point will have its own enter capability.

[6] Thus, SCAP protected-subsystem enter capabilities and domain enter capabilities are somewhat analogous to CAP-I outform and inform enter capabilities, respectively. However, SCAP allows multiple processes to execute in the same domain, simultaneously.

as described in Section 6.7.1, a command-interpreter domain will be required at the access class of each process that the user may start. Those domains, of course, would share any read-only code, but would have distinct stacks and local data areas to prevent any information leakage. Note that the command-interpreter domains are not trusted to enforce non-discretionary controls.[7]

6.8 Message Passing and Procedure Calls

Lauer and Needham [135] make a strong case that message-oriented systems and procedure-oriented systems are equivalent, that programs in either system will closely resemble each other, and that the performance of comparable programs, measured by queue lengths, waiting times, and service rates, etc. will be identical. This section examines the duality of message passing and procedure calls and argues that in the case of a cross-domain call that does not require locking, it is easier to optimize performance with the procedure-call model than with the message-passing model.

Lauer and Needham specifically compare programs that synchronize by message passing and programs that synchronize with *monitors* from the Mesa programming language [156]. They note that simple textual changes can automatically convert a message-passing program to a monitor program or vice versa. Based on this concept of duality, Herbert's CAP-III operating system [93, 95] combines domains and processes into a single abstraction and implements the cross-domain call as a message sending and receiving function. Mach [2] likewise provides only message passing as a way to implement cross-domain calls. Watson's CAP2 system [226] took advantage of the duality in a quite different form. Watson allowed a domain to be called, either as a procedure or as a process. Both types of calls passed arguments in a capability segment, like the CAP-I ENTER instruction [231], but the process form only allowed a single call to be outstanding at a time. Thus, Watson's CAP2 lacked the message-queuing facilities of Herbert's CAP-III. By contrast, SCAP allows multiple processes to execute in a domain simultaneously, thus avoiding the CAP2 restriction.

Lauer and Needham made a strong argument that the performance of a message-passing system should be the same as the corresponding procedure-oriented system. If one carefully examines the performance of cross-domain calls, one finds that the predicted duality of performance is not always present. Specifically, Lauer and Needham compared Mesa monitors with a message-passing system, and Mesa monitors include a locking operation, complete with a queuing mechanism to wait for the lock. A simple cross-domain call has no such lock and

[7]To implement discretionary Trojan-horse protection, as described in Chapter 9, the command-interpreter domain would require some trust, but would still be subject to non-discretionary controls.

queuing mechanism and requires significantly fewer machine cycles to implement. If all cross-domain calls required a lock and queuing mechanism, then the duality of performance argument would hold, but in fact, there are many, useful cross-domain calls that require no such locks. Should such calls be executed very frequently, as could easily occur in the implementation of a domain-structured operating system, then the overhead could become prohibitive.[8]

There is a partial dual of the non-locking cross-domain call in the message passing system of the GEC 41XX series of machines [75]. The GEC 41XX supports a microprogrammed message-passing system that supports not only a queued-message system, but also a fixed-message system. The fixed-message system allocates precisely one message buffer for a particular process-to-process channel and does no locking. Thus, fixed-message passing gains some of the performance advantages of a non-locking cross-domain call. Even when passing a fixed message, the scheduler code must still run to check for higher priority messages. A cross-domain call would not invoke the scheduler and would therefore execute more quickly. If the fixed-message-passing mechanism transferred control directly to the target process, then it would be an exact dual of the non-locking cross-domain call. There are hazards in using fixed-message passing, because there is nothing to prevent the sending process from overwriting a message with a second message, if the receiver has not yet been scheduled. As a result, GEC [75, Section 3.4.3] recommends against using fixed-message passing for most applications.

The fact that additional performance gains are possible with non-locking cross-domain calls in no way diminishes the results of Lauer and Needham. Their primary point was that a procedure-based system, using monitors, could achieve the same results as a message-passing system. With that result, it was then possible to develop the concept of remote procedure calls [21]. My point is that a procedure-based system not only will achieve the same results as a message-passing system, but will also perform better.

[8]The argument in this section is based on the author's personal experiences in implementing a layered security kernel, described in [145]. The details of that kernel design have not been published, but the kernel included a layer that could be treated as a domain that required no locking. The layer in question was on a critical performance path and locking or message passing would have been prohibitively expensive.

Part III

Improving Security

Chapter 7

Solving the Confinement Problem

A major problem for capability systems is a fundamental inability to solve the confinement problem, defined in Section 3.1.1. The problem arises, because the basic definition of a capability says that possession of a capability is both necessary and sufficient to gain access to an object. Furthermore, capabilities can be passed from one protection domain to another, either as arguments, or through storage in shared objects. By contrast, an access-control-list system requires that the access control list be checked prior to granting access to the object, and the addition of a non-discretionary check is quite natural.

A simple example of the problem is a protected subsystem containing a Trojan horse. Surreptitiously, the Trojan horse can obtain a capability granting write permission to a low-access-class object. How could the Trojan horse get such a capability? Since capabilities can be stored anywhere in the system, there are various possibilities. The capability might have been stored as part of the protected subsystem, or it might have been stored in some object that has public read permission. Storing a write-capability for a low-access-class object in a place that is readable by anyone is not a violation of security. When the protected subsystem is passed a capability to a high-access-class object, the Trojan horse copies the high-access-class data into the low-access-class object, to be later retrieved by an agent who lacks authorization for high-access-class data.

This chapter[1] first examines attempts to support the lattice model with traditional capability systems. It then introduces the new, secure capability architecture (SCAP) that solves the confinement problem in a very clean and simple way. Finally, the chapter contrasts the SCAP architecture with other, similar, modified capability architectures. The SCAP architecture is the driving force behind most of the ideas in this dissertation. It will appear again, most sig-

[1]This chapter is based, in part, on a paper [119] presented at the 1984 IEEE Symposium on Security and Privacy.

nificantly in the upcoming chapters on traceability of access (Chapter 8) and revocation (Chapter 11).

7.1 Attempts with Traditional Capabilities

Incorporating the lattice security model into an access-control-list system, such as Multics [168], is relatively straightforward, because all protection rights are associated with objects, and because rights cannot pass from subject to subject without operating-system intervention. When the operating system intervenes, the lattice-security rules can be enforced. By contrast, capability-based systems normally allow unrestricted flow of access rights from subject to subject as part of procedure calls. Two traditional capability systems, HYDRA and PSOS, have proposed support for the lattice model. Each of these approaches has drawbacks (discussed below) that are overcome in the new, secure-capability scheme.

7.1.1 HYDRA

HYDRA [238] was a capability-based system developed at Carnegie-Mellon University to study both capability systems and tightly-coupled multiprocessors. HYDRA attempted to solve the confinement problem by introducing a special right called *unconfined rights* or *UncfRts* to limit the propagation of capabilities. A domain in HYDRA is confined if it is called without UncfRts. Such a confined domain loses the right to store capabilities or information into any inherited (and potentially shared) objects. The confined domain may create new objects, but cannot share those new objects with any other domains. The confined domain is limited to modifying the arguments with which it was called.

Unfortunately, the HYDRA solution to the confinement problem is both overly restrictive and not well-suited to protected subsystems outside the security kernel. In HYDRA, a confined domain loses the right to store into any inherited objects, even into objects where the lattice security model would have permitted storage. As a result, software that was written for an unconfined environment will probably not work when run in a confined environment, even when all operations are at a single access class and no security violations in fact take place. Furthermore, these restrictions on storing mean that system programs such as the filing system, command-language interpreters, compilers, and editors cannot be confined. For example, an editor might maintain a checkpoint facility using inherited objects. Such an inherited checkpoint file would be banned in a confined domain. Since formal verification of software is very difficult in practice, most software a user runs (including these utilities) must be untrusted and run in a confined environment, so that only a small security kernel remains to be verified. Under these assumptions, the HYDRA solution to the confinement problem becomes unworkable.

7.1.2 PSOS

PSOS, the Provably Secure Operating System, is a design for a capability-based system with proven strong security properties. The basic PSOS design does not support the lattice model. Instead, PSOS contains an optional *secure object manager* layered above the basic elements. The secure object manager implements objects called *secure documents*, and it enforces confinement by ensuring that capabilities with write permission are never stored into secure documents. Thus, improper propagation of write permissions in violation of the confinement property is barred. PSOS prevents improper propagation by a complex set of rights that separately determines whether a write capability can be stored.

PSOS has never been implemented and remains only a design concept.[2] The design as presented in [166] appears to be overly restrictive about capability propagation and could have difficulties when a user wished to transport software from a PSOS that did not use the secure object manager to one that did. In particular, such software might make explicit calls on modules that were concealed by the secure object manager. In addition, such software could not function with the secure object manager if it stored write capabilities into objects. Even if the software attempted no security violations, the secure object manager would prevent the storing of write capabilities and thus cause the software to malfunction.

7.2 The Secure Capability Architecture

SCAP contains a table of descriptions of objects, and capabilities refer to entries in this table. The exact implementation of this table is not important here, save to say that all capabilities for a given object will point at the same entry. To be practical, a capability system must cache the results of evaluating capabilities in a hardware lookup table, so that efficiency shall not suffer greatly. The table is structured so that it is possible to flush all entries in the cache derived from a given entry in the central object table.[3] Further, the cache provides an associative search or direct lookup so that the overhead of capability evaluation is not unduly expensive. To avoid the obvious storage channels, the central object table is only be inspected by the security kernel, and the indices into the central object table (that must be stored in capabilities) are not be visible to the user. Chapter 15 discusses specific implementations of capability caches with emphasis on their performance implications.

[2] The Honeywell Secure Ada Target described in Section 7.4.4 evolved from the PSOS design.

[3] Flushing of this sort is available in the *capability unit*, i.e., capability cache, of the Cambridge CAP computer [231]. Section 15.2 discusses SCAP's management of the capability cache in much greater detail.

7.3 Non-Discretionary Security

Figure 7.1 illustrates the mechanisms of SCAP with an example of a user U invoking a missile-trajectory-analysis subsystem M from a process U. U will have logged into the system at some authorized access class A (e.g. unclassified, confidential, secret, top secret) and owns a file called D that contains information at this level about trajectories to be analyzed.

Subsystem M contains two embedded capabilities for files Y and Z. Files Y and Z are both classified at access class Low, where Low is less than access class A. The embedded capabilities grant read and read/write permission to the two files, respectively, but the non-discretionary rules would be violated, if M were allowed to write into file Z.

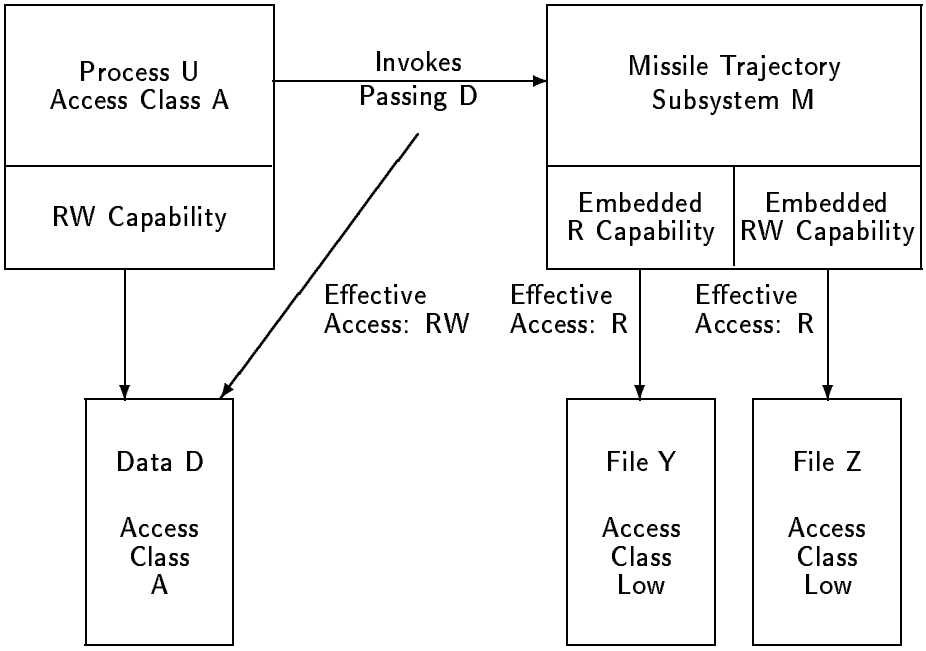

Figure 7.1: Non-discretionary Limited Capabilities

The problem being addressed is that of non-discretionary security: to prevent the flow of material in one access class to a lower access class. The Bell and LaPadula requirements state that a process may read information from an object of lower or equal access class and may write information to an object of equal or greater access class. In the example in Figure 7.1, the process acting for U will be labelled as running in access class A. Therefore a capability granting read access to an object must only be successfully evaluated if the object is marked as belonging to access class A or lower. Capabilities granting write access must only be successfully evaluated if the object is marked as belonging to access class A or higher. Initially, let us assume that the access class of the object and the

access class of the process are directly available to the capability unit of the CPU. Enforcement of the Bell and LaPadula rules then becomes trivial. There are two major drawbacks to making such information directly available to the capability unit. First, it needlessly adds complexity to the capability unit. The capability unit would need hardware logic to compare sensitivity levels and category-set bit-masks. Not only would such logic be very complex, but it would be on one of the most critical performance paths in the entire CPU. Second, it restricts the CPU to supporting only the Bell and LaPadula lattice model, and not some other, incompatible security model that might be devised in the future.

The caching property of the system will allow the lattice model to be implemented more effectively. If the processor does not automatically load capabilities into the cache, but instead causes a fault or trap to the most privileged software domain (called the security-kernel domain), then the security-kernel domain can evaluate whether the lattice model permits the capability to be used. The CPU hardware and microcode need not understand the lattice model. They need only ensure that capabilities are loaded into the cache only upon command of the security-kernel domain.

How would a protected subsystem operate under the lattice model? Suppose that all writable objects in M belonged to the highest access class and all readable objects to the lowest class. In that case, M could be run by any process at any access class, although no objects in M would be simultaneously readable and writable. In practice, the objects of M will not be so simply classified, and any reduction in classification of writable objects, or increase in classification of readable objects reduces the access classes at which M can be run. For this reason, a subsystem will have some *a priori* set of access classes in which it can run without hindrance. To this purpose, it should be possible for a program to inspect the access class in which it is run so that it can elect to retire gracefully, rather than being forcibly prevented from accessing objects outside its domain of protection.

This leaves some questions about data preserved between runs of the subsystem, such as accounting records. At what access class should they be classified? Either they should be classified in the highest access class to enable *writing up*, or else the domain should have a separate account file classified at each access class and the subsystem must choose on the basis of the access class of the caller. In this respect, the restriction of use of objects by a domain should be applied dynamically, so that a domain may contain embedded capabilities for objects inaccessible in the present access class and may continue running, provided that no attempt is made to use these capabilities. Figure 7.1 shows that subsystem M contains two embedded capabilities, one for a read-only file Y at access class Low (Low $<$ A) and one for a read-write file Z at access class Low. However, the augmented capability architecture will ensure that if M is operating on behalf of U at access class A, M's effective access to Z will be restricted to read-only.

Thus, even if M contains a Trojan horse, M will be unable to violate the Bell and LaPadula security model and compromise the data D.

To implement non-discretionary security then, it is necessary to label every process with the access class of the user logged in to it and to label each object with the access class to which it belongs. It is then possible to implement the Bell and LaPadula *read down* and *write up* rules to prevent information flow between access classes. These rules will be applied uniformly to all objects in which data can be stored, namely, files, data segments, capability lists, filing system directories, and inter-process communication objects. Uniform enforcement is possible, because a capability must be present in the cache before a process can perform any action, and because loading capabilities into the cache is controlled by the security-kernel domain.

The security-kernel domain could grant certain other domains exemptions from the lattice security model. Such domains might be used to implement portions of the security kernel itself, to implement a sanitization or downgrading facility, to enforce the lattice model on finer grained objects, such as database records, or to structure a monolithic kernel domain into a collection of smaller domains. Such privileged domains are the analog of the trusted processes that can be found in many existing security kernel implementations [153, 80]. Such trusted domains must be verified, just as the security kernel domain must be. The use of a capability architecture to limit the access of even privileged domains may make the security verification of the code of the domain easier. (It is beyond the scope of this dissertation to discuss security verification. The interested reader may find a useful treatment of the subject in [36].)

7.4 Comparison With Other Systems

This sections contrasts the SCAP secure capability architecture with a number of other capability-based systems. These systems were chosen, either because of their similarity to some aspect of SCAP, or because they are more recent attempts to solve the capability-confinement problem.

7.4.1 System/38

The IBM System/38 supports a concept of authorized and unauthorized pointers that is similar to the secure capabilities of SCAP.[4] In the System/38, unauthorized pointers cannot be used without first checking an access control list (called a user profile in the System/38 documentation). For efficiency, the owner of an object or anyone with *create-authorized-pointer* rights to an object may create

[4]Hank Levy pointed out this similarity.

an authorized pointer that carries access rights to the object within itself and does not require checking of the access control list.

If the System/38 had only unauthorized pointers, it could support the SCAP architecture. Because the System/38 permits a user to create, store, and pass an authorized pointer, an implementation of the lattice security model could be bypassed. The System/38 manuals recommend that the user avoid the use of authorized pointers, because "a user can pass them to other users who have not been explicitly authorized to use an object." [102, p. 2–53] However, the creator of an object can always create an authorized pointer to that object, leading to the potential for a lattice-security-model violation.

7.4.2 SWARD

The IBM SWARD system [164] also has some similarity to SCAP.[5] The SWARD operating system [31] defines *access sets* to control inter-user sharing. Access sets are similar to access control lists. SWARD also contains a primitive control over the propagation of capabilities, in that a SWARD capability cannot be copied unless the capability itself grants copy permission. This type of propagation control seems similar to that in PSOS, but SWARD has not attempted to support the lattice model, and it is unclear from the available documentation whether confinement is possible.

7.4.3 Monash Password-Capability System

The Monash University password-capability system [7] offers a solution to the confinement problem that is equivalent to SCAP's, although implemented in a completely different fashion.

The Monash capabilities are stored as normal values in a user's memory space, with no tagging or capability segments. Instead, each capability consists of two 64-bit fields. One field contains the name of the object, and the other field contains a randomly generated password for the object. The security of the system depends on the extremely low probability of guessing a correct password and trusts possessors of capabilities never to allow them to appear in objects that are readable to unauthorized individuals.

Clearly, such a system is extremely vulnerable to a Trojan horse attack, since the Trojan horse need only expose the value of the secret password to leak access to an object. The Monash system claims to solve the confinement problem by encrypting the passwords of capabilities that grant write permission. Every process in the Monash system has an operating-system maintained *lockword* equal in size to the password. The value of the lockword must not be known by the executing process. If a process wishes to use a capability that grants write

[5] Dorothy Denning pointed out this similarity.

permission, the operating system will first exclusive OR the password P of the capability with the lockword L, giving an encrypted password Q, such that $Q = P \oplus L$. Passwords that grant only read permission are not modified. If a process P1 with lockword $L1$ wishes to run a potentially Trojan-horse-laden package, it creates a new process P2 and specifies a *lock value* V to be used by P2. As opposed to lockwords, lock values are specified by the process. P2 is created with a lockword $L2$, such that $L2 = L1 \oplus V$. P1 now passes capabilities to P2. For those that grant write permission, P1 must first exclusive OR the passwords with V. Since the password has already been exclusive ORed with L1, the result is a password that P2 can use for write permission. However, if P2 gets a password with write permission from any source other than P1, that password will not be properly exclusive ORed with $L1 \oplus V$ and will therefore not work. Thus, P2 can be confined to only write into objects that P1 has approved. The Monash paper shows how P2 can pass capabilities on to subordinates P3, etc. without violating the confinement property.

To support the lattice security model, the process P1 would have to be the creator of all the subsidiary untrusted processes. Prior to passing any capability, P1 would check the non-discretionary access classes and check any access control lists. P1 is thus the analog of the SCAP security kernel.

The important similarity between SCAP and the Monash password-capability system is that the use of lockwords makes capabilities necessary but not sufficient to gain access to an object, just as in SCAP. The difference between the Monash system and SCAP is that the Monash system makes its decisions at the time the capability is passed, rather than at the time it is used. If the Monash operating system simply did the security checks at the time of use (and cached the results), then the notion of lockwords and encryption of passwords would be superfluous.

7.4.4 Honeywell Secure Ada Target (SAT)

Shortly after publication of the initial paper on the SCAP architecture, Boebert similarly pointed out the confinement problems of conventional capability architectures [24]. His work has led to Honeywell's Secure Ada Target (SAT) machine [28, 27] that implements a security strategy quite similar to SCAP. Essentially, a SAT capability is not made available for actual use until both non-discretionary and access-control-list checks have been made. SAT, however, uses a quite complex hardware coprocessor, called the Tagged Object Processor[6] (TOP). By comparison, SCAP can be implemented on unmodified RISC processors or on a VAX processor with small microcode changes.

[6]The SAT program evolved from PSOS and has since be renamed the LOCK program. The Tagged Object Processor has been renamed SIDEARM [184].

7.4.5 KeyKOS

KeyKOS [177] is a capability-based operating system for the IBM System/370.[7] KeyKOS achieves confinement by a mechanism called *factories*. Essentially, a factory is a mechanism for creating new instances of protected subsystems. The factory mechanism provides a way to inspect the newly created domain to ensure that it contains only those capabilities, called *keys* in KeyKOS, that it is supposed to contain and none that a Trojan horse could exploit.

It appears that KeyKOS achieves confinement at a significant cost in performance. The factory mechanism provides total isolation by not only ensuring that a newly created domain has no capabilities to allow writing information from a higher access class to a lower access class, but also ensuring that the new domain has no capabilities that allow reading information of a lower access class. To permit such sharing of low level information, a *filter* must be implemented that executes a trusted program in a separate domain [105, pp. 18–19]. Thus, for a high-access-class domain to read low-access-class information, it must execute two cross-domain calls and returns. By contrast, SCAP simply provides a capability with read permission to a low-access-class memory segment, so such sharing occurs as fast as any other instruction references. Furthermore, a cross-domain call in KeyKOS is described [105, p. 25] as requiring "about 200 instruction cycles for a typical invocation." This language is ambiguous, and if an *instruction cycle* is a microcode cycle, then the KeyKOS cross-domain call performance is very impressive. If, however, an *instruction cycle* is the time for a typical IBM System/370 machine instruction and an invocation is a cross-domain call, not including a cross-domain return, then a KeyKOS cross-domain call seems to be relatively more expensive than a CAP-I cross-domain call.[8] Furthermore, the SCAP cross-domain call optimizations, described in Chapters 17 and 18, achieve better performance ratios than CAP-I.

7.4.6 Flex

The Royal Signals and Radar Establishment has developed a capability-based system called Flex. Although Flex was not originally designed to address the confinement problem, Wiseman [234] is developing classes of trusted type managers that can enforce non-discretionary security controls. Wiseman's prelimi-

[7] KeyKOS was originally developed by the Tymshare Corporation under the name GNOSIS [111]. KeyKOS is now sold by Key Logic, Inc., a company formed specifically to develop and market KeyKOS.

[8] Cook [45] reports a cross-domain call and return requiring the equivalent of approximately 114 instructions. Cook measured the cost of both a call and return, while KeyKos measured the cost of just a call. Assuming that call and return are roughly equal in cost, this makes the ratio 200:57 or 3.5:1. This type of comparison is highly suspect, because the benchmarks that were run were quite different in character. The only safe conclusion is that KeyKOS cross-domain calls appear to be relatively more expensive than CAP cross-domain calls.

nary design seems to rely on the creation of a secure type manager for classified objects. The only way one can make use of a capability for a classified object is by actually calling the classified object manager. It is not clear from his paper whether the classified object manager will interpret all references or whether capabilities to the underlying objects will actually be released. In the latter case, it is not entirely clear how those capabilities will be confined. In some respects, the Flex design appears to resemble the PSOS secure document manager, and might suffer from some of the same problems. However, only a preliminary design of the Flex confinement solutions has been published thus far, so it is premature to make any judgements. Flex also has a number of interesting features, discussed in Section 9.5.3, for dealing with discretionary Trojan horses.

7.5 Kain and Landwehr's Taxonomy

Kain and Landwehr [113] have developed a taxonomy of capability-based systems to better understand why some can easily support non-discretionary controls, while others seem to have great difficulty. Their taxonomy is based on considering the following sequence of six questions about the life of a memory segment that is to be addressed through a capability.

1. What happens when a capability is created?
   (a) No access rights are inserted.
   (b) Access rights are inserted.
2. What happens to the prepared-for-access capabilities that describe a segment, if the security attributes of that segment are modified?
   (a) Access rights are not changed.
   (b) Capability is flagged for future change.
   (c) Access rights are updated immediately.
3. What happens to the stored-in-memory capabilities that describe a segment, if the security attributes of that segment are modified?
   (a) Access rights are not changed.
   (b) Capability is flagged for future change.
   (c) Access rights are updated immediately.
4. What happens when a capability is copied?
   (a) Access rights are not changed.
   (b) Access rights are further restricted by context rules.
   (c) Access rights are set to the maximum consistent with policy.

(d) Access rights are are updated properly by trusted software.

5. What happens when a capability is prepared for use?

   (a) Access rights are not changed.

   (b) Access rights are restricted by policy.

   (c) Access rights are set to the maximum consistent with policy.

6. What happens when the processor attempts to use the capability?

   (a) No checks are made.

   (b) The reference is checked against available rights.

   (c) The reference is checked against maximum possible rights.

The answers to these questions determine how well or poorly the capability system can enforce the confinement property. In particular, question one is most critical in determining the ease of enforcing non-discretionary rules. Essentially, question one asks whether binding of access rights to a capability occurs at the time of creation or whether binding is delayed until the capability is actually used. Delayed binding of access rights is the strategy used by both SCAP and SAT to implement the lattice security models.

Questions two and three deal with immediate revocation. If access rights in a capability are not changed when the security attributes of the object change, then revocation is impossible. The choice between flagging and immediate update is a question of performance. Flagging for future change usually will give better performance, as many of the capabilities that exist may never be used.

Question four examines how capabilities are copied from domain to domain. Restrictions on copying can be used to enforce confinement, but they have all the problems described above in Section 7.1.2.

The answers to questions five and six are directly related to the answer to question one. If access rights are bound at creation time, then no access rights changes are required, either at time of preparation for use or at time of use. If access-rights binding is delayed, however, then the access rights must be modified at time of preparation or at time of use.

Applying the taxonomy, a conventional capability system, such as CAP-I or the Plessey System 250 is *baaaab* or *badaab*, while a system like PSOS is *baadab*. By contrast, the SAT is *aaaacb*, and SCAP is *abbabb*. (Kain and Landwehr [113] incorrectly categorized the SCAP system as identical to SAT, although the conclusions that they drew from the incorrect categorization were valid. The differences between SCAP and SAT appear to be in revocation, rather than support of non-discretionary controls. They based their characterization on the 1984 paper on SCAP [119].

Chapter 8

Traceability of Access Problems

8.1 Asymmetric Views of Security

Capability and access-control-list systems have always been viewed as equivalent, because they are simply alternate views on the same Lampson access matrix. In practice, the two systems are not necessarily identical because of the differences in representing privilege. Users frequently ask the follow two questions of an access-control system:

- To which objects does a given subject have access?
- Which subjects have access to a given object?

The questions are symmetrical. In a capability system, the first is easily answered by inspecting a single capability list, but the second requires a search of all capability lists. For access-control-list systems, the second question is easily answered by inspecting a single access control list, but the first requires a search of all lists.

Expressing security policies with pure capability systems is less obvious than with access-control-list systems, because the second question is asked much more frequently than the first. In the world of security, one is concerned with who is to be granted access to particular data. A security officer investigating an incident needs to know who has access to a compromised object. It is much less common for a user to want a list of all the objects to which he or she has access. As a result, most actual commercial systems have been based on access control lists, rather than capabilities.

The *traceability-of-access* problem is the problem of determining who has access to a given object. This chapter will show how the SCAP architecture that was designed to solve the capability-confinement problem can also solve the traceability-of-access problem and provide the benefits of both a capability-based approach and an access-control-list-based approach.

8.2 Discretionary Security with SCAP

This section modifies the example of a missile trajectory subsystem from Figure 7.1 on page 70 by considering discretionary security controls. Suppose in this new example shown in Figure 8.1 that the subsystem M was written by a user V who wishes to steal a capability for the data D passed to M by U and that V can work in the same access class as U. V's strategy might be to cause M to store a copy of the capability for D in file F and then subsequently to run M himself and cause it to give him a copy of the capability. If there is a revocation scheme for disabling argument capabilities after completion of a domain call, V might try to subvert the revocation by running another, parallel instance of M in his own process and passing the capability via a shared capability list common to both instances. A brute-force solution to these copying problems is to insist that all capability lists in a domain that are shared between instances cannot be writable, but this is a rather restrictive view and limits the utility of protection domains.

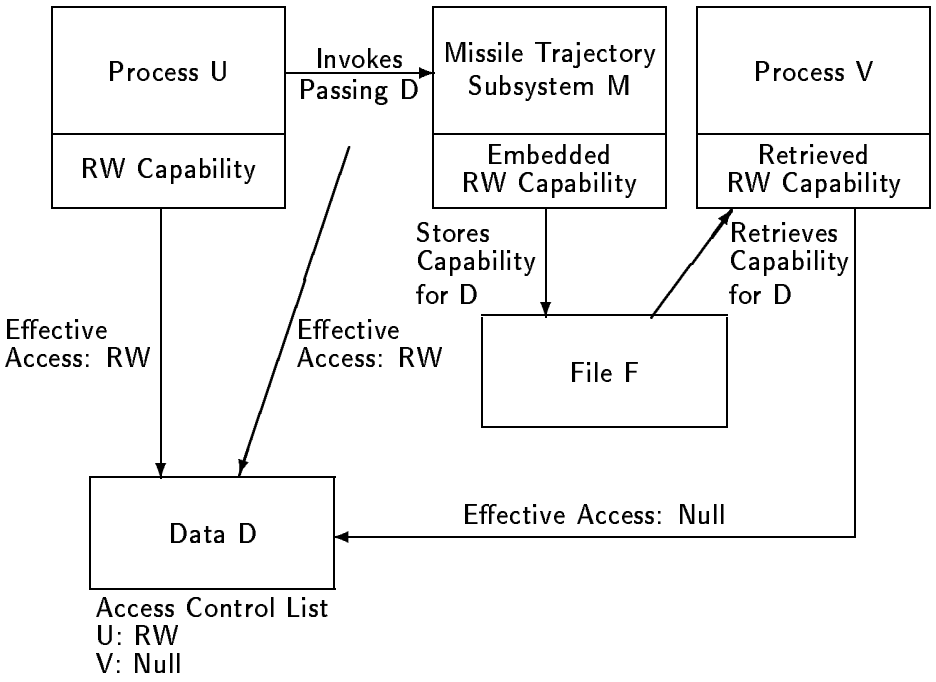

Figure 8.1: Access-Control-List Limited Capabilities

The full solution is to include access-control-list inspection as part of the basic, capability-evaluation mechanisms. Suppose each object description includes some representation of its access control list. Further, make the user name of the user logged into a process available to the capability machinery. Discretionary

controls can then be applied if the rules are imposed that the access control list should be checked as part of capability evaluation and that capabilities from a shared capability list in different instances of a domain are held at different places in the capability cache. (The capability unit of the Cambridge CAP computer has exactly this property.) Then, the first time a domain comes to use a capability, the access control list will be checked. In the example above, the instance of M run by V will not be able to use a capability for D stored in file F, because V is not in the access control list for D.

Using the hardware and microcode of the system to handle access control lists would make the capability unit needlessly complex and restrict the system to a particular style of security policy. The same caching strategy that was used for non-discretionary controls can also support access control lists. Note that the access control list need only be checked the *first* time a capability is used by an instance of a domain; thereafter the cached copy is sufficient. Therefore, the security-kernel domain can evaluate the access control list at the time that it loads the capability into the cache and can refuse to load the capability, if the access control list so directs. So, by a simple change to the capability mechanisms that support intervention during the evaluation of capabilities, the system can support arbitrary non-discretionary and discretionary security policies. This type of intervention is analogous to missing-segment-fault processing in Multics [168] and similar operating systems. Given a large enough cache to prevent excessive re-evaluation of capabilities, the performance cost should not be excessive.

In the context of discretionary security, there is another line of attack by which V can acquire a copy of the data D. The technique is simply to make a copy to another object within M. (It should be noted that D cannot be overwritten by this route, but only stolen.) The access control list on the second object must indicate that the object is writable by U so that M can make the copy and readable by V so that the data may be read out subsequently. It is precisely this problem, inherent to any discretionary policy, that motivates practical security systems to employ non-discretionary and discretionary policies in conjunction. By varying the access classes of U and V appropriately, the confinement property can be relied upon to restrict information flow from D to the second object.

The security manager in a protected system needs to engage in operations such as modifying the access class of information and setting up group access control lists. These privileges can themselves be represented as capabilities that are recognized by the security kernel, and the mechanisms outlined to allow them to be kept in the filing system and treated as *bona fide* capabilities or privileges.

It is also possible to cope with immediate revocation in the scheme without resorting to the complication of indirection through special, revoker capabilities found in other capability systems [179]. If an access control list is modified for a particular object, all entries in the cache for that object must be flushed. This will cause all capabilities for the object to be evaluated afresh, and any changes

in discretionary policy will be noted directly.[1] Revocation is discussed further in Chapter 11.

Thus, SCAP supports both discretionary and non-discretionary access controls by augmenting a basic, capability mechanism. It is important to note that the scheme relies on:

- carrying information about access classes and access control lists down to the most primitive parts of the kernel, and
- taking some care in the rules for leaving evaluated capabilities in a capability cache.

The basic rule is that when a capability is first used in an instance of a domain it must be subject to both access-control-list checking and non-discretionary checking. Such a scheme is easy to implement on a variety of machine architectures. In the Cambridge CAP Computer, it could be built into the microcode that loads capabilities into the capability unit. In either a RISC implementation or in the VAX-11/730 implementation of SCAP, the rule could be built into the page fault handler of the operating system.

Furthermore, SCAP provides the lattice security model and traceability-of-access transparently to most applications programs. That is, a software module can be encapsulated as a protection domain, and as long as it operates at a single access class and makes no security violations, all of its capability operations will work correctly.

In most capability machines, the creation of domains and organization of capabilities are done by language compilers, and the user is quite unaware of the underlying machinery until his program commits some transgression. This combined transparency is especially important, because not all software developers and users may run the lattice security model or a capability system and may not design software to be constrained in such ways. However, users of the lattice security model will want to run such software, albeit in properly confined domains of protection.

[1]The use of the capability cache in the revocation procedure is certainly a form of indirection. However, the Redell revocation scheme uses a similar cache and requires more indirection.

Chapter 9

Discretionary Trojan Horses

The notions of non-discretionary controls were developed to deal with Trojan horse attacks. All of the non-discretionary security models deal with Trojan horses by subdividing information into a set of access classes. Only Trojan horse attacks *between* access classes are prevented. These non-discretionary models have dealt well with a large class of problems in the military and also can be used for some commercial applications [144]. Many applications, however, still wish to use discretionary access controls, such as capability lists or access control lists (ACLs), to implement their security policies. Since discretionary controls are inherently vulnerable to Trojan horse attacks, there is growing concern [63] that some form of Trojan horse protection be developed for pure, discretionary environments.

It is has been claimed [238, page 107] that a capability-based system can alleviate discretionary Trojan horse problems, because a suspect program could be run in a separate domain of protection and could be granted capabilities only to the particular files that are required to perform its function.

This chapter[1] shows that conventional capability systems cannot solve this problem for practical applications. A solution based on the secure server techniques developed for non-discretionary protection appears to solve many of the discretionary Trojan horse problems both for SCAP and for conventional access-control-list (ACL) systems.

9.1 Directory Management

A discretionary Trojan horse can gain access to all of a user's files, because the operating system provides a directory manager that translates human-readable file names. The Trojan horse need only quote a selected file name, and the directory manager will provide access, if the unsuspecting user of the Trojan

[1]This chapter is a revised version of a paper [117] presented at the 1987 IEEE Symposium on Security and Privacy.

horse has the appropriate access rights. Figure 9.1 shows a Trojan horse in the FORTRAN compiler that surreptitiously modifies a user's `LOGIN.COM` file, while compiling the user's program.

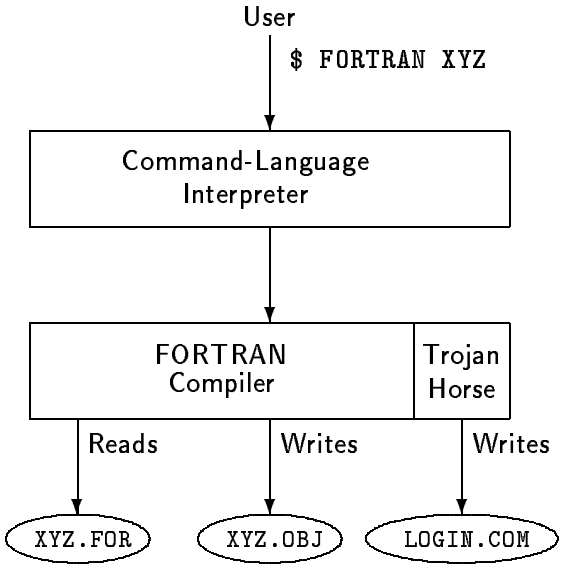

Figure 9.1: Trojan Horse in Action

As mentioned above, it is frequently claimed that a capability-based system can alleviate this class of discretionary Trojan horse problem, because the suspect program could be run in a separate domain of protection and could be granted capabilities to only the particular files required to perform its function. Close examination of the directory management reveals the problem.

In a capability-based system, the directory manager is responsible for storing capabilities to various objects, associating human-readable names with these stored capabilities, and returning capabilities to the user when the appropriate name is requested. In the simplest case, there would be a directory for each user, storing that user's capabilities. However, it is clearly unreasonable to require that each user have a private capability for objects that are shared by many users. For example, it would be far too expensive and cumbersome to store one capability for each potential user of the FORTRAN compiler. Instead, a single capability for the compiler could be stored in a public directory of all compilers, and all users would possess a capability for that public directory.

Many capability directory managers represent different access rights for different users by associating with the object something similar to an access control list to describe who may retrieve a capability for an object, given that the user

in question already possesses a capability for the containing directory.[2] For example, the CAP-I Operating System [231, chapter 4] uses an access matrix to encode who may retrieve a capability for an object.

Name-to-capability translations by a directory manager reintroduces a security problem that capability systems were attempting to remedy. The capability system was supposed to limit a domain to only those objects to which it either had embedded capabilities or for which capabilities were passed as arguments. For example, if a user runs a text editor in an access-control-list system to modify a file, the editor can open any other file to which the user may have access. Ostensibly, a capability system allows the user to encapsulate the editor and only allows the editor access to the specified file. However, most text editors allow the user to enter file names while the editor is running, instead of requiring that the names be specified when the editor is called. The editor must have access to the user's directory, and even in a system like the CAP-I operating system, the editor could retrieve capabilities for files that the user did not intend.

The problem here is not inherent in the design of capability systems. If the capabilities were used properly, the discretionary Trojan horse would not succeed. The problem is that using a capability system properly requires extreme discipline on the part of the software authors and imposes limitations on the human interface. Most authors and users are unwilling to accept those limitations, and as a result, even the CAP-I text editor proved vulnerable to the discretionary Trojan horse attack.

The CAL operating system [205, page 57] addressed the problem by defining two search lists: one for commands entered by the user from a terminal and one for all other programs. Each search list consisted of a set of directory capabilities that allowed the program to retrieve and/or create files. The second search list usually contained much less powerful capabilities. While this solution limited the problem, a malicious system command could still gain access to objects not intended by the user.

9.2 Name-Checking Protected Subsystem

A better solution is to use the command interpreter's knowledge about the commands that are to be invoked by interposing a special, name-checking protected subsystem between the suspect program and the directory manager. The purpose of the protected subsystem is to inspect all requests for name-to-capability translations and compare them with the requests actually typed by the user and with the behaviour patterns expected of the program in question.

[2] Saltzer and Schroeder [182] discuss in more detail the need for directories to support capability sharing.

The name-checking subsystem uses the same command-definition tables as the command interpreter, itself.[3] Commands and user programs obtain their parameters by explicitly calling the name-checking protected subsystem, rather than receiving the parameters on the stack. This method of obtaining parameters is analogous to the use of the PARMS protected subsystem in CAP-I [231, page 44] or the CLI-callback mechanism in VAX/VMS. Furthermore, a program will be able to translate a name to a capability only by calling the same name-checking subsystem. Direct access to the directory manager will be forbidden.

Thus, a program or system command can only retrieve a capability from the directory manager by calling the name-checking subsystem, so the program's requests for objects can be checked. For system commands, the command-language database defines what types of objects are required. The name-checking subsystem knows that the FORTRAN compiler required read access to the source code in a file with a user-supplied name having a suffix of `.FOR`. Likewise, the name-checking subsystem knows that the compiler created new files with the same user-supplied name, but suffixes of `.OBJ` and `.LIS` to store the object code and the compiler listing, respectively. However, if the compiler attempts to create or to write into a file named `LOGIN.COM`, the name-checking subsystem will recognize that such a file name is not usual for the FORTRAN compiler.

Given an unexpected name, the name-checking subsystem has two choices. It could simply return an error to the compiler, but such a strategy is overly restrictive. A better approach is to query the user about the request. Queries are particularly appropriate for programs that are not in the command-language database. The query would inform the user of the file name and the access mode requested. Any actual implementation should support a user-settable option to control whether to query or to abort. The user could also specify new rules for name checking, just as a user of VAX/VMS can specify new command tables.[4]

Figure 9.2 shows the Trojan horse in the FORTRAN compiler blocked by a name-checking protected subsystem that queries the user when the Trojan horse attempts to write into `LOGIN.COM`.

The implementation of the name checking must be in a protected subsystem to ensure that the right to retrieve capabilities from directories is not abused. The name-checking subsystem must be able to guarantee that input came from a human being and not from some program masquerading as a human being. In this, the name-checking subsystem requires a *trusted path* to the user,[5] much

[3]The VAX/VMS Command Definition Utility [220] uses such tables.

[4]An obvious drawback to querying the user is that the user may always answer "yes", regardless of the content of the query. User apathy can only be resolved by frequent security training and by auditing all responses to queries.

[5]The need for a trusted path between the user and the secure operating system was first identified by Bell, Fiske, Gasser, and Tasker [13] in early 1972. The trusted path provides guaranteed communication between the human being and secure software, complete with two-way authentication.

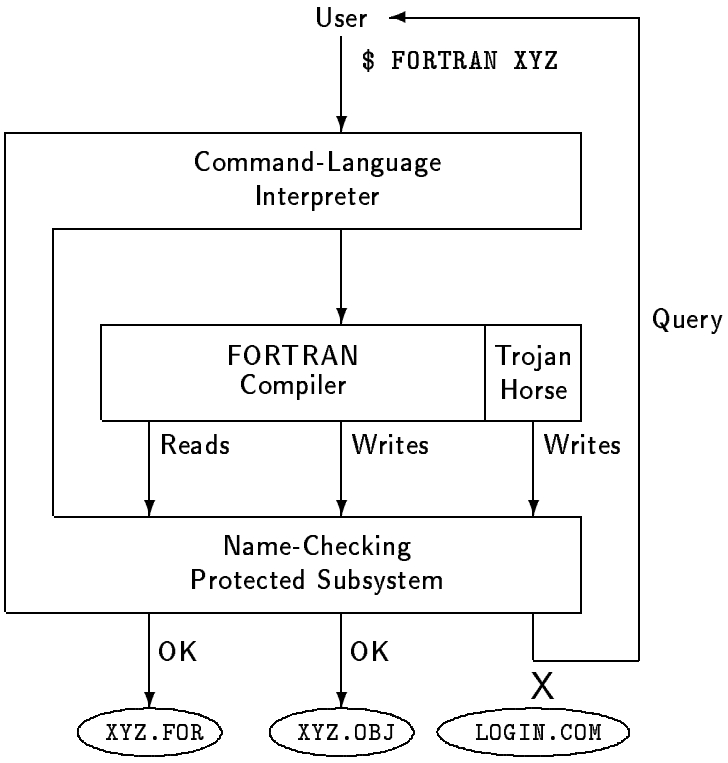

Figure 9.2: Trojan Horse Blocked

like the secure server of a security kernel. However, unlike the secure server, the name-checking subsystem is only enforcing discretionary access controls.[6]

The example above showed how the name-checking subsystem might be used from a compiler that makes very stereotypical object references. A text editor, by contrast, may reference arbitrary objects based on user input to the text-editor's command stream, rather than to the command-language interpreter. This could result in a bad human interface, if the user typed a command to the text editor, and the editor then called the name-checking subsystem to query the user if the file name were correct. A better solution is to change the text editor to not query the user directly for the file name, but rather to ask the name-checking subsystem to query the user. The name-checking subsystem can assure itself

[6]Since the name-checking subsystem does not enforce non-discretionary controls, it need not be formally specified or verified to meet the A1 requirements of the National Computer Security Center [59]. Since formal verification of the full command interpreter would likely be very difficult, it is quite important to not require such verification for an A1 rating. The name-checking subsystem must also control installation of new command tables through the trusted path. Otherwise, a Trojan horse could attack the system by first installing new name-checking rules in a new command table and then later taking advantage of those new rules.

that the human being (and not a Trojan horse) actually specified the file name, and the human being only has to reply to a single query, rather than two.[7]

The name-checking subsystem must also support the use of scratch files. The simplest solution is to force such files into special, temporary directories, as suggested in [79] for securing the UNIX */tmp* directory.

9.3 Name Translation in Batch Jobs

The strategy described thus far appears suitable for programs that run interactively with a human being present at a terminal. However, the strategy breaks down when one considers large batch jobs that may reference many different objects. In particular, if we consider the UNIX *make*(1) command [212, pp. 1-244 – 1-246] or DEC/MMS (Module Management System) [215], we can see that a human being may issue a single command that references hundreds or thousands of distinct files. The user would be quite unwilling to run such jobs interactively, because they may take many hours of CPU time. Instead, there must be a facility to ensure that such large batch jobs do not gain access to files that were not intended by the user. There are several possible solutions to the problem of large batch jobs, each appropriate in different situations.

9.3.1 Special Directory Trees

The first solution is to construct a special directory tree containing only objects that are to be operated upon by the batch job. Then grant a capability for the root of that directory tree that allows the batch job to retrieve capabilities directly. This solution is straightforward to implement, but requires that the user carefully construct directory trees that contain only objects on which the batch job is allowed to operate. This solution can also be used for interactive applications, but requires that the user plan ahead even more. Special directory trees are likely to already be used in large software development projects.

9.3.2 Pre-Compiled Batch Jobs

The second solution is to have a special protected subsystem that will compile the batch job or the make file into an intermediate form. The compilation pro-

[7] The human interface of the name-checking subsystem can be simpler than that of a *secure server*. A human being must depress a secure attention key to provide two-way authentication of the human to the secure server and the secure server to the human. Two-way authentication is needed, because the user may type secret information (such as a password) to the secure server. However, the name-checking subsystem will never require secret information from the user that must be kept secret from the applications program. Therefore, although the name-checking system will require an assured path to the human (and not to a Trojan horse), the user will not require the assurance of the secure attention key.

cedure will examine each command that is to be executed and will record in the intermediate form what capabilities it will need. The compilation will be done interactively, and the user can be queried (just as for an interactive command) if the batch job appears to reference unexpected objects. The intermediate form of the batch command procedure must be stored as a protected object, so that untrusted code cannot modify it. Essentially, the compilation of the batch file and the later interpretation of the intermediate form are additional functions of the PARMS protected subsystem discussed in Section 9.2. It should be noted that compilation of large batch jobs and make files can have significant performance advantages, because a large percentage of a batch job's execution time may go into parsing and interpreting command procedures. The Burroughs B6700 Work Flow Language (WFL) [228] is an example of a compiled batch command processor.

Clearly, the special, directory-tree solution is much simpler than pre-compiled batch jobs. The principal drawback of special directory trees is that they require considerable advance planning on the part of the user, and any error in that advance planning could provide an opportunity to a discretionary Trojan horse. While pre-compilation is a complex procedure, it is much less prone to user error.

9.3.3 Additional Approaches

Pre-compiled batch jobs suffer from two serious problems. First, a compiler is needed, not just for the command language and the make utility, but also for other utilities that have complex command languages that could run in batch, such as VAX DATATRIEVE or an SQL relational database language. Pre-compilation may not be possible at all for complex user-written utilities. Second, the compilation process could easily generate hundreds of file names to be authorized, and the typical user would quickly become bored and probably miss the one Trojan horse attack amongst the hundreds of valid references.

Two alternatives to pre-compiled batch jobs have been suggested recently: wildcard authorization and post authorization. Neither of these ideas have been studied in depth, as yet, but both look promising for future research in preventing discretionary Trojan horses.

9.3.3.1 Wildcard Authorization

Kahn [112] has suggested a scheme of wildcard authorization, in which the user authorizes entire classes of file references, prior to starting the batch job. For example, the user might issue the command:

```
$ AUTHORIZE WRITE DUA0:[KAHN.WORK]*.OBJ;*
```

This command would authorize the batch job to write into any file of type `.OBJ` in Kahn's WORK subdirectory. In a sense, wildcard authorizations are a

better method of specifying special directory trees. Of course, the AUTHORIZE command would have to be issued through the trusted, name-checking subsystem to avoid Trojan-horse attacks.

9.3.3.2 Post-Authorization

Lomas [147] has suggested a method of post-authorization in which a batch job is allowed to modify or create any files it wishes. However, any modifications to existing files would be made to new versions of the files, rather than the existing versions, and all the files, whether modified or created, would be marked inaccessible. Similar steps would be taken for operations such as file deletion, access control list changes, etc. At some time later, the user would return and authorize the new files through the trusted, name-checking subsystem. The advantage of Lomas' scheme is that no pre-compilation is required, and only those files that are actually modified or created require authorization. Files mentioned in the batch job, but not actually touched, would not require authorization.[8]

9.4 Access-Control-List Systems

Thus far, this chapter has examined the discretionary Trojan horse problem in the context of a capability-based system, but recognizing expected file names can also be applied in an access-control-list (ACL) system. For interactive sessions, the approach is the same. The command interpreter knows what file names to expect, based on information from the command-definition tables and on the information typed by the user. If the program requests an *unexpected* file name, the user is queried about the legitimacy of the operation.

For batch jobs, however, only the pre-compiled batch job strategy will work. The special directory-tree approach depends on the ability to grant a capability to a particular batch job and not to all batch jobs that may be run by that user. Conventional, access-control-list systems cannot express such selective granting of access rights.

9.5 Alternate Strategies

This section contrasts the approach for a name-checking protected subsystem with other attempts to deal with the discretionary Trojan-horse problem.

[8] Post-authorization could become quite complex in the case of files referenced remotely over a network. In principle, one could construct a network post-authorization protocol, but the details and synchronization requirements are likely to be difficult.

9.5.1 Strict Need-to-Know Policy

One of the earliest solutions proposed for the discretionary Trojan-horse problem was in the Case Western Reserve University's non-discretionary security model by Walter [225]. The Case model suggested implementing a *strict need-to-know* policy in which a process is not allowed to copy information from one object r to another object r', unless all of the users authorized to read r' were also authorized to read r. An analogous policy could be constructed for *strict need-to-modify*. However, Case found the security scheme "much too rigid to be of any use, except perhaps in the special case of a small environment." [225, page 15] They found the complexity prohibitive for even such simple cases as adding a new user to the list of users authorized access to some particular object.

9.5.2 Enhanced Linker

Boebert and Ferguson [25] have proposed an enhanced dynamic linker as a partial solution to the discretionary Trojan-horse problem. Their solution (developed concurrently with and independently from the proposals in this chapter) similarly interposes a protected subsystem between the suspect program and the actual file system. Boebert and Ferguson force all file references to go through a trusted dynamic linker that compares the name of the user who invoked the program, the name of the originator of the program, and the name of the owner of any data files. If the user invokes a program owned by someone else, and that program contains a discretionary Trojan horse that attempts to tamper with the user's files, the dynamic linker will recognize the name mismatch and raise the appropriate alarms.

The primary difference between Boebert and Ferguson's proposal and this proposal is that the dynamic linker bases its decisions solely on the name of the user and the name of the originator of the program. If I use a borrowed FORTRAN compiler, I want that compiler to read my source file and produce an object file. The name-checking subsystem uses the knowledge built into the command-definition tables to recognize expected and unexpected actions of the suspect program. Thus, the false alarm rate should be lower using the name-checking protected subsystem. The mechanism of interposing a name-checking subsystem is the same in both proposals. Only the decision-making process differs.

9.5.3 Flex Cartouches

The Royal Signals and Radar Establishment's capability computer [234], called Flex, has a unique way of representing capabilities that are stored within more complex data structures. [201] (Flex uses a tagged capability architecture and permits capabilities to be intermingled with data.) When an object is displayed

on the screen, the Flex editor distinguishes between ordinary text and capabilities by displaying the capability in a *cartouche*, or box. Anything displayed in a cartouche is a value, pointed to by a capability; while anything not displayed in a cartouche is simply text. Figure 9.3 shows a very simple example of what Flex calls an *edfile*. The edfile contains text, a capability to an object containing an integer and a capability to another edfile.

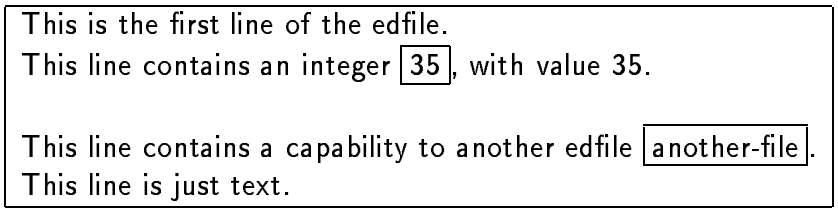

Figure 9.3: Example of Cartouches in a Flex edfile

The important thing about cartouches is that they represent the capabilities, themselves, and the user can manipulate the capabilities directly, using a window-based editor with a mouse.[9] As a result, many objects require no names at all, and the opportunities for discretionary Trojan horses to exploit name-to-capability translation are nearly non-existent. Flex has not been designed particularly to deal with Trojan-horse problems, and it does not appear to have a mechanism to prevent the display of a bogus cartouche. However, the human interface of Flex, with its direct manipulation of capabilities, presents an intriguing possibility for future approaches to limiting the damage potential of Trojan horses. The author's lack of direct access to a Flex computer prevents further exploration of the use of cartouches in this dissertation. However, there seem to be quite promising opportunities for future research on the use of a secure window manager to limit the damage potential of both discretionary and non-discretionary Trojan horses.

9.6 Limitations of the Technique

Recognizing unexpected file requests appears to be an effective technique to limit the damage that a discretionary Trojan horse can do. The technique appears most effective against unauthorized tampering or sabotage, because all unexpected file requests can receive human review and audit. Of course, this technique does nothing to stop a Trojan-horse-laden program from tampering with the files that it is supposed to process. The files could be destroyed, or the program could simply give wrong answers in certain cases. Since such tam-

[9]Flex is currently implemented on re-microprogrammed ICL Perq workstations.

pering often cannot be distinguished from non-malicious program bugs, security techniques will not be useful here.

Recognizing *unexpected* file requests is of less use in preventing unauthorized release of information, because the Trojan horse could surreptitiously encode sensitive information in its legitimate output files, using any of the usual storage-channel techniques for communication. This result should not be surprising, given Harrison, Ruzzo, and Ullman's results [89] that the confinement problem is undecidable for generalized access matrices, such as most discretionary access-control systems.

The reason the technique can work at all is that computer systems are used in extremely stylized ways. By codifying that information into the command-definition tables, we can build simple, protected subsystems that can detect when a discretionary Trojan horse violates those patterns of normal use. The user must adjudicate whether violation of the normal patterns is legitimate or malicious, but because the detection system is table-driven, the frequency of user queries can be kept to a minimum.

Chapter 10

Implementing Commercial Integrity

The Clark and Wilson commercial integrity model [40] was introduced in Section 3.2.3. This chapter[1] presents how the SCAP architecture can be used to implement the Clark and Wilson model and contrasts that design with other proposed implementations.

The model contains two classes of objects: constrained data items (CDIs) and unconstrained data items (UDIs). CDIs are the objects to be protected, and UDIs are used for simple data input. The model defines two classes of procedures to operate on CDIs: integrity verification procedures (IVPs) that check the CDIs for integrity and consistency and transformation procedures (TPs) that change the set of CDIs from one consistent state to another.

The model contains a set of five certification rules, called **C1** to **C5**, and four enforcement rules, called **E1** to **E4**. The rules are defined on page 38.

10.1 Implementation

Examination of the Clark and Wilson model immediately suggests an implementation based on protected subsystems. CDIs can be built out of abstract data types with TPs as the operations of the type manager. Sealed capabilities, described in Section 4.4 appear sufficient to support the enforcement rule **E1**. It was lack of support for **E1** that Clark and Wilson most criticized in both the Biba integrity model [18] and the Lipner commercial integrity model [144].

10.1.1 Certification Difficulties

Supporting **E1**, however, is critically tied to certification rule **C2** that requires that all TPs be certified valid. Not only must the TPs be certified to take CDIs

[1] A paper [115] based on this chapter is scheduled for presentation in April 1988.

from valid states to other valid states, they must also be certified not to pass their access rights for the CDIs to other non-certified procedures. The problem is that certification of TPs is extremely difficult. One would initially be tempted to use software correctness proof techniques, but these techniques are notoriously difficult to apply[2]. The Bell and LaPadula lattice security model went to great effort to minimize the portion of any system that would require certification. The primary purpose of their confinement property is to confine any applications code that could not effectively be certified.

Thus, the Clark and Wilson model should be implemented in such a way that inspecting and proving things about the security properties of the system are easy, without requiring full correctness proofs of the TPs. Even with full correctness proofs of the TPs[3], separation of the security issues from the basic algorithmic correctness issues would be helpful, because one would like to change the list of who may run a particular TP without having to reprove the code of the TP. Therefore, the relations discussed in rules **C2** and **E2** must be implemented as part of the security kernel of the system, rather than as part of the code of the TPs themselves.

10.1.2 Crucial Role of the Audit Trail

Clark and Wilson call for the maintenance of an audit log in rule **C4**. As with most other security models, however, they do not recognize the critical role that the audit trail must play. In particular, the separation-of-duty rule, **C3**, can only be enforced by using information about past actions. The audit trail is the primary source of such information.

More generally, permission to execute certain transactions or to modify certain CDIs can only be granted if certain previous transactions have been executed by specific individuals. Any system that supports such a dependency rule must deal with possible violations of that rule. For example, if user A executed some transaction TP1 that in turn made it possible for user B to execute transaction TP2 exactly once, user B might maliciously attempt to execute TP2 more than once. If user A executing TP1 grants user B permission to execute TP2, TP3, and TP4, each exactly once, the system must ensure that B does not maliciously execute TP2 three times instead.

Since dependencies on previous transactions can be made arbitrarily complex by applications designers, any system to enforce such dependencies must make

[2]See De Millo, Lipton, and Perlis' classic paper [54] on the social processes involved in formal verification.

[3]IBM's Hursley Laboratory together with the Oxford University Programming Research Group have done some work on formal specification of IBM's CICS transaction processing system [237]. However, full correctness proofs of arbitrary user-written TPs remain impractical, today.

the audit history available to the access-control software. This is in direct contrast with almost all previous security systems that have viewed the audit trail as a subsidiary function for determining what may have happened after the fact.

The role of the audit trail in determining access policy was recognized by Fernández, Summers, and Wood [69, pp. 62 and 91]. Their notion of *history-dependent control* is primarily aimed at the statistical-inference problems in databases, rather than at separation of duty. They do recognize the need to record past actions on the database and to make future requests conditional on those past actions.

10.1.3 Implementing with Secure Capabilities

One can limit the need for certification of the TPs and make inspection easier by using the properties of SCAP's secure capabilities. Capabilities in SCAP are necessary to gain access to a protected object, but are not sufficient by themselves. Therefore, one can use sealing to implement the abstract-type-management facilities required for rule **E1**. SCAP can also intercept attempts to exercise enter capabilities for the TPs and require that special access-control-list checks be made to enforce the separation-of-duty requirements. Such an access control list (ACL) for a TP would contain the relations from rule **E2**.

The kernel itself audits all invocations of TPs and modifications of CDIs. The audit records include information on who performed the operation, what TP was invoked, and what authorization information was used to grant access. To deal with the separation-of-duty requirements of rule **C3**, one could simply make use of this information. The audit trail is a very large and complex data structure, however, and the cost of searching it on every invocation of a TP would be prohibitive.

Instead, a special kind of capability is added to be used as a token that a particular TP has been executed. Although they take the form of capabilities to prevent unauthorized tampering, *token capabilities* are in fact copies of individual audit records. Making separate copies of the audit records and passing them around as capabilities is solely a performance improvement to clarify exactly which audit record authorizes a future transaction.

Assume that running TP1 is prerequisite to running TP2. The access control list on the entry to TP2 would specify that the caller must be a specific person who possesses a token capability for TP1. Token capabilities record both the name of the transaction and the name of the user who executed the transaction. The user's name must be recorded, so that one could ensure that TP1 and TP2 were executed by *different* people, if required. Executing a particular TP may grant a user the right to execute several other TPs. The system must ensure, however, that each TP is executed the proper number of times, and that tokens authorizing the execution of two different TPs, each exactly once, are

not instead used to execute one of the subsequent TPs twice. To prevent this abuse, token capabilities also contain the name of the TP that they authorize, and a token capability can be used only once.[4] Once used, the token capability is marked to prevent further use. Note that the token capability is not revoked or destroyed upon use. A bit within its internal state is set to record that it has been used. Since there may be a large number of token capabilities in use at any time, revocation upon use could impose an excessive performance burden on the system. Instead, the used token capability is left in the user's capability list, for deletion at some later more convenient time.

It is important to realize that token capabilities do not carry any more information than is already recorded in the audit trail. They simply provide a mechanism for making the proper audit records available for integrity policy decisions on a timely basis with minimal overhead for searching.

Entries in a separation-of-duty ACL must be more complex than the simple ACLs that were supported in systems such as Multics. Each entry must consist of a boolean expression in which the first term is the name of the user who proposes to execute the transaction, and the other terms are token capabilities for required predecessor transactions. The boolean expressions support the following operators:

- + for logical and,
- | for logical or,
- $\sim$ for logical negation,
- ! for a separation-of-duty wildcard in which the userID is allowed to be any value except that used in any of the other terms of the expression.[5]

Three examples of separation-of-duty ACLs follow. The first ACL contains a single entry that specifies that Smith may execute the TP, but only if JONES has previously executed TP1.

```
(IDENTIFIER=SMITH+JONES.TP1,ACCESS=EXECUTE)
```

The second example shows an ACL with two entries in which either SMITH or JONES may execute the transaction, but only if JONES previously executed TP4 and SMITH previously executed TP7.

```
(IDENTIFIER=JONES+JONES.TP4+SMITH.TP7,ACCESS=EXECUTE)
(IDENTIFIER=SMITH+JONES.TP4+SMITH.TP7,ACCESS=EXECUTE)
```

[4]In principle, one could define token capabilities that could be used a specific number of times. Lacking a specific requirement for that level of complexity, this proposal restricts tokens to one-time use, although nothing in the design precludes reusable tokens.

[5]The VAX/VMS operating system uses the character * for wildcarding portions of a numeric form user identification code. See [86, Section 4.3.4.1] for details.

The third ACL is an example requiring true separation of duties. JONES is allowed to execute the transaction, but only if he holds token capabilities proving that TP4 and TP7 have been previously executed and that TP4 and TP7 were executed by two people other than JONES and different from each other.

```
(IDENTIFIER=JONES+!.TP4+!.TP7,ACCESS=EXECUTE)
```

The syntax used in these three examples is based on the syntax for access-control-list entries (ACEs) used in the VAX/VMS operating system [86].[6]

Entries in the ACL of a TP must not be modified in arbitrary fashions. Rather than allowing modification based on a control-permission bit in the ACL itself or allowing modification by the "owner" of the TP, one must treat the ACL of a TP as a CDI itself and apply the same levels of enforcement and certification to the TPs that are allowed to modify the ACLs of other TPs.

10.1.4 Handling Groups of Users

The simple token capabilities of Section 10.1.3 do not efficiently describe many of the real requirements that could be present in a commercial separation-of-duty system. For example, it may be sufficient that two TPs are executed by any two people in a department as long as they are different people. The simple scheme described thus far would require listing on the access control list all the combinations of people in that department. Management of such lists would quickly become impossible.

Access control lists in systems like the VAX/VMS operating system already support a mechanism to describe groups of users by allowing three kinds of identifiers in an access-control-list entry. There are UIC (user identification code) identifiers that correspond to individual users; there are general identifiers that identify groups of users; and there are system identifiers that tag jobs as BATCH, INTERACTIVE, or NETWORK, etc. A given process in the system may hold several of these identifiers.[7] One would like to be able to require that a TP may not be invoked unless some other transaction had been invoked by anyone who holds some general group identifier.

Therefore, when the ACL is checked and the transaction field of a token capability matches, but the user field does not, one must check to see what other identifiers that user may hold. The protection checking routine compares each identifier against those required in the ACE. If a match is found, then that token capability satisfies that term of the boolean expression in the ACE.

[6]For this dissertation, I have extended the syntax to support the boolean expressions required for separation of duty. The actual VAX/VMS operating system does not have such extensions.

[7]The VAX/VMS operating system constrains the names of identifiers to be unique. Thus, there can never be confusion between a user identifier and a group identifier.

To support separation of duties, just as an ! is a wildcard matching any UIC that does not appear in a different term of the ACE, an ! followed by a group identifier means anyone who holds that particular group identifier, but does not appear in a different term of the ACE. Thus, this ACE specifies that the TP may be executed by JONES, but only if someone in the PHYSICS group has previously executed TP4 and someone in the CHEMISTRY group has previously executed TP7. JONES, the person in PHYSICS, and the person in CHEMISTRY must be three different people.[8]

```
(IDENTIFIER=JONES+!PHYSICS.TP4+!CHEMISTRY.TP7,ACCESS=EXECUTE)
```

10.1.5 Security Policy, Auditing, and Recovery Management

The audit trail described above is not just for security use. The same record of transactions is required to implement the *two-phase commit protocols* that are needed to provide database integrity in the presence of system crashes. As described by Gray [83], the two-phase commit protocol requires log records of all transactions to ensure that any transaction either completely succeeds or does not occur at all. These log records are exactly the records needed for security-policy decisions and for after-the-fact security audits.[9]

10.2 Related Work

There has been a variety of work related to commercial data integrity and separation of duties, besides the Biba [18] integrity model and the Lipner [144]

[8]Actually, they must hold three different UICs. A system manager could grant three UICs to the same human being, and the computer could not tell the difference. However, such assignment of multiple UICs to a single person is unwise, and the documentation for the system manager should advise against such practice. A user who must take on different roles at different times should be assigned additional identifiers that represent those roles, rather than multiple UICs.

[9]Integrating security policy with auditing and the two-phase commit protocols provides yet another benefit. One can use the two-phase commit protocol to assist in solving the *secure readers–writers problem.* The secure readers–writers problem derives from lattice security models, in which a shared database is read by processes at a high security level and written by processes at a low security level. All references to the database must read and write consistent data, but it is a violation of security policy to allow the low-level writers to know when the high-level readers are reading. The problem has been solved by Hinke and Schaefer [99] and by Reed and Kanodia [181] using eventcounts to time stamp updates to the database. A reader may read inconsistent data, but learns upon completion if a writer has interfered with the read. At that point, the reader must back up and retry the operation. It is the back-up-and-retry mechanism that the two-phase commit protocol already provides. The case of a writer interfering is merely another kind of event that requires retrying the transaction. Steve Lipner first suggested that two-phase commit could be used in conjunction with eventcount locking to make implementing retry easier.

commercial integrity model. Other authors have explored the Clark and Wilson model in ways somewhat different from those proposed in this chapter. This section reviews that related work and contrasts it with my proposals.

10.2.1 AAS

During the late 1960s, IBM Corporation developed an Advanced Administrative System (AAS) [233] to automate their order-entry systems. AAS incorporates an interesting approach [163] to the separation-of-duties problem in a very large commercial, transaction-processing environment. The problem facing the AAS implementors was that the system included a very large and changing number of TPs, and that the central administrators could not easily decide what separations of duties were actually required.

AAS solves this problem by implementing a *conflict matrix* to record separation-of-duty requirements. Each application, consisting of a number of TPs, has an *owner*. The owners are all required to identify which other TPs conflict with their TPs. For any given pair of TPs, TP_i and TP_j, if either owner reports a conflict, then no user will ever be authorized to run both TP_i and TP_j.

The management and procedural aspects of any security system cannot be minimized, and it appears that a conflict matrix of this type can be quite helpful in the enforcement of rule **E2**. For example, an IVP could be run whenever separation-of-duty access control lists were changed, specifically comparing the ACL entries with a conflict matrix. Other strategies may also be possible.

10.2.2 RSS

Herbert and Needham [97] proposed a registration-and-sequencing server (RSS) to provide a mechanism for ordering the flow of transactions in a local area network. The server would enforce a state-machine definition of the ordering of transactions and could enforce a policy much like Clark and Wilson's separation of duty. State transitions in RSS were controlled by capabilities, to ensure that transactions were not invoked by the wrong people or out of order. RSS was designed as a separate server, primarily to deal with the problem that many servers in the network may not have an inherently secure system. However, the RSS design did not preclude implementation using access control lists (ACLs) on individual objects, as proposed in this chapter.

10.2.3 Cascaded Network Connections

I proposed a mechanism [114] for forwarding authentication information between nodes in a network, using a central authentication server, such as Girling's [78], combined with a proxy-login mechanism. Essentially, the central authentication server would release token capabilities, much like those in Section 10.1.3. The

proxy-login mechanism, combined with an extended access-control-list mechanism provided secure access to network services. The authentication-forwarding mechanism particularly dealt with what I called *cascaded network connections.* A cascaded network connection occurred when a user invoked some network server, P, and in order to provide its service, P then invoked some other networked server, Q. The authentication information of the original user had to be passed along from P to Q to enforce security controls. Estrin [66] has developed a scheme for inter-organizational network security that similarly deals with cascaded network connections, using Biba's non-discretionary integrity model to limit information flows.

The Clark and Wilson model of separation of duties is clearly more powerful than protection based on cascaded network connections. Unifying the transaction-integrity model with network authentication forwarding should produce a simpler and at the same time more powerful tool for secure distributed processing. That unification is outside the scope of this dissertation, but should be an interesting area for future research when SCAP is integrated into a distributed environment.

10.2.4 Program-Integrity Policy

Shirley and Schell [195, 194] proposed a program-integrity policy that appears to address Clark and Wilson's rules **E1** and **C5**. Program integrity requires enforcement of two conditions:

Simple Program-Integrity Condition: If a subject has *modify* access to an object, then the program integrity of the subject is greater than or equal to the program integrity of the object.

Program-Integrity Confinement Property: If a subject has *execute* access to an object then the program integrity of the object is greater than or equal to the program integrity of the subject.

Simple program integrity seems to address the issue of transforming UDIs into CDIs in a protected fashion. Program integrity confinement deals with assuring that CDIs are only operated on by TPs and that TPs are protected. Shirley and Schell used program integrity to provide a formal basis for the use of protection rings to support a security kernel. They showed how protection rings provide a program-integrity policy and how that program-integrity policy allows a security kernel to enforce the lattice security model.

Shockley [196] has carried the program-integrity work forward and proposes a scheme using integrity categories and special *trusted subjects* to support the requirements of the Clark and Wilson model. Shockley's design is discussed in Section 10.2.7, together with a similar design by Lee.

10.2.5 Secure Committees

Rabin and Tygar [176] have proposed a mechanism they call *secure committees* to implement separation of duties. They use Shamir's cryptographic secret-sharing algorithm [193] to require that a minimum quorum of committee members agree on certain actions. The difficulty which this approach shares with most cryptographic operating-system security techniques is that the software to implement the cryptographic protection must be penetration resistant. Furthermore, the information must appear in plaintext at some time in order to be used, and effective operating system protection must be available at those times. Making all of that software penetration resistant is comparable in difficulty to constructing a secure kernel that can protect the data without the performance impacts of cryptographic algorithms.

10.2.6 Assured Pipelines

In 1985, Boebert and Kain [26] proposed an alternate to the Biba integrity model using the concept of *assured pipelines* in the Honeywell Secure Ada Target (SAT) machine.[10] The SAT processor implements a form of secure capabilities, similar to those in the SCAP secure capability architecture [119]. Assured pipelines provide a mechanism for ensuring that data of particular types can only be handled by specific trusted software. The SAT contains Domain Definition Tables (DDTs) and Domain Transition Tables (DTTs) that define the allowed operations of the assured pipelines. The DDT and DTT are defined to be static in the initial version of SAT, but dynamic forms could be equivalent to the token capabilities and separation-of-duty ACLs in this chapter.

10.2.7 Enforcing Clark and Wilson with Integrity Categories

Independently, Lee [136] and Shockley [196] (See Section 10.2.4.) have developed formulations of the Clark and Wilson model using Biba integrity categories and a mechanism that Lee calls *partially-trusted subjects*. A partially-trusted subject is allowed to transform data within some limited range of integrity access classes. By marking each class of CDI with a distinct integrity category and implementing TPs as partially-trusted subjects, both Lee and Shockley achieve the Clark and Wilson goals with purely a lattice model. Lipner anticipated this use of integrity categories in his 1982 paper [144], but his downgrading techniques were cruder. The partially trusted subjects are very similar to the protection domains of SCAP and of SAT. The only distinctions appear to be in implementation details.

[10] The SAT project has since been renamed the Logical Coprocessor Kernel (LOCK) [184].

A major difficulty with using integrity categories to protect CDIs and TPs is management. A large system, like IBM's AAS, may have thousands of distinct TPs, while most lattice model designs to date have considered sixty-four categories to be a large number. Furthermore, categories tend to be managed centrally by a single security authority, while the security policy in commercial transaction system probably needs to be more decentralized, with individual applications managers creating TPs and granting authorizations. My separation-of-duty ACLs are aimed at providing that level of management flexibility.

Managing a large number of categories is a difficult problem, but one might take advantage of the fact that relatively few category combinations are used, and the categories could be stored in a list format (much like an access control list), rather than in a bit-mask format as is usually done. See [116] for more detail.

10.3 Trojan-Horse Problems

As discussed in Section 10.1.1, formal verification of all the TPs and IVPs is very difficult and expensive. If a TP contained a Trojan horse (or even just a bug), then it could either process a transaction improperly or, more subtly, it could attempt to pass a capability for a CDI to someone else in the system who could then tamper with the CDI.

The capability-passing problem is precisely that with which the lattice security models and the secure-capability architecture of SCAP are designed to deal. The simplest solution is to run the TPs of a particular application in a separate security category (similar to the suggestions of Lipner's commercial integrity model [144]). The lattice model would prevent the TPs from directly passing capabilities (or indeed any other data) to applications that are not authorized that particular category. One particular TP in the application would be granted the right to sanitize information by removing the category. Thus, the proofs of confinement could be limited to a particular TP, rather than all the TPs of the application. The difficulty with assigning a category to each application is that most systems implement a relatively small number of categories. This is the same problem that Lee and Shockley would have in managing integrity categories and similar solutions would apply.

The issue of correctly processing the transaction itself remains one of proof of program correctness, but separating the security questions should simplify the proof process. In commercial environments, where TPs would be certified on basis of test results only, separating the security questions is even more crucial. The security questions are dealt with in the proofs of the security kernel. The separation-of-duty questions can be answered by inspecting the access control lists and category assignments. If the separation-of-duties requirements are simple, then the inspection may be possible by hand. Otherwise, IVPs would have

to automate the inspection, and those IVPs would themselves require certification. Even if the inspection requires automated certified procedures, it should be possible to separate the proofs of algorithmic correctness of the TPs from the proofs of inspection and from the running of the IVPs. This should simply the proofs by removing interdependencies.[11]

10.4 Performance Problems

While the design outlined so far is secure and meets Clark and Wilson's goals, it has one potentially severe problem—performance. The mechanisms proposed in [119] provided for capabilities whose use was limited by additional checks for access control lists and lattice-model constraints. The performance costs of those additional checks were considered minimal, because the checks would be performed only once. Thereafter, the fully authorized capability could be cached and used for most memory references. The commercial-integrity policy requires checking the access control lists and the token capabilities for every transaction. Such costs could easily become unacceptable.

The use of token capabilities to encapsulate the precise audit record required for a check eliminates the need for searching the audit trail to find the previous authorizing transactions. Such a search would have required examining all audit records of the system in reverse order and would have been very expensive.

Token capabilities reduce the performance cost of the security checks, but they still do not provide the level of caching that was suggested in [119]. However, the security mechanisms in existing database management systems require similar recomputation of access checks on each transaction.[12] Such checks on each transaction should not be a major concern, because they are overwhelmed by the performance costs associated with the database queries, themselves. Query optimization [50, Chapter 16] by compiling database requests into machine code and heuristically selecting optimal retrieval strategies is a major issue in database design. By comparison, the costs of security checks should be quite small.

10.5 Retrospective

This chapter has shown how the Clark and Wilson integrity model can be implemented using the secure capability architecture of SCAP. The Clark and Wilson

[11] It is possible that some application might require such a complex separation-of-duty policy that it could not be implemented with just token capabilities and access-control-list entries and would require special code in the TPs. Such a level of complexity would break down the proof separation. However, there is no evidence, to date, that such complex applications actually exist.

[12] Fernández, Summers, and Wood [69] and Date [51, Chapter 4] provide good tutorial introductions to database security systems.

model is of particular significance, because it is the first attempt to formalize the requirements of commercial security that appears to accurately model the real requirements. Earlier efforts, such as Biba's and Lipner's, while moving in the correct direction, did not completely describe the commercial needs.

The use of the audit trail as input to the access-authorization decision is a critical part of the model and is what most significantly differentiates the Clark and Wilson model from previous security models. The notion of token capabilities as an optimized way of referencing the audit-trail information appears to be critical to an efficient implementation.

It is most important to recognize that Clark and Wilson are solving a different security problem from that addressed by the lattice security models. The lattice models deal with protecting information from unauthorized release. The Clark and Wilson model deals with protecting information from unauthorized tampering.

There is one serious concern with the implementation strategy of token capabilities proposed in this chapter. The complexity of the mechanism, particularly when group identifiers are used in conjunction with separation-of-duty requirements, makes user errors quite likely. Simplicity of security mechanism is always very important to reduce the chance of human errors and to make audit analysis easier. The mechanism proposed here is quite complex, and a simpler one would be desirable.

The integrity category formulations do not appear any less complex to use. Indeed, some might argue they are more complex, given the difficulties of managing large numbers of categories. There remains a great deal of research to be done in this area before practical systems are implemented.

The Clark and Wilson model is at the same point with protection against tampering that the lattice models were in early 1973 with protection against leakage. The next few years of security modeling and implementation will show whether the model is strong enough to meet security requirements and can be implemented in a sufficiently simple manner to allow average users to comprehend it.

Chapter 11

Improved Revocation Algorithms

This chapter examines the need for immediate revocation, and then shows two new algorithms for implementing revocation that are much simpler and more efficient than existing algorithms. The two algorithms are appropriate for shared and unshared page tables, respectively. Although the prefered implementation of SCAP is with unshared page tables, both strategies are presented, so that SCAP could be implemented with either organization.

11.1 Need for Revocation

There is a frequent debate in operating system design about whether it is better to implement immediate revocation of access rights, interrupting current users of an object, or to apply changes in access rights only to new users who activate an object after revocation has happened. Immediate revocation has its benefits in removing an unauthorized user as soon as possible. However, immediate revocation has been complex to implement within the operating system, and it forces applications programs to include code to handle the case of suddenly losing access to data. Multics is one of the very few operating systems to actually implement immediate revocation. This chapter presents new algorithms for revocation that make implementation completely straightforward and that eliminate the complexity argument against immediate revocation.

Close examination of the interactions of storage-quota mechanisms and non-discretionary controls shows that immediate deletion of objects is essential to avoid certain types of storage channels [118, Section 8.2.4]. Immediate revocation is a prerequisite for immediate deletion, so a secure operating system must implement some form of immediate revocation.

Assume that storage quotas are used to limit the amount of on-line storage (presumably disk space) consumed by each user. A user may have many subjects

operating on his or her behalf at different access classes. The user may create objects at different access classes and therefore must have storage quotas at different access classes. In particular, an object at a given access class must be charged against a storage-quota account at the same access class, because when the size of the object changes, the storage-quota account must be both read (to determine if additional space is available) and written (to record the use of the space).

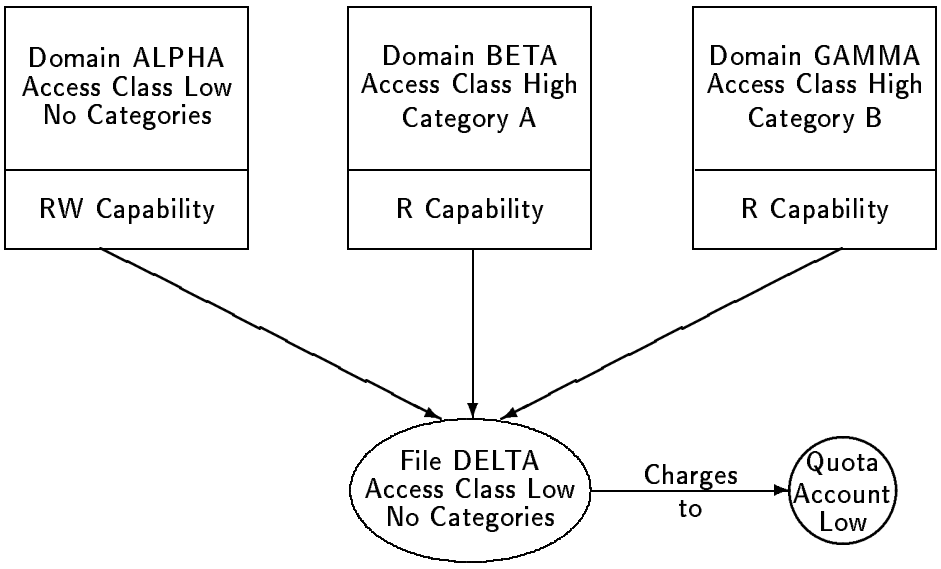

Figure 11.1: Quota Causing Storage Channel Problem

Figure 11.1 shows a simple example of the need for immediate deletion. File DELTA is classified at access class Low, with no categories. There are capabilities for the file held by three domains, ALPHA, BETA, and GAMMA. The three domains are at different access classes, such that the access classes of BETA and GAMMA are both higher than the access class of the file DELTA. Because they have different categories, the access classes of BETA and GAMMA are disjoint. Furthermore, file DELTA charges its disk space to a quota account at the low access class. (The quota account must be at the same access class as the file, because anyone who can either extend or truncate the file must be able to modify the quota account.) If domain ALPHA deletes its low-access-class capability to file DELTA, intending that the storage of DELTA be reclaimed, the problem arises. ALPHA is not allowed to know of the existence of the capabilities held by BETA and GAMMA, yet ALPHA can observe the value of the low-access-class, quota account. (For simplicity, assume that no other objects are charged to that quota account.) If the system did not immediately delete the file DELTA and reduce the charging to the quota account, then ALPHA could infer the existence of at least one other capability. A carefully arranged Trojan horse could use

this path to infer the existence of higher-access-class capabilities, by ensuring that file DELTA is private to the Trojan horse running at the high access class and the spy running at the low access class. The system could not upgrade the file DELTA to the level of the higher-access-class capabilities, as the remaining capabilities are disjoint. It cannot upgrade to the least common subset of the higher-access-class capabilities, namely access class High, no categories, because that would still leak information.

The system could eliminate the information flow by charging the storage usage of file DELTA, not to a single quota account at access class Low, but rather to a quota account for every existing capability at the access classes of those capabilities. Such a quota system would indeed eliminate the illegal information flow. It would also make the quota system useless for its intended purpose, accounting for the usage of available storage space. Further, if the sum of all allowed quotas were larger than the total available space, then the undesired information flow would reappear, not as changes in the values of the quota accounts, but as visible occurrences when the disk runs out of space.

Thus, immediate deletion of objects and therefore immediate revocation are required for an operating system to support both non-discretionary controls and storage-quota mechanisms. Whether immediate revocation is actually offered as a system feature is a question of marketing and user requirements. The operating system support must be present in order to avoid the illegal information flows.

11.2 Revocation Difficulties

Immediate revocation of access has always been viewed as a complex feature to implement in an operating system, and particularly in a capability system. Because capabilities for an object can be copied and stored in other objects, one cannot easily search for and invalidate all capabilities that grant access to a particular object. Even in a small system, a search of the entire file system would be very time consuming (neglecting any security implications of the search itself). If the system is distributed over a network, and capabilities could be held on other machines, then the search becomes totally impractical. Redell developed the first practical scheme for capability revocation through indirection [179]. This section will examine the complexity of the existing schemes for revocation as background to the new proposal for revocation with eventcounts.

11.2.1 Multics Revocation with Back Pointers

Multics implements revocation by maintaining a set of back pointers appended to the end of the *active segment table* as shown in Figure 11.2. The active segment table is a system-wide table that contains the page tables for each object currently active in the system. Whenever a process activates an object, the

process's segment descriptor for the object must be set to point to the correct page table in the active segment table. At the same time, Multics keeps a set of back pointers associated with each of the page tables, with one back pointer for each process that has a segment descriptor pointing to that page table.

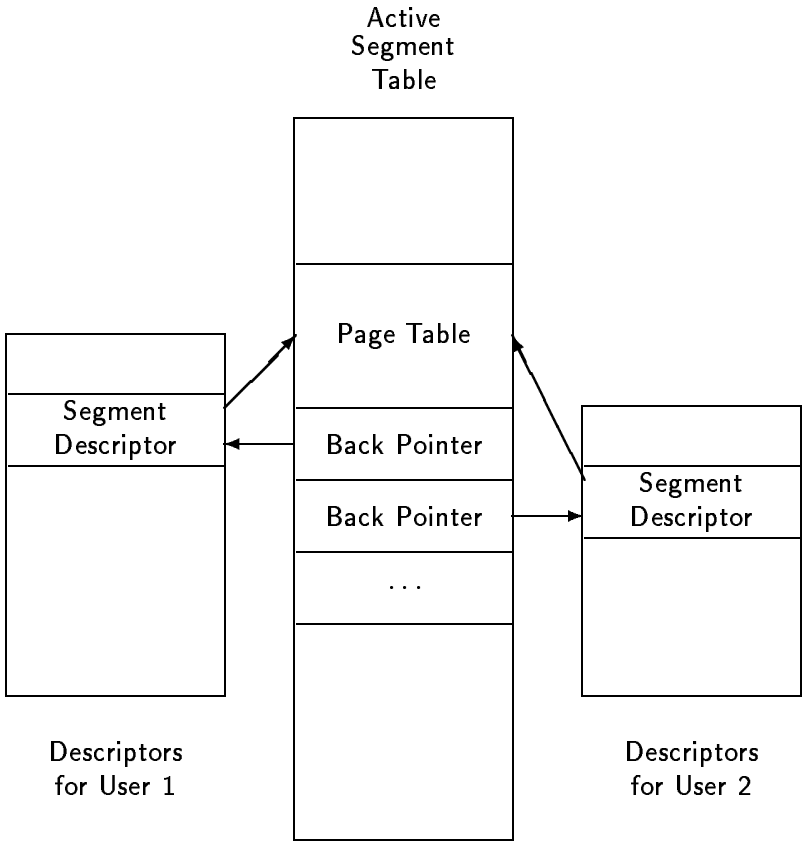

Figure 11.2: Multics Revocation Scheme

When access to a segment is revoked, the operating system goes to the active segment table entry and follows each of the back pointers, setting missing-segment faults in the appropriate segment descriptors. As soon as a process attempts to reference the segment, it will take a missing-segment fault. The missing-segment-fault handler will attempt to recompute the access rights of the object, and if access for this process has been revoked, then the process will receive an access-violation fault. Thus, the revoking process need only follow the back pointers and set fault bits. All other processes will recompute their access rights individually upon each process's next reference to the segment in question. Since Montgomery [157] has shown that the frequency of sharing is very low, the cost of setting fault bits and recomputing access rights will be quite low.

The difficulty with maintaining back pointers is that the operating system has no way of knowing how many processes may be have a given object in use.

As a result, the back pointers must be allocated dynamically from a pool of so-called trailer records. Managing the trailer-record pool can be complex, because it is possible to run out of trailer records before the active segment table is full. Running out of trailer records can cause a system crash and must be avoided. Further, running out of trailer records can constitute a storage channel in a security kernel.

It is certainly feasible to manage trailer records by forcing the deactivation of objects when the trailer-record space becomes full. However, a perverse program that activates the same object many times with different segment numbers could use all of the trailer-record space, even with no other objects available to be deactivated. A security kernel must handle even such perverse cases to avoid the possibility of a storage channel and to reduce the opportunity to deny service.

The security kernel could implement a trailer-record quota for each user. Then, each user would run out of trailers independently, avoiding both storage channels and system crashes. Such a quota mechanism would require administrative support to determine how many trailer records should be allocated for each user. Relating the usage of the quota to actual program behaviour would be very difficult, even for very sophisticated users, and would be nearly impossible for naïve users. The VAX/VMS operating system has already shown this difficulty with its large number of rather obscure, per-user quotas. While such quotas solve serious allocation problems in the operating system and prevent system crashes due to resource exhaustion, it is very difficult to explain to a user why his or her FORTRAN program requires more active-segment-table trailer-record quota or more buffered-I/O-byte-count quota.

11.2.2 Redell's Revocation with Indirection

Redell [179] introduced the concept of a revoker capability whose purpose was to support revocation of access.[1] Initially, the owner of an object would have a normal capability to the object. The owner could copy that capability and pass it to other users and would have no ability to later revoke access. However, if the owner of a capability creates a revoker capability to the object, then the owner could pass capabilities that point to the revoker capability, rather than directly to the object. The capabilities that the owner passes point indirectly through the revoker capability, before getting to the object itself. Then, the owner could revoke the capabilities that had been passed to other users by destroying the revoker capability through which the users are required to go to gain access.

Figure 11.3 shows an example of how Redell's scheme operates. The owner of an object has a capability for that object and wishes to grant capabilities to User 1 and User 2. Therefore, the owner creates two revoker capabilities for the

[1] Herbert implemented Redell's revoker capabilities in CAP-III [95, Section 10.4].

object, and two capabilities that point to the revoker capabilities. The owner gives those capabilities to User 1 and User 2. The owner must create two different revoker capabilities to be able to separately revoke access to User 1 and User 2. If the owner had created only one revoker capability, then revoking access to one user would revoke access to the other, as well.

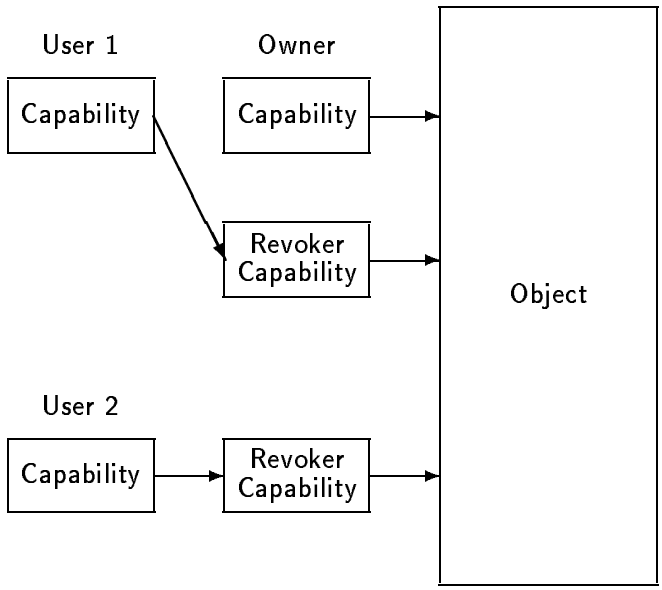

Figure 11.3: Redell's Revocation Scheme

Redell's revocation scheme can be applied recursively. Figure 11.4 shows how User 1 could then decide to grant a capability to User 3 for the same object as before. User 1 creates a revoker capability and grants User 3 a capability that points to the revoker capability. Now, User 1 can revoke User 3's rights to the object, and the object's owner can revoke User 1's rights as well as User 3's rights. Indeed, the owner may not even know of User 3's existence.

Redell's scheme suffers from two problems. First, the indirection through a large number of revoker capabilities could adversely affect system performance. A properly designed translation buffer, however, could cache the results of the capability indirections and make most references go quickly. The translation buffer would have to be cleared whenever a capability was revoked, but such revocations are quite infrequent. Second, Redell's scheme is complex, because it requires different kinds of capabilities (normal and revoker capabilities). It also requires applications to plan ahead about revocation and be sure to grant only the proper kinds of capabilities. This level of complexity appears in the applications programs themselves, and is likely to result in subtle security errors when those programs are implemented by other than security professionals. Run-

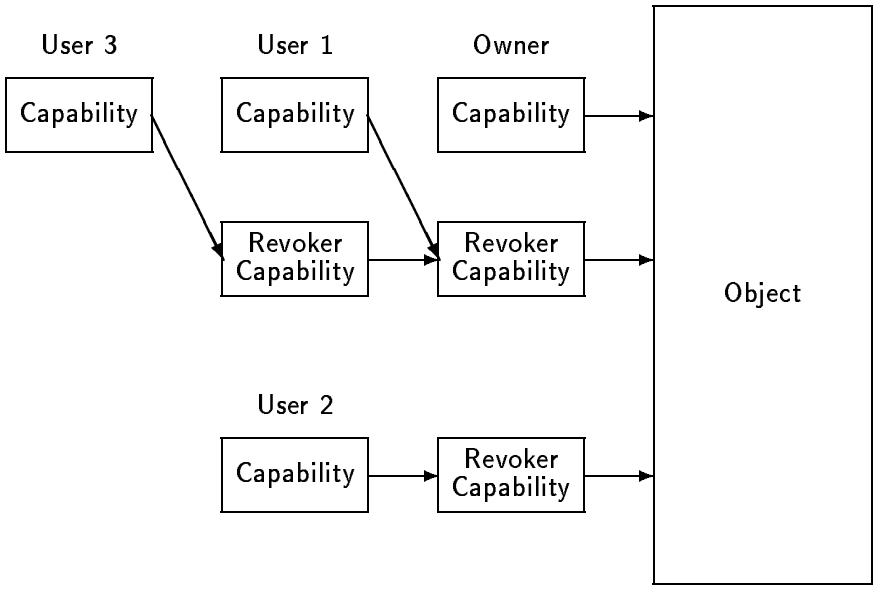

Figure 11.4: Recursive Redell Revocation

time libraries could hide some of the complexity, but a simpler scheme with less indirection would be preferable.

11.3 Revocation with Eventcounts

I have developed a new scheme, called *revocation with eventcounts*, that avoids both the complexity of Multics' back pointers and the complexity and indirection of Redell's revoker capabilities. Instead, the new scheme uses Reed and Kanodia's *eventcounts* [181] to identify activation instances of an object. Revocation with eventcounts is most appropriate for systems in which one set of page tables is shared among all users of a shared object. This is the type of paging used in Multics, as described in Section B.2.

Objects that are currently in use are called *active objects*, and the page tables for the active objects are stored in an operating system data structure called the *active object table (AOT)*.[2] There is one AOT entry for each active object. Each AOT entry contains the page table of the object and an eventcount for the revocation algorithm. All the capabilities for an object contain pointers to the single AOT entry for that object. Each time an object is activated or when the object's access rights are changed, the eventcount in the object's AOT entry will be incremented.

Each capability for an object contains a copy of an eventcount value. When the capability is first used, the eventcount value from the active object table

[2]The active object table (AOT) is the counterpart of the Multics active segment table.

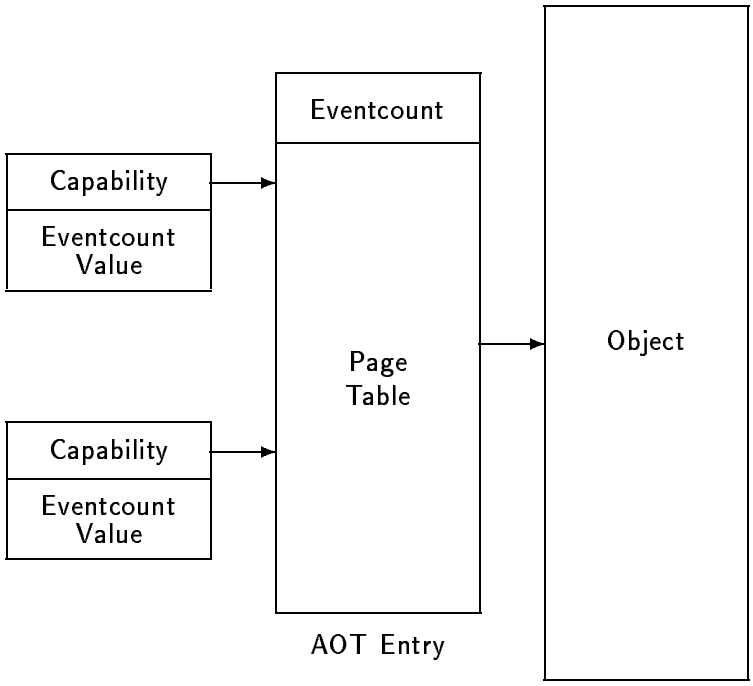

Figure 11.5: Revocation with Eventcounts

entry is copied into the capability. Note that the eventcount value is stored in the capability, but is not loaded into the translation buffer. When the access rights of an object are changed or the object is to be deactivated, the system increments the eventcount in the AOT entry and flushes the translation buffers of all processors in the system.

When the processor takes a translation buffer miss, it compares the eventcount values in the capability list and in the AOT and notices that either the object has been deactivated or that access rights must be recomputed. Recomputation must be done by every process sharing the object, but Montgomery's results [157] show that the frequency of actual sharing is quite low. Therefore, the actual number of recomputations required is small.

Comparing eventcount values in every translation buffer miss can be expensive, because translation buffer misses are very frequent events. However, the comparison need only be made on segment descriptor misses. Segment descriptor misses occur much less frequently than page descriptor misses. The eventcount comparison could be done in processor microcode, but would be easier to implement with a software TB miss handler. See Section 15.3 for a detailed discussion of filling the TB from software.

Adding eventcounts to each capability also increases the memory usage for capability lists. Because memory costs are declining much more rapidly than processor costs, this tradeoff of increased space for reduced complexity should be

worthwhile. Since the eventcounts are not needed in the translation buffer, the only cost is increased usage of primary memory.

The net effect of this new revocation strategy is to achieve Multics-style revocation with significantly less operating system complexity.

11.4 Revocation by Chaining

The strategy of revocation with eventcounts depends on there being a single page table for an object, even if that object is shared among several users. Multics is a good example of a system with shared page tables. However, if the page tables are not shared, as in the VAX [138] or the System/38, then the eventcount scheme will not work. For this type of processor, the operating system must know where each of the page table entries that maps a particular page is located, so that page can be removed from memory.

In VAX/VMS, shared pages are allocated in a special software structure, called the *global page table* [124, Section 14.3], that contains pointers to all the page table entries that map a particular shared page. These pointers are analogous to the trailer records in Multics, and are just as difficult to manage. In particular, shared memory must be specially declared by the programmer, and quotas limit how many shared pages may be used, so as to limit the number of pointers.

Revocation by chaining is the new scheme for revocation with unshared page tables. It avoids the complexity of both the Multics trailer records and the VAX/VMS global page tables, although at the cost of increased memory usage. Revocation by chaining is implemented by logically extending each page-table entry to add a chain field. The chain fields link together all page-table entries (PTEs) that map the same physical page. The chain fields are set up in a circular queue, such that the last PTE's chain field points to the first PTE in the queue. When a page is first brought in, it is not shared, and the chain field is set to point to the PTE itself. Each subsequent time the page is mapped, the chain field of the previous PTE is set to point to the new PTE, and the chain field of the new PTE is set to point to the first PTE. Thus, given any PTE that maps a page, the operating system can quickly find all other PTEs that map that same page, simply by following the chain. Since most pages are never shared, most chains will contain only one entry. Figure 11.6 shows four PTEs chained together. The PTEs might be in page tables for different address spaces or some or all of them might be in the same page table, if the page had been mapped to several different virtual addresses. (The multiple mappings could occur if a capability were copied or refined, in preparation for passing as an argument.)

The drawback of revocation by chaining is that it can double the size of the page tables. For example, PTEs in the VAX architecture are 32 bits long, and the chain field would have to be another 32 bits. On a machine with a large

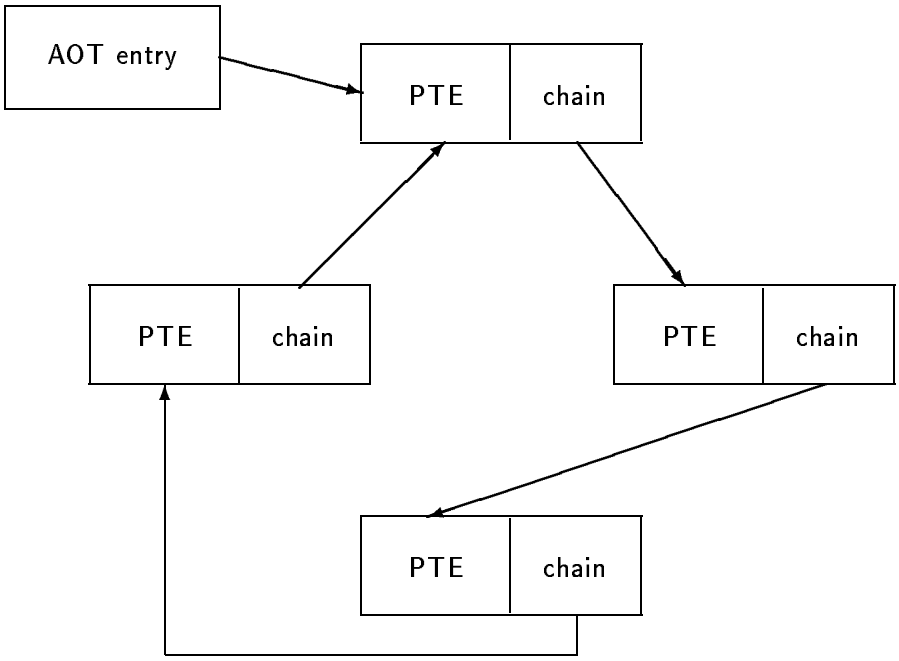

Figure 11.6: Chained Page-Table Entries

virtual address space and a small page size, such as the VAX, this growth in the page table size could be unacceptable. Memory costs are declining, however, and with larger page sizes, revocation by chaining might be quite acceptable. More significantly, if the processor used a hashed page table, then revocation by chaining becomes very attractive. Chapter 16 discusses how unshared, hashed page tables are preferred for SCAP, and Section 16.5 specifically covers revocation by chaining in a hashed page table.

Chapter 12

Secure Garbage Collection

This chapter shows how the system-wide garbage collection schemes that most capability systems implement cause fundamental problems with the lattice security models. After examining and rejecting several unacceptable solutions, I show how a payment-for-resources scheme, such as provided by the Monash University password-capability system or the Amoeba system, can provide secure, automatic storage management for SCAP. None of these techniques is new. What is new is combining non-discretionary quota cells with rent collection to solve the security problems present in garbage collection.

12.1 Source of the Problems

Most existing capability systems support some kind of system-wide garbage collection system to delete unreachable objects automatically. Such garbage collection can greatly simplify the programmer's job by eliminating the need to explicitly delete objects. This section shows, however, that system-wide garbage collection cannot be securely implemented in a system that also supports non-discretionary access controls and on-line storage quotas.

In systems that implement the lattice security model, it is common to find low-access-class objects that are shared by many users of the system. For example, compilers and text editors are objects that are shared by all users. As a result, domains at many different access classes may hold capabilities for the shared objects. Any object's disk space must be charged against an on-line storage quota of the same access class as the object. Otherwise, a writer of the object who extended the length of the object would have a storage channel available for illicit communications.

If there is an object with many capabilities at high access classes, but only one capability for the object at the low access class, and if a low-access-class subject deletes the capability for the object, then the low-access-class subject can determine if the garbage collector later deletes the object by watching the on-line storage-quota account to which the object is charged. When the account

value changes, the object has been deleted. (To avoid complications, a malicious user could arrange that no other objects are charged to that account.)

Changes to the storage-quota-account values can be caused, not just by the actions of the low-access-class subject, but by the actions of high-access-class subjects that may also have capabilities for the object. As long as those high-access-class capabilities exist, the garbage collector cannot delete the object. As a result, an information flow exists from the high-access-class subject to the low-access-class subject that is in violation of the non-discretionary controls. That flow exists even if the garbage collector is guaranteed to be correct and Trojan horse free.

The garbage-collection problem is closely related to the revocation problem. However, the garbage-collection problem appears insoluble in its stated form, while the revocation problem is one of complexity and performance. Section 12.4 discusses alternatives to system-wide garbage collection that can be made secure, but that vary in how many of the benefits of garbage collection can be preserved.

12.2 Solutions that Do Not Work

The information flow might be avoided by upgrading the object when the last capability at the access class of the object were deleted. However, just as in the revocation case in Section 11.1, if the remaining capabilities for the object are from incomparable access classes, then there is no way to select a class to which to upgrade the object without creating an information flow between the two incomparable access classes.

Thus, based on these arguments, it appears impossible to simultaneously implement a system-wide garbage collector, an on-line storage quota system, and non-discretionary security controls. Since both non-discretionary controls and storage quotas are essential to SCAP, the system-wide garbage collection must be omitted. The next sections will examine alternatives to system-wide garbage collection that retain at least some of the benefits of garbage collection.

It is essential to note, however, that only system-wide garbage collection that crosses non-discretionary access-class boundaries is ruled out. Applications that function entirely within a single access class (such as a LISP implementation) may freely implement garbage collection. Only system-wide garbage collection creates the security problem.

12.3 Quota Management

The most obvious solution to the garbage-collection security problem is not to implement system-wide garbage collection, but to simply to rely on the disk-quota mechanism. This solution has the obvious drawbacks that the user must

explicitly delete objects that are no longer required and that objects can become lost. Furthermore, unless carefully designed, the disk-quota mechanisms themselves can have information flow problems.

12.3.1 Multics Quota Problem

The Multics, disk-quota mechanism is a good example of how information-flow problems can occur. Multics disk quota is allocated to directories in the file system. Segments charge their disk usage to their parent directory's quota. A directory may have either a terminal or a non-terminal quota. A terminal quota on a directory means that the quota charges against this directory are charged here. A non-terminal quota on a directory means that any quota charged to this directory should actually be charged to its parent directory. Thus in Figure 12.1, the disk usage for segments D, E, and F are all charged to directory A.

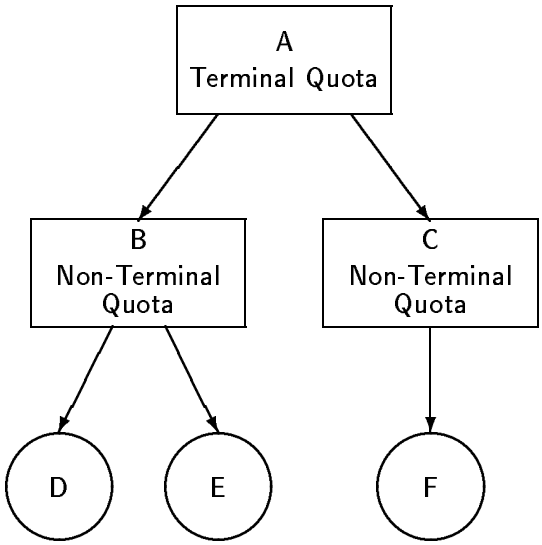

Figure 12.1: Multics Quota Example

The security problem arises if the non-discretionary access classes are not all identical. If the access class of directory C is higher than the access class of its parent directory A, then disk usage by any of the children of C (such as F) will be visible at the access class of A, and an illegal information flow will occur. (Note that by the compatibility property of the Multics file system, the access class of the children of a directory must be greater than or equal to the access class of the directory.) In Multics, any directory whose access class was strictly greater than the access class of its parent was called an *upgraded directory*. The Multics Access Isolation Mechanism (AIM) solved the information flow problem by requiring that upgraded directories must always have terminal quotas and by strictly controlling how quota was allocated to upgraded directories. (See

Whitmore [229] for details of the AIM disk quota solution.) The drawback of this solution is that managing quota for upgraded directories becomes significantly more complex for the user.

12.3.2 Quota Cells

Mason [151] proposed a new quota system for Multics, in which each segment charges its quota usage to a distinct quota cell. Quota cells were not necessarily tied to the directory structure, and both available quota and usage charges could be moved from one quota cell to another. The storage channels in Multics quota can be completely eliminated by assigning access classes to particular quota accounts and by charging the quota usage of a segment to a quota cell of the same access class.

The VAX/VMS operating system handles disk quota very similarly to Mason's proposal. It stores the name of a disk-quota cell in the file header of every file, rather than tying disk quota to the directory hierarchy. VAX/VMS currently assigns one quota cell per physical device to each user. If each user were given a quota cell for each access class in which the user held files, then the quota charging would have no information flow problems. Specifically, the user would only have to allocate quota to access classes, through a secure server mechanism to avoid any remaining storage channels, but such allocations would not have to be done frequently.

12.4 Payment Systems

The only solutions to the garbage-collection, information-flow problem discussed thus far involve elimination of garbage collection and are therefore satisfying only to security fanatics who are unconcerned with usability requirements. However, the Monash University password-capability system [7] implements an alternative to garbage collection, called *rent collection* that maintains many of the benefits of garbage collection, yet can be made storage-channel free.

The Monash system defines a notion of *virtual money* that is used to pay for resource usage. Objects in the Monash file system must contain a capability for an account of money. Periodically, a rent collector inspects all the objects in the file system and deducts the rent from the each object's accounts. Any object whose account has gone to zero is considered garbage and may be deleted.

The Monash system designed rent collection as a convenient way of achieving automatic object deletion without having to find all references to objects or having to maintain reference counts. Instead, the user of an object is expected

to keep paying the rent, as long as he or she is interested in retaining that object.[1] (Rent could either be paid in advance or in response to a low-money interrupt.) As a result, the Monash rent collector is much simpler than a typical garbage collector, particularly in a distributed environment.

Rent collection has one other advantage that its designers did not anticipate. When combined with the secure, quota-cell system, it provides a storage-channel-free mechanism for automatic reclamation of storage. An object would charge its rent against an account of money at the same access class as the object. When the account went to zero, the object would be automatically deleted. If some other user were interested in using the object, that other user could pay extra virtual money into the account from a process at the level of the object. The user could then use the object from a process at a higher access class, without fear that the object would be deleted and without creating any information flow problems.[2]

Mullender has further examined the ideas of virtual money in his design for the Amoeba bank service [162, Chapter 7]. His proposals for virtual bank accounts and virtual currencies (with virtual exchange rates) seem compatible with the Monash rent collection and with assigning a distinct access class to each virtual bank account. The next research step (not part of this dissertation) will be to actually implement a system using rent collection to see if the user interface will be acceptable, even with the complication of multiple access classes.

[1] In practice, the system would have to allow some grace period before actually deleting an object. A user might have forgotten to pay the rent, or the system might have been down for a lengthy period of time.

[2] Having one user pay the disk usage bills of another user makes the actual accounting of who is using what disk space more complex. However, the system managers only wish to limit total disk usage to that which is available. In this case, the first user's ability to consume disk space is reduced by exactly the amount transferred to the second user.

Part IV

Improving Performance

Chapter 13

Performance Overview

Part (IV) of this dissertation deals with obtaining acceptable performance from SCAP. This chapter is a brief overview of what the performance problems are and how they are addressed in Chapters 14 through 19.

13.1 Performance Problems

The principal reason that capability systems have not been widely successful is that their performance has usually not been as good as that of non-capability systems implemented with the same technology. While capability systems do offer certain significant advantages over non-capability systems, such as support for non-hierarchical protection domains, the market for those benefits has not been large enough to overcome the performance disadvantages. Indeed, only one capability-based system can be said to have achieved commercial success, the IBM System/38. The System/38 is not sold as a capability system, but rather as a small business machine to run programs written in the RPG III programming language. The capability structure is used as an aid to implementing systems programs and to support a single-level store incorporating a relational database management system.

What is required is a tradeoff among the complexity of the implementation, the performance of the implementation, and the actual requirements of real programs. Since real programs rarely use the more complex features, those features can be safely implemented entirely in software, reserving hardware mechanisms for assisting only the most performance-critical functions. Furthermore, non-security sensitive mechanisms may be implemented at compile-time, rather than at run-time, achieving still further performance gains.

The most dramatic example of performance problems in a capability-based system has been the Intel 432 processor. Various benchmarks [88][1] showed the In-

[1]The methodology of the benchmarks in [88] has come under severe criticism [140] for improper measurements of VAX-11/780 performance. In particular, the effects of compiler op-

tel 432 to perform much slower than the comparable, VLSI technology Intel 8086 processor.[2]

Colwell undertook a major study [44] of the Intel 432 to determine where the performance costs came from and to propose alternate implementations to alleviate the problems. I have used Colwell's results and proposals to examine how a secure capability-based system could be implemented to achieve maximum performance.

13.2 Applying RISC Technology

Current CPU and memory technologies have led to a better cost/performance ratio for CPUs implemented with relatively simple architectures. These simpler architectures have been termed *reduced instruction set computers* or *RISC* machines [172]. RISC machines are typically characterized by simple, uniform-sized instructions, larger register sets, and memory operations that are restricted to loads and stores. RISC machines achieve their improved cost/performance ratios over complex microcoded machines, because of the decline in cost of fast memories [230]. It is possible, today, to build extremely fast primary memories and cache memories that run at speeds that previously had to be restricted to small microprogram stores. As a result, RISC architectures have user-accessible instruction sets that are similar to microinstruction sets, and compilers that generate such code directly.

However, much of the early work on RISC technology ignored the impact of operating system issues, such as process switching, memory management, and security, on the performance of the overall computer system. While the more recent commercial RISC designs, such as the MIPS Computer Systems chip [161], the Hewlett-Packard Spectrum [19], the IBM RT PC [98], the Acorn RISC machine (ARM) [206, Section 4.3.3], and the Am29000 chip [110] have considered operating system design issues, those considerations have largely been limited to how best to implement a conventional operating system, such as UNIX.

The key contribution of this dissertation to improving capability-system performance is the recognition that RISC technology can provide major performance benefits to the SCAP architecture. While motivated by RISC technology, this part of the dissertation also looks at how the performance of SCAP on a CISC (Complex Instruction Set Computer) can be improved. Some of the techniques apply to both classes of processors, and others apply only to RISC

timization (or lack thereof) were not factored out of the performance results. Indeed, Colwell's later study of the Intel 432 [44] showed lack of compiler optimization to be one the major causes of the poor performance.

[2]The Intel 432 and the Intel 8086 were both implemented in Intel's HMOS process. The Intel 8086 was a single chip with 29,000 transistors [122]. The Intel 432 was a three chip set with approximately 100,000 transistors per chip [178].

machines. Chapter 15 discusses optimizations using improved translation-buffer design. Chapter 16 examines hashed page tables as a method to simplify demand paging in conjunction with capability addressing. Chapter 17 discusses optimization of cross-domain calls, both for CISC and RISC designs, with experimental results shown in Chapter 18. Finally, Chapter 19 shows how the RISC optimizations for memory management and for cross-domain calls lead to an approach for handling interrupts and exceptions for real-time processing.

Chapter 14

Programming Generality

14.1 Costs of Programming Generality

One of the major causes of poor performance both in capability-based systems and in other advanced operating systems has been adherence to the doctrine of *programming generality* when crossing protection boundaries. Programming generality was first espoused by Dennis [57] as a means of achieving reusability of software by defining strictly modular interfaces and ensuring that routines do not depend on how they are called. In a sense, Dennis' programming generality was an ancestor of today's ideas of modular programming and object-oriented programming.

Schroeder and Saltzer [192, p. 158] extended Dennis' concept of programming generality by adding the requirement that "it must be possible to change the protection environment of a program or a group of programs without altering the internal structure of the program or group." It seems logical that programming generality should extend across protection boundaries. However, attempting to provide such generality has caused severe performance and complexity problems in a number of system designs, and worse still, has led to certain classes of security vulnerabilities that might not have otherwise existed.

14.1.1 Argument Validation

The Multics argument-validation mechanism is an excellent example of programming generality leading to trouble. Argument validation is used to counter a variety of penetration attacks in which a caller of a more privileged domain passes the address of an argument. The address is supposed to point to memory locations that belong to the caller, but the address actually points to memory locations that are not accessible to the caller. As a result, the more privileged domain is tricked into either revealing secret information, or improperly modifying its own database, or both.

An example of an argument validation attack on the GE-645 version of Multics is shown in [121]. Multics included a complex and time-consuming argument validation routine that was run on every supervisor call. That argument validator could be tricked by using some of the highly complex addressing modes of the GE-645 that allowed infinite indirection with auto-incrementing of any of the indirect addresses.

Rather than eliminating the complex addressing modes, the next version of the Multics processor, the Honeywell H6180, added a mechanism that checked the origin of any pointer before allowing its use [192]. For a limited set of constructs, an address passed to a domain would be checked against the access rights of the caller, rather than the access rights of the called domain.[1] The H6180 hardware argument validation, however, suffered from several drawbacks. First, it added complexity to the hardware and slowed down every memory reference. Second, pointers in memory and address registers had to be larger to contain the ring numbers that were always checked. Third, it was not complete and did not actually meet the requirements of programming generality. Programs still had to perform argument validation checks, because a malicious caller could still attack in a variety of ways that the Multics hardware did not check. Arguments that consisted of complex data structures could be malformed, even if the pointers all were legal. Worse still, the caller might asynchronously modify the arguments after the called domain had checked them, but before they had actually been used. Several attacks of this sort are described in [70]. The defence against such attacks is to copy arguments to safe storage prior to validation.

Although the Multics argument validation mechanism added a great deal of complexity to the processor and reduced the performance of every memory reference, it did not achieve the stated goal of programming generality. Programs that were the target of cross-ring calls had to be coded much more carefully than ordinary programs. The expectation of programming generality led many programmers to make serious mistakes, because they incorrectly expected that complete argument validation was being done automatically.

By contrast, if each protection domain has its own unique address space, and parameter pointers are passed only as capabilities, then argument validation becomes much simpler. Incoming capabilities are mapped into different locations of the address space of a called domain. If the calling domain passes a parameter for which the called domain already possesses a capability, those two capabilities will appear at different locations in the address space with different access rights associated. No special argument validation is required at all. Further, it is not meaningful to embed a virtual address in a memory segment, because the calling domain does not know where the argument will be mapped. Therefore, it is simple to procedurally ban the passing of virtual addresses between protection

[1] This checking worked even when the address was passed to a third domain.

domains. Instead, the interface specifications should call for passing arguments either by value (in registers) or by abstract datatype reference, that is, by a capability to a memory segment or domain.

Argument validation is made simpler and faster by ensuring that each domain's address space is distinct and forcing aliasing in the address of arguments. This argues against unique-ID addressing, in which all protection domains run in the same universal and extremely large address space. See Section 15.2.4 for other security arguments against unique-ID addressing.

14.1.2 Procedure Calls in the Intel 432

The implementation of procedure calling in the Intel 432 is another example of programming generality taken to extreme. The Intel 432 contains a cross-domain call instruction that, while very powerful, is quite expensive to execute.[2] The Ada compiler for the Intel 432 treats all procedure calls uniformly and always generates the cross-domain call instruction. The difficulty with this is that most procedure calls do not cross protection boundaries, but are to internal procedures of the same protection domain (of the same *package*, in Ada terminology). Colwell points out in [44, Section 4.2.1.3], however, that internal procedures do not require the protection checking of a full cross-domain call, because the compiler can perform the checking, either at compile time or in generated code. Thus, the internal procedures could have been implemented assuming the same protection domain, and the Ada compiler could have generated much less expensive call instructions, such as *branch_and_adjust_stack* or *branch_intersegment*, rather than always generating the very expensive cross-domain-call instruction. Colwell's benchmarks indicate that such a change could result in a performance improvement of over 20%. Colwell concludes that this was simply an example of poor compiler optimization. Organick [169, Section 4.4.2], however, describes the use of the cross-domain-call instruction for all calls as a benefit, because the context records of all procedure invocations are uniform. Whether due to excessive programming generality or insufficient time or resources to do better compiler optimizations, the result was to diminish performance.[3]

[2] Colwell [44] discusses where many of the machine cycles go in implementing the cross-domain call instruction and suggests various ways to improve its performance. Chapter 17 will discuss further performance improvements in the design of cross-domain calls.

[3] The Multics procedure-call instruction was similarly very expensive, because it supported crossing protection ring boundaries. However, the Multics PL/I compiler could optimize internal procedure calls, although it did not optimize calls to external procedures in the same protection ring.

14.2 Costs of Capability Refinements

Refinement, as defined in Section 4.1, is very simple to implement in capability systems that do not support demand paging. In a system like CAP [231], a capability contains the physical address of base of the object in memory and the length of the object. A refinement operation simply builds another capability by adding some value to the base field and subtracting some other value from the limit field. The resulting refined capability is no more expensive to use than the original capability. Figure 14.1 shows a simple example of a refined capability for the middle portion of a large object. The refined capability also has had its access rights restricted from read-write to read-only.

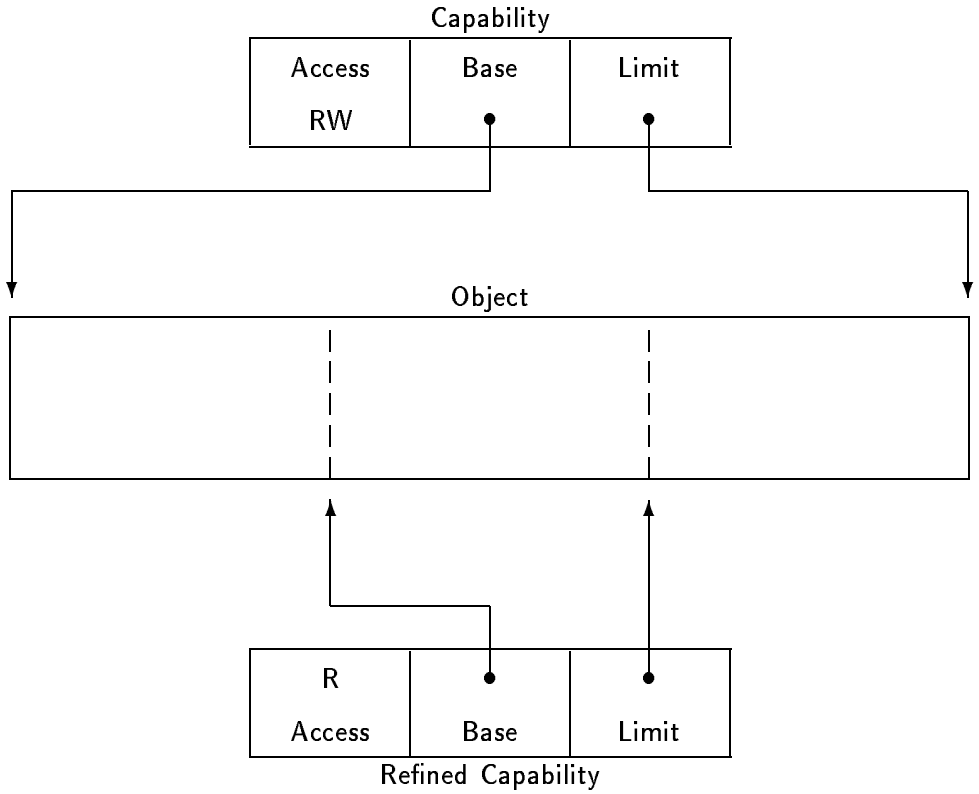

Figure 14.1: Simple Refinement Example

If the capability system supports demand paging, then the cost of refinement becomes significant. A paging system (including those with both segmentation and paging) does its address translation by splitting the virtual address into several fields and using those fields as indices into the page tables. For example, a VAX virtual address is 32 bits long. The two high-order bits are used to select the address region: P0, P1, or S0. The middle 21 bits are used as the index into

the page table for the region, and the low 9 bits are used as the offset within the selected page. (See Appendix D for more details on the VAX architecture.)

By contrast, a system with segmentation but not paging (such as most capability systems) does its address translation by adding the virtual address to the base field of the capability and comparing the virtual address against the length field. Address translation in a system with segmentation, paging, and refinement requires adding the base, comparing with the length, and then all the field splitting and indexing as well. While some or all of the field splitting and indexing can be done in parallel with the base and length checking, there will be at least a cost in extra hardware complexity and chip area, if not actual performance degradation on every memory reference. Bishop [22, Section 5.3] suggests one scheme for implementing such a combination of objects that start on arbitrary addresses with a linear demand-paged virtual memory. His scheme appears to allow the length check to be computed in parallel with the page-table lookups and access-control checks. However, the complexity of his address-translation hardware appears significantly higher than that in VAX processors. Since Bishop's design has never been implemented, no accurate assessment can be made of its actual costs.

This apparent problem of excessive cost for refinements comes from approaching the problem with programming generality as a principal goal, that all references to objects go through the same mechanism. The problem is that some references, namely those that are properly page aligned, can be much less expensive than other fully general references. Examining how refinements are actually used can offer much more efficient solutions to the problem

14.2.1 Uses of Refinements

In existing capability systems, such as the CAP, refinements are used for two principal purposes: storage management and argument passing. The CAP operating system [198] allocated segments to users by mapping large pieces of primary memory into a single segment and then computing a refinement. SCAP does not need refinements for storage management, because there is an underlying paging mechanism. Indeed, it is the underlying paging mechanism that has led to concerns over the costs of refinements.[4]

Refinements are also important for securely passing arguments between mutually-suspicious subsystems. When one domain wishes to pass a block of storage to another domain, it wants to pass only those bytes that are to be used and no

[4]It could be argued that with memory costs dropping so dramatically, perhaps there is no longer a need for paging, and one could simply use segmentation. Certainly, if primary memory were infinite, then there would be no external fragmentation problems to avoid. However, demand for address space seems to be increasing at a pace comparable with memory technology advances, so the likelihood of external fragmentation disappearing is small. Certainly page sizes need to increase significantly, but fixed allocation units still appear advantageous.

others. Similarly, if a called domain returns a capability to a block of storage, the calling domain should not gain access to any bytes other than those intended. Forcing arguments to be aligned on page boundaries can result in unnecessary argument copying. Further, there will be memory wasted due to internal fragmentation of the pages. Memory costs are coming down very rapidly, however, so the wasted memory seems not to be serious.

The copying issue is more significant, because most refinements that are passed as arguments in CAP are used for I/O purposes. If unaligned refinements were not supported, subsystems such as the VAX/VMS Record Management System (RMS) would be unable to transfer records into the user's work space with a single copy operation. Assuming that the user specified a non-page-aligned location in a READ statement, and assuming that the record retrieved from disk was not page aligned (likely for variable-length-record files), then RMS would have to copy the record into a page-aligned data structure and then the user's language-run-time system would have to copy the record again into the user's variable. Further, it would be impossible to support RMS locate mode I/O [85, Section 8.2.1], in which RMS returned a pointer to the actual buffer, rather than copying the record at all.

14.2.2 Restricting Refinements

Ideally, the cost of refinements should only be incurred when using a refined object. Other references to objects should run at full speed. This goal can be achieved by making refined objects a special data type and allowing only certain special instructions. The SCAP processor supports the following instructions to deal with refinements: REFINE, EXTRACT, and INSERT.

The REFINE instruction takes four input arguments: a capability for a data segment (which could itself be a refinement), a base address, a length, and an access specifier (read-only, read-write, etc.). It returns a refined capability for the sub-segment specified by the base and length. REFINE checks that the base and length lie within the existing segment. The refined capability gets the requested access rights, restricted by the original rights, but has its data type set to *refined-data-segment*, rather than *data-segment*. A refined-data-segment capability is just like a data-segment capability, but the segment can only be operated on by the REFINE, EXTRACT, and INSERT instructions. If the beginning and end of the refinement are page aligned, then REFINE will return a normal capability.[5]

The EXTRACT and INSERT instructions are very similar to the Move Character (MOVC) instruction in the VAX architecture. They copy bytes out of or into a refined segment, starting at a specified location for a specified length. If

[5]This implementation of REFINE requires that access rights be specified on a per-page basis.

the destination length is longer than the source, then a padding character is used to fill the remaining bytes of the destination. EXTRACT and INSERT check the base and bounds fields of the refined capability and refuse to read or modify bytes that are outside the refinement.

EXTRACT and INSERT can be implemented either in software or in microcode. In software, a refined capability is mapped into the address space with no access in user mode, but full access in kernel mode. To actually reference the refined object, kernel calls to the EXTRACT and INSERT operations must be made. The EXTRACT and INSERT operations could be coded explicitly in the application to gain maximum performance (after testing to see if the object is a non-page-aligned refinement, of course). Alternatively, non-paged-aligned refinements could be made completely transparent by implementing a handler for access-violation faults that would invoke the EXTRACT and INSERT operations on behalf of the application, just as MicroVAX systems include fault handlers to emulate unimplemented instructions. The choice between explicit coding and fault handling depends on the actual frequency of refinements in the particular application in question and both options should be made available to the programmer.

If EXTRACT and INSERT are expected to be used a great deal, then the cost of a standard kernel call may be excessive. In such a case, the instructions could be implemented in microcode or in a highly optimized kernel call, such as the extracode routines found on the Atlas [127].

Chapter 15

Translation Buffers

The address translation buffer (TB) is the most performance-critical portion of a memory management architecture. It remembers recently-used memory descriptors, so that the CPU need not fetch segment and/or page table entries repeatedly. Without a translation buffer, a memory reference on a virtual memory machine could require from two to four memory cycles, compared to a single reference on a non-virtual memory machine. With a good translation buffer, miss rates of 0.1% to 2.0% are possible [199], making the average number of memory cycles required to complete a fetch only slightly more than one.

Many of the trade-offs in the design of effective translation buffers are dependent on the particular hardware technology used for implementation. As a result, it is impossible to specify exactly which features of a translation buffer are essential to a high-performance, secure system. Instead, this chapter develops a taxonomy of the various types of translation buffers and indicates the impact of security on their design. Although very important to translation buffer performance, the level of associativity is not covered in this chapter, as the associativity design choices have little bearing on security. However, Appendix C contains a brief tutorial on the subject, as the level of associativity of the VAX-11/730 translation buffer became an important issue in the prototype implementation, described in Chapter 18.

15.1 Hardware-Visible Segmentation

Whether segmentation should be visible to the hardware is the first and most significant option in translation-buffer design. Segments, in principle, must be able to start and end on arbitrary boundaries and must be separately protected from other segments. In a machine without paging, the translation buffer would simply contain segment descriptors, each consisting of a base address, a bounds register, and a set of access rights. Address translation adds the base to the virtual address, compares the virtual address against the limit, and checks the access rights. These three operations can be performed in parallel, so the check

can be quite fast. Segmentation without paging can be very inefficient, however, due to external fragmentation and the need for frequent recompaction of physical memory.

The translation buffer can combine segmentation with paging in one of two ways. Either the translation buffer can store segment descriptor entries, as well as page descriptors; or the translation buffer can store only page descriptors, with the operating system (or the microcode) arranging the page descriptors to the segment-descriptor requirements. The former approach (with segment descriptors in the translation buffer) offers automatic limit checking on data structures, while the latter approach (using only page descriptors) can only check addresses to the granularity of the page size.

15.1.1 Segment Descriptors in the Translation Buffer

Hardware array-bounds checking requires that segment descriptors be present in the translation buffer. Colwell [44, page 23] argues strongly that array-bounds checking is an extremely useful function of a segmented memory, and that checking to the granularity of pages is insufficient. However, simple base-and-bounds checking detects only a small fraction of possible addressing errors. For example, if a segment consists of a linked list, the offsets forming the forward and backward links must not only lie within the bounds of the segment, but they must also point to the beginning of valid entries in the list, rather than to the middle of entries. The applications designer has two choices: either take object-oriented programming to the limit and store each entry of the list in a separate segment, or program in additional checks on the links. If list elements are small and numerous, placing each list element in a separate segment could be extremely expensive. By contrast, an object-oriented programming language can perform some of the necessary checks at compile time and generate code to perform the balance at run time.

15.1.2 Segment Descriptors in Software Only

If the translation buffer holds only page descriptors, then the address-translation function is much simpler and can be made faster. The translation buffer simply looks for an entry corresponding to the virtual address. Assuming that a match is found, the physical page number is concatenated with the offset within page from the virtual address. Note that concatenating bit fields is generally simpler and executes faster than the addition and comparison needed for a base-and-bounds check. Access permissions, of course, can be checked in parallel. If no match is found, then the translation-buffer fill mechanism must be invoked, as discussed in Section 15.3.

Bounds checking on data structures, as opposed to bounds checking required for security, can better be done by the compiler, either by flow analysis on the source code, proving that bounds violations cannot occur, or by including run-time checks in the generated code. If the processor were pipelined with branch prediction, the run-time bounds check could execute at least partially in parallel with the references to the data.

Bounds-checking cost should not be paid as part of every memory reference. Instead, memory references should run as fast as possible, and more complete argument validation and reference checking should be included explicitly in the code of the program and only where needed.

15.2 Context Switching

When the CPU must switch between one address space and another, the translation buffer must ensure that entries from the old address space are not confused with entries from the new address space. This section examines four techniques for dealing with this context switching problem: translation-buffer flushing, translation-buffer swapping, address space numbers, and unique-ID addressing.

15.2.1 Flushing

The simplest approach to the context switch problem is to flush the entire contents of the translation buffer on every context switch. Flushing on context switch ensures that only entries from the new address space will be present in the translation buffer. There are two drawbacks to flushing the translation buffer frequently. First, the cost of flushing itself can be very considerable (hundreds or thousands of CPU cycles), because the translation-buffer hardware must be optimized for access time speed rather than flushing speed. As a result, most translation buffers do not have hardware for clearing entries, and flushes must be implemented by microcode loops that write invalid entries into the translation buffer. The translation buffer must be built from the highest performance memory available. The extra logic to automate flushing would likely be expensive to implement and would likely introduce at least one extra gate delay in the most performance-critical part of the entire CPU, the memory-access path.

The second drawback of flushing is that when the processor switches back to the old address space, the translation-buffer entries will have to be reloaded. Depending on the size of the translation buffer and the mean number of instructions between context switches, this reloading overhead can be considerable. Clark and Emer report [41] a mean headway between context switches of 5,100 to 8,600 instructions for typical VAX-11/780 timesharing workloads. Their results indicate that a significant fraction of translation-buffer misses are due solely to context

switching and that those misses cannot be reduced by simply increasing the size or level of associativity of the translation buffer. Similarly, Schroeder [191] reports that over half the translation buffer misses in Multics are due to complete flushes.[1] Despite these costs, complete flushing of the buffer remains the most effective technique for many configurations, particularly if there are relatively few translation-buffer entries.

15.2.2 Translation-Buffer Swapping

One could reduce the cost of refilling the translation buffer after a context switch by saving and restoring all the buffer entries as part of the context switch itself. To make such swapping worthwhile, one would need special hardware to transfer the entire translation-buffer contents to and from primary memory in less than the sum of all the normal refill times. The cost of such specialized, high-speed hardware is likely to be prohibitive. Thus, translation-buffer swapping is not likely to be worthwhile, at least with present memory technologies.

15.2.3 Address Space Numbers (ASNs)

The cost of refilling the translation buffer after context switching can be reduced by storing entries from more than one address space simultaneously. Essentially, the tag in the translation buffer would be extended by an address space number (ASN) that would be assigned by the operating system to each distinct address space.[2] The CPU would provide a current address space number on every search of the translation buffer. Assuming that the operating system switches from one address space to another and then back to the first, entries could remain in the translation buffer to be re-used after the switch back. Of course, as Clark and Emer point out [41], the translation buffer must be quite large to make address space numbers worthwhile. Declining memory costs, even of the very high-speed memory chips, make such large translation buffers more feasible.

An address space number would be assigned dynamically to each process, as it was scheduled to run. To prevent the tag in the translation buffer from getting too large, the number of ASN's must be limited. (A limit of 256 ASN's or an 8-bit extension to the tag might be reasonable for current technology.) Since there can only be a limited number of ASN's, the system will eventually run out of them. If the translation buffer supports selective flushing, then the operating system could choose particular ASNs to be reassigned. Such selective-flush hardware

[1] Multics flushes the translation buffer after page faults as well as after context switches, so this number is significantly larger than for the VAX.

[2] ASNs were used first in the University of Manchester MU5 computer [158, 159]. Both the CAP-I [223, Section 5.4] and CAP-III systems [95, Section 12.5] use the CAP capability unit to provide the equivalent of ASNs by storing capabilities from more than one protection domain simultaneously.

requires parallel searching and is therefore likely to be expensive. Instead, the operating system could completely flush the translation buffer and recycle all the ASNs. Such flushes would occur much less frequently than context switches.[3]

15.2.4 Unique-ID Addressing

The address space numbers discussed in the previous section are invisible to the user. They are assigned by the operating system and exist only in the translation buffer. Alternately, one could make the address space numbers visible to user software by making segment numbers unique during the life-time of the system. In this case, all users share a common address space, and pointers are universal. Systems that use unique-ID addressing include the IBM System/38 [100] and a Data General machine that has not been marketed [52, 29].[4]

This technique of unique-ID addressing is attractive, because it eliminates many problems of translating addresses from one context to another. It has several drawbacks, however. First, if an object is shared between two users who have different access rights, the translation buffer must differentiate between those users, and a single TB entry for the object may not be sufficient.

Second, sequentially-assigned unique IDs can be used by a Trojan-horse program as a storage channel to violate confinement requirements [131, 143]. The storage channel can be eliminated by encrypting the sequential IDs with a strong encryption algorithm and a key known only to the operating system. Predicting the next unique ID would then be equivalent to a known-plaintext attack on the algorithm. Depending on performance needs, the operating system could either use decryption or a hash table to locate the object corresponding to a given encrypted unique ID.[5]

Third, the use of unique ID addressing forces the processor to use a much larger virtual address (128 bits or more), even though the actual address-space requirements of user programs are much smaller. A wider virtual address makes all of the addressing logic of the processor bigger, more costly, and slower.[6] While address spaces larger than 32 bits are likely to be needed in the near term, a jump to 128 bits, solely to provide unique addresses, does not seem justified. For

[3]Prof. David Wheeler has pointed out that the frequency of flushing could be reduced even further if the operating system maintained a count of the number of TB entries currently in use by any particular address space. Such a count could be maintained, either by the translation buffer load function returning the ASN of the entry that was replaced or by the operating system duplicating the TB insertion algorithm and keeping a map of the TB in primary memory. Of course, maintaining such counts would add to the cost of processing TB misses, and that cost would have to be traded off against the reduction in the frequency of TB flushes.

[4]The Data General processor that used unique-ID addressing appears to have been described as the FHP machine in *The Soul of a New Machine* [125].

[5]Prof. David Wheeler suggested encrypting sequentially-assigned unique IDs.

[6]Fabry [67] describes how such addressing logic might be implemented.

example, the Data General unique-ID machine uses 160-bit addresses in its most general addressing mode. These are shortened to 46-bit non-unique addresses for most data paths through the processor.

Fourth, unique-ID addressing makes argument validation more difficult. This problem is covered in Section 14.1.1.

15.3 TB Fill in Software or Hardware

Most processors today implement translation-buffer filling in microcode or in special hardware logic. However, some RISC architectures view a translation-buffer miss as a fault to be handled by software, just as page faults are handled by software.[7] Certainly, the logic to translate a virtual address is complex, and the software instructions to implement that translation should run at close to microcode speeds in a RISC machine. The frequency of translation-buffer misses is high enough that this hardware/software decision must be made carefully. The MIPS Computer Systems designers argue strongly for software filling.

> The advantage is simplification of bus interface, pipeline stall, and exception-handling, areas well-known to be troublesome. The chip area saved offers more TLB[8] entries, and O.S. flexibility is preserved for both UNIX variants and non-UNIX systems. The price is a slightly longer refill time than might be done by dedicated hardware. Since the total TLB penalty is the product of miss penalty times miss rate, it can be kept reasonable by lessening the miss rate .... [55, p. 143]

However, the IBM RT PC designers argue just as strongly for hardware refill.

> Memory management units that depend on processor intervention are much slower due to the overhead in passing control to the processor which must then save and restore registers and return control to the memory management unit in addition to the page table memory lookup function. [222, p. 62]

Given these two conflicting opinions, the choice between software and hardware resolution of TB misses is not clear-cut. The page-table structures and the size and structure of the translation buffers of the IBM RT PC and the MIPS Computer Systems chip differ greatly. Any of these factors could have accounted for the differences between software and hardware TB-fill performance seen in the two processors. Furthermore, some of the costs of context switching that IBM RT PC designers fear could be alleviated by dedicating a special register set to the TB-miss handler.

[7] Handling translation buffer misses in software originated in the University of Manchester Atlas computer. [126]

[8] MIPS uses the term, *translation lookaside buffer*(TLB) where I use *translation buffer*(TB).

Software TB fill does offer significantly more flexibility to the operating system designer, as the MU5 designers pointed out [159, p. 129]. The life of a typical operating system is many years, and during that time, both changing software requirements and changing hardware technology may make the operating system designer wish to change the page-table structures. With hardware TB fill, the page-table structures are fixed by the processor architecture, and the operating system designer has no flexibility. With software TB fill, the operating system designer can choose any page-table structure. Indeed, the same CPU could support two different operating systems with radically different page-table structures. A time-sharing system might use a multi-level-table approach, such as described in Figure B.2. In contrast, a real-time control system might use a single-level page table to minimise the contribution made by address translation to interrupt latency times.

There is one further argument for software TB fill. The revocation with eventcounts algorithm described in Section 11.5 is much simpler to implement if TB filling is done in software. Note that revocation with eventcounts would only be implemented on a machine with hardware segmentation and shared page tables. On machines with unshared page tables, like the VAX, revocation would be accomplished with chained page-table entries, as described in Section 16.5.

15.4 Shared-Memory Multiprocessors

The algorithms discussed so far in this chapter assumed a single processor system. If the system consists of several symmetric multiprocessors with shared memory, then certain changes are required. The paging databases will require locking to prevent simultaneous updating of data structures. The design of such locking strategies is beyond the scope of this dissertation.

More critical to the hardware design is the requirement for translation-buffer consistency across processors. If a page table entry is changed, not only must it be flushed from the current processor's TB, but it must also be flushed from all other processors' TBs.

This section presents the concept of a *snoopy translation buffer* as an improved method for multiprocessor-translation-buffer management.

15.4.1 Flush with Interprocessor Interrupts

Multics solved the TB-consistency problem by completely flushing the TBs of all processors whenever any entry was changed. Flushing was accomplished by sending an interprocessor interrupt to all CPUs and waiting for a response. While this approach assures consistency, it leads to many more TB flushes than are required, because most pages are not shared [157]. Further, such frequent flushing would likely make the address-space-number mechanism ineffective.

Flushes can be reduced by using the VAX translation-buffer-invalidate-single (TBIS) function. When a page table entry is changed, the CPU does a TBIS, and inserts the virtual address and address space number into a queue for other CPUs. The first CPU then sends interprocessor interrupts to the other CPUs which inspect the queue and perform the TBIS operations. A large number of interprocessor interrupts are still required, even though most often, the translation buffers on other CPUs do not contain the entry in question.

The number of interprocessor interrupts can be reduced for systems without shared page tables. For such systems, a particular page table entry can only appear in the translation buffer of a particular CPU, if the process that owns that page table entry has been recently scheduled on that CPU. If the scheduler keeps a list associated with each process of on which processors the process has run, then interprocessor interrupts need only be sent to those CPUs that could possibly have the page table entry. In a system with a large number of processors, such that any given process will only have run on a small fraction of the processors, this strategy will significantly reduce the number of interprocessor interrupts. Note that the definition of *recently scheduled* is that the process has been scheduled on a particular CPU since that CPU's translation buffer has been completely flushed. Complete flushes will occur when address space numbers are periodically recycled, as discussed in Section 15.2.3.

15.4.2 Snoopy Translation Buffers

The interprocessor interrupt approaches, discussed in the previous section, all potentially affect software performance, because many CPUs will have to handle the interrupts, yet often the relevant page table entry may not be present in their translation buffers. Thus, one CPU invalidating a single entry may cause a large number of CPUs to take action.

A *snoopy translation buffer*, analogous to the *snoopy cache* [123, 9, 208] that has become popular in multiprocessor design, can avoid this interrupt traffic. Snoopy caches watch bus traffic to detect writes to shared blocks. A snoopy translation buffer similarly watches bus traffic for translation buffer invalidations.

Each translation buffer has a port connecting to the interprocessor bus. When an entry is invalidated, the buffer also broadcasts the virtual address and address space number (ASN) to other translation buffers. When a translation buffer invalidate operation comes by on the bus, each translation buffer performs an invalidate cycle, using that virtual address and ASN. Note that for this strategy to work, ASNs must be assigned on a system-wide basis, rather than separately for each CPU.

The Fairchild CLIPPER 32-bit microprocessor implements a form of snoopy translation buffer on the CLIPPER bus [42, Sections 3.5.3 and 7.4.2.2]. A CPU can flush a translation-buffer entry for a specified page, either within the local

CPU only or in the translation buffers of all CPUs attached to the bus. The CLIPPER memory management units do not support ASNs, so the broadcasted translation-buffer invalidations would only be correct if all the processors were running in the same address space. For a snoopy translation buffer to be useful, either the processors must support ASNs, or all processors must be in the same address space.

Chapter 16

Hashed Page Tables

Capability-based systems, such as SCAP, tend to use their virtual address spaces sparsely, as separately protected objects must be stored in separate segments. Sparse use of the virtual address space can lead to extremely large page tables or to multiple levels of pages, such as implemented in the Multics processor. This chapter examines the use of *hashed page tables* (also called *inverted page tables*) as a method for making the size of page tables a linear function of the amount of physical memory, rather than a function of the amount of virtual address space used. Hashed page tables were first used on the IBM System/38 computer.[1]

This chapter first presents the design issues associated with hashed page tables and then presents my experimental results of implementing hashed page tables in the microcode of the VAX-11/730. The chapter assumes basic familiarity with the demand paging schemes of Multics and the VAX. Appendix B contains a brief tutorial on those and other schemes.

16.1 Implementing a Hashed Page Table

The IBM System/38 [100] and the IBM RT PC [98] each support large, sparse virtual-address spaces. The System/38 supports an address space of 2^{48} bytes with a page size of only 512 bytes. Instead of multiple levels of page tables, the System/38 hashes the virtual address into a table whose size is proportional to the amount of physical memory on the processor. The IBM RT PC uses a very similar technique to map an address space of 2^{40} bytes.

Figure 16.1 shows the data structures required to perform the following address-translation algorithm: A hash function is computed on the virtual address. The result of the hash is an index into a hash table. The hash table entry, in turn, contains a pointer to the appropriate page table entry (PTE) in the

[1]While the schemes discussed in this chapter all assume that the hash table is stored in primary memory, Thakkar and Knowles have proposed [209] an all hardware implementation, based on an extension of the University of Manchester MU6-G computer.

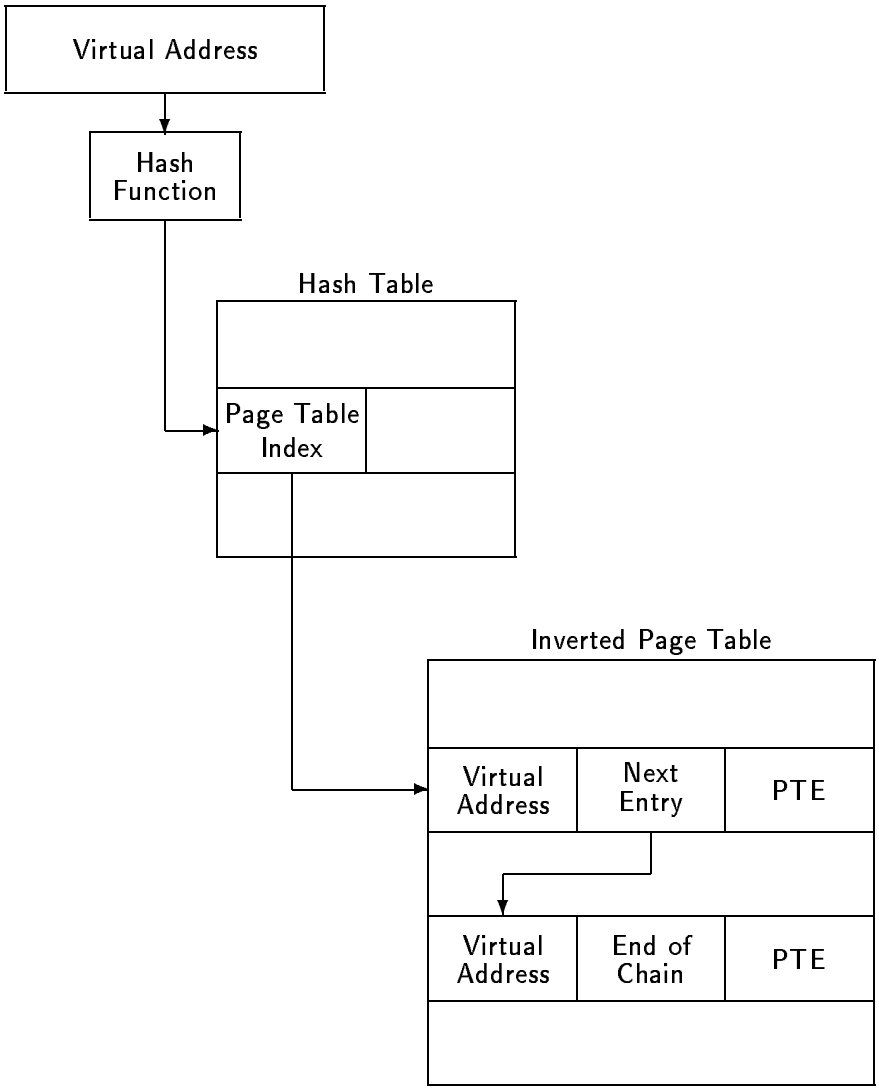

Figure 16.1: IBM System/38 Hashed Address Translation

inverted page table (IPT). More than one virtual address could hash to the same entry in the hash table. When a collision occurs, the additional PTE is stored at some other location in the IPT, and a chain is constructed from the first entry through all entries that hash to the same location in the hash table. The hash table and the inverted page table are stored separately, so that it is easy to add or delete entries without reorganising the hash table.

Performance of the hashed address translation depends on three things: the probability of collisions (or equivalently, the average number of hash-table lookups required to translate an address), the speed of the hash function computation, and the number of memory references required to translate an address, assuming no collisions.

16.2 Probability of Collisions

Probability of collisions is the first determinant of hashing cost. Assuming that the hash function produces uniformly distributed results and that in the steady state, all pages of physical memory are in use, then, as derived by Morris [160], the average number of probes, N, required for an IPT of size P, and a hash table of size H is given by:

$$N = 1 + \frac{1}{2\frac{H}{P}}$$

Thus, if the hash table is twice the size of the IPT, the average number of probes is 1.25. The number of entries in the IPT is exactly the number of pages of physical memory. If the amount of memory devoted to page tables and related data structures is to be kept to a fixed percentage of primary memory, it is possible to derive how large the hash table can be. In the case of the VAX, with a page size of 512 bytes, and a PTE size of four bytes, a hash-table entry requires four bytes, and an IPT entry has an overhead of 8 bytes[2] in addition to the PTE. Thus, if the hash table is twice the size of the IPT, each page uses 20 bytes, or 3.9% of primary memory. If the processor had a larger page size, then the ratio would be much better. With a page size of 4,096 bytes, only 0.5% of primary memory would be used for paging structures.

It is important to note that the amount of memory used for paging structures in this system is a constant. There is no need to limit the size of a virtual address space just to limit the amount of page tables required, as is commonly done in both VAX/VMS and UNIX. Sparse use of virtual address space does not incur penalties. The only limit on virtual-address-space size is the amount of disk available for paging and/or swapping.

16.3 Hash Function Costs

The time required to compute the hash function is the second determinant of hashing costs. Gehringer [76] suggests the use of polynomial division, as described in [128, pp. 512-513]. Polynomial division, however, is complex to implement. The IBM System/38 uses a simpler hash function that divides the high bits of the virtual address into three fields, and computes the exclusive-or of the two high fields with the reverse of the low field.[3] This function should produce a

[2]For the hashed page table to be actually used for all processes without the need for swapping hash tables, the virtual-address field in the IPT must be extended by an address space number (ASN), just as for the translation buffer. (See Section 15.2.3.) At first glance, adding an ASN field would increase the size of an IPT entry. The bits representing an offset within a page are not required in the IPT. Those bits can be used to store the ASN, instead.

[3]The size of the fields selected from the virtual address will determine the size of the hash table. Varying the hash table size will requiring varying the size of the particular fields used in the hash algorithm.

uniform distribution, assuming a large number of small objects, or a small number of large objects, or a mixture of both. Reversing the bits of the low field can be implemented in hardware at no cost in performance. In a processor lacking such hardware, the reversal could be implemented by a table look-up or omitted entirely.

16.4 Number of Memory References

The number of memory references required to translate an address, assuming that no collisions occur, is the third determinant in the cost of hashing. The IBM System/38 and the IBM RT PC both separate the inverted page table from the hash table and resolve collisions by chaining. This strategy makes it easy to invalidate entries when a page is removed from primary memory. Translating an address, however, requires three memory references: one to read the page-table index from the hash table, one to read the virtual address from the IPT, and one to read the page-table entry from the IPT. While the two reads from the IPT can probably be optimized into a single double-length read[4], the read from the hash table cannot be so optimized.

If the collisions were resolved using open addressing with linear probing [128, pp. 518–521], then the number of memory references can be reduced. Figure 16.2 shows a simpler hashed page-table structure that requires only two memory references for most translations. Just as before, a hash function is computed on the virtual address. The result is an index into the hash table. Unlike the chaining case, the hash-table entry contains the page-table entry (PTE) directly. Assuming the virtual addresses match, the PTE can be immediately loaded into the TB with no further memory references. Collisions are resolved by incrementing the hash table index by one entry and trying the comparison again. Because collisions are designed to be rare, the vast majority of TB misses can be filled by two memory reads. Because the two reads are for adjacent locations, the memory controller can probably optimize the reference into a single double-length read, just as in the chaining case.

The principal drawback of linear probing is that removing entries from the hash table is much more difficult. If several entries have collided, and one wishes to remove one of them, then several entries may have to be shifted up. In the case of a page table, however, removing entries from the table is much less frequent than translation buffer misses, so the extra time needed to remove entries should not be excessive. Therefore, using a linear-probing scheme for collision resolution

[4]For example, the VAX-11/730 [217, p. 6–46] can read a 32-bit longword from memory in three microinstructions: one microinstruction to initiate the read, one unrelated instruction while the memory controller performs its functions, and one to transfer the data. However, the VAX-11/730 can read a 64-bit quadword in only four microinstructions, adding one microinstruction at the end to transfer the second 32-bit longword.

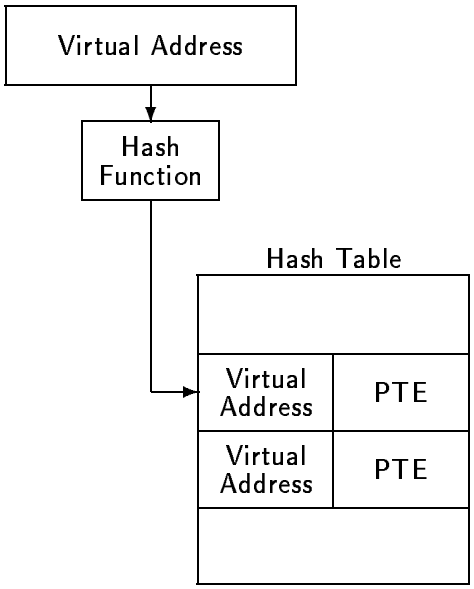

Figure 16.2: Hashed Address Translation with Open Addressing

should give better overall performance than the chaining technique used in both the IBM System/38 and the IBM RT PC.

Linear probing also has the disadvantage that as the table gets full, entries tend to cluster around particular points, raising the collision rate above what is predicted by the simple analysis in Section 16.2. Clustering is caused by a large number of search keys that are grouped consecutively. Knuth [128, pp. 520–526] suggests several possible solutions including incrementing the hash table index not by one, but by a constant c that is relatively prime to the number of entries in the table, or by using a second hashing algorithm to resolve collisions (called double hashing). Fortunately, clustering only becomes a problem when the hash table is nearly full, and the hashed page table will never be more than half-full. Therefore, the added complexity of double hashing will not be needed.

Note that the linear-probing approach also uses less memory than the chaining approach. Assuming that the virtual address and page table entry each require four bytes and that the hash table has twice as many entries as there are pages of physical memory, then only 16 bytes are used for each physical page, compared with 20 bytes in Section 16.2. For a page size of 512 bytes, this means that 3.1% of primary memory is used for the hashed page tables. For a page size of 4,096 bytes, the ratio drops to 0.4%.

16.5 Revocation by Chaining

As discussed in section 11.4, the revocation-with-eventcounts strategy depends on a shared page table to function. In a machine with inverted page tables access rights are stored in the page-table entries. Therefore, two processes sharing the same page cannot share the same page-table entry.

I introduced the strategy of revocation by chaining as a simple way to handle revocation for unshared page tables, but the strategy had the serious deficiency that it could use a very large amount of memory, proportional to the size of the virtual address space. That deficiency vanishes in a system with an inverted page table, because the space is proportional to the size of the physical address space, rather than the virtual address space.

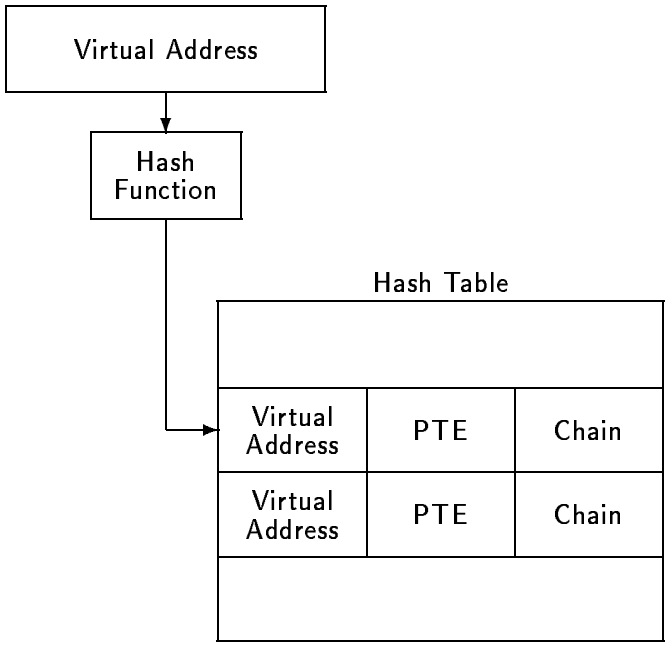

Figure 16.3: Hashed Address Translation with Shared Page Chains

The chaining pointer is allocated with each entry in the hash table. Using these pointers, the operating system constructs a linked list of all page-table entries that map to a particular physical page. Extra pointers in the hashed page table are easy to manage, because they are allocated once, at system-startup time. No special quotas or other mechanisms are needed, so the code to manage them should be straightforward. Figure 16.3 shows hashed address translation with open addressing, as in Figure 16.2, but with the chain pointers added. Adding the chain pointers increases the hash table size to 24 bytes per physical page. For a page size of 512 bytes, this means that 4.7% of primary memory is used for hashed page tables. For a page size of 4,096 bytes, the ratio drops to 0.6%.

16.6 Hashing Experiment

This section describes an experiment of implementing a hashed-page-table design for SCAP on the VAX-11/730. The design supports hashing for process-space pages (P0 and P1 spaces) and for capability extensions (S1 space). However, system space (S0 space) is still handled with a system-space page table, just as it is in a normal VAX. The rationale for not hashing S0-space addresses is discussed below in Section 16.6.5.

16.6.1 Structure of the Hashed Page Table

There is only one hashed page table for the entire system. It contains page table entries (PTEs) for all currently active domains. Each PTE is identified by an address space number (ASN) to distinguish it from PTEs belonging to other domains. The hash table is located in contiguous physical memory. When the processor detects a miss from the translation buffer, the microcode computes the hash function of the requested virtual address to give the entry number within the hash table. The microcode then multiplies the hash-entry number by the hash-entry size and adds the contents of the hash-table-base register. The result is the physical address of the initial probe point.

The microcode next fetches the hash-table entry to check if the high 23 bits of the virtual address and the ASN match the hash-table entry. If they do not match, then a collision has occurred. Collisions are resolved by open addressing with linear probing, as described in Section 16.2. The microcode adds the hash-entry size to the physical address and tries again. The collision-resolution search is terminated when the microcode finds an entry containing a termination mark. If the microcode reaches the end of the hash table during a collision-resolution search, it wraps around to the beginning of the table. In the unlikely case that a search completely wraps around and returns to the original probe point, the search will terminate. When the collision resolution search terminates for either reason, the microcode generates a translation-not-valid exception to the operating system. (See Section 16.6.4 for a discussion of how entries are removed from the hash table.)

Note that the number of entries in the hashed page table is determined by the hashing function. The size of the hash table, however, must be a function of the size of primary memory (Houdek and Mitchell [100] suggest twice as many hash entries as pages of physical memory.) Thus, the hashing function must change as the size of physical memory changes. For the SCAP experiments, a constant size of physical memory is assumed.

This hashing algorithm assumes that one hash-table entry corresponds to one page of physical memory. However, the performance of Berkeley 4.2bsd UNIX was enhanced by simulating a larger page size of 1,024 bytes, compared with the

VAX hardware page size of 512 bytes [11]. Berkeley 4.2bsd UNIX achieves this performance enhancement by mapping pairs of page-table entries. The SCAP microcode takes advantage of this operating system behaviour by storing hash table entries only for the first 512-byte page of a 1,024-byte pair. Since UNIX always contiguously maps with identical page protection such pairs of pages, the SCAP microcode can generate the translation-buffer entry for the second page of the pair by not using bit 9 of the virtual address in the hashing algorithm. After locating the hash entry, the microcode sets the low bit of the PTE to match bit 9 of the virtual address.

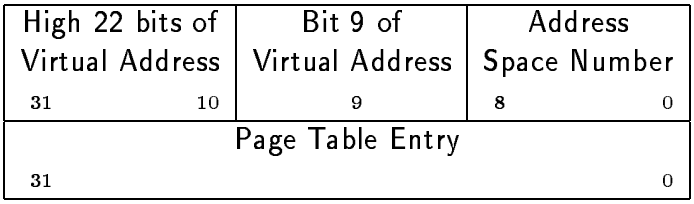

Figure 16.4: Hashed-Page-Table Entry Format

This change cuts the size of the hashed page table in half, freeing considerable memory for other use.[5] The resulting form of a hash table entry is shown in Figure 16.4. This tailoring of the microcode to the operating system's requirements is an example of how the RISC technique of vectoring translation buffer misses to operating-system-supplied code can significantly improve the performance of a system. See Section 15.3 for a more complete discussion of software versus hardware implementations of translation-buffer-miss handlers.

16.6.2 Hashing Algorithm

The hashing algorithm itself must map 22 bits of virtual address into an index into the hashed page tables. The hash must produce a uniform distribution to minimize the number of collisions in the hash table itself. Since most domains will use the same virtual addresses for the base of the code segment (page zero of P0 space) and for the base of the stack (the high page of P1 space), the hash function must include the address space number (ASN) to ensure that page zero of each domain does not hash to the same location.

The hash must also be easy to compute, given the operations of the VAX-11/730 microengine. Thus, an operation which reverses a set of bits, such as suggested in [100] is not a good choice, because the VAX-11/730 has no such

[5]Note that no matter how cheap memory parts become, the amount of memory dedicated to page tables must be a fixed percentage of the physical memory present. This percentage can be reduced only by using larger page sizes. The example here uses microcode to simulate a larger page size, despite the translation buffer hardware's use of a smaller page size.

operation. The VAX-11/730 microinstruction set is summarized in Appendix E and is described in detail in [217] and in [218].

Given four megabytes of primary memory, there are 8,192 pages of 512 bytes each. Assuming that pairs of physical pages are mapped as described above, and assuming that the hash table should have twice as many entries as pages of physical memory, then the hash table should have 8,192 entries of eight bytes each, and the hashing function should map 22 bits into 13 bits.

One possible hash function, shown in Figure 16.5, is to exclusive-or bits 19–31 of the virtual address with bits 10–22 of the virtual address and exclusive-or that result with the 9-bit ASN, left shifted by 4 bits. The ASN is left shifted, because the initial versions of SCAP will have bits 30 and 31 of the virtual address almost always 0. The bits will be 0, because S0 addresses will initially come out of the separate system space page table, and S1 addresses are not yet used. Left shifting the ASN assures some variability in the two high order bits.[6]

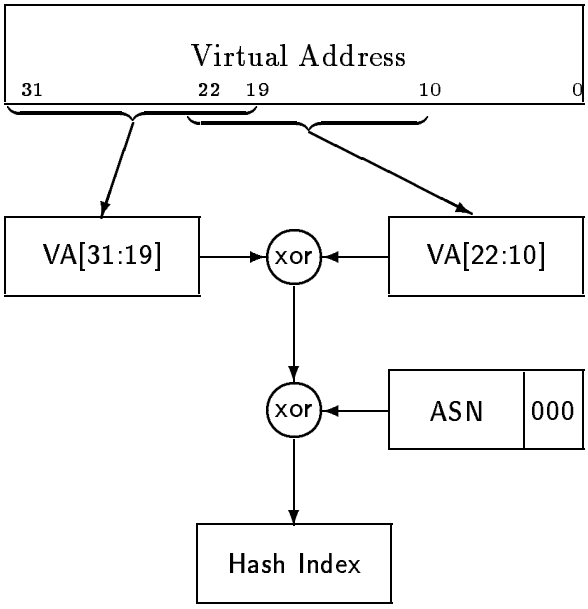

Figure 16.5: Hashing Algorithm

Figure 16.6 shows a VAX-11/730 microcode sequence to implement the hashing function of Figure 16.5. The microcode assumes that the virtual address and the address space number are stored in local store location LS[VA] and LS[ASN], respectively. The sequence uses LS[TEMP] and working register 0 as scratch

[6]Chang and Mergen [35] describe the hashing algorithm of the IBM RT PC as being the exclusive-or of the segment number and the page-within-segment number. This simple a hash function could have difficulties with uneven use of the hash table. If applications more frequently use the lower-numbered pages of each segment, the result would be biased toward the beginning of the hash table.

locations. It returns the value of the hash, multiplied by 8, in working register 1. The hash is multiplied by 8, so that it can be used to index 8-byte hash table entries. Note that the 9-bit right rotate operations require a delay before their results are available. Therefore, instructions are inserted between the MEM REQ and MOV MEM.DATA instructions to avoid stalling the processor. Finally, note that the masking operations are limited by the selection of constants stored in local store at microcode-initialization time. The microengine has no immediate-mode operands, so constants must be either computed and stored at initialization time (using up valuable local store locations) or computed at run time. The microcode in the figure has been simplified to illustrate how the hash could be computed. The actual implementation has to be inserted in line with other microcode. It uses different working registers and local store locations.

```
MEM REQ [ROTATE.BYTE.RIGHT] ADRS[VA] DT[LONG]
                          ; initiate the 9-bit right rotate
MOV LS[ASN] TO WR[0]      ; move ASN to WR[0]
MOV MEM.DATA TO LS[TEMP]  ; put shifted result in LS[TEMP]
MEM REQ [ROTATE.BYTE.RIGHT] ADRS[TEMP] DT[LONG]
                          ; shift VA right another 9 bits
MOV LS[TEMP] TO WR[1]     ; get VA[22:10] into WR[1][13:1]
MOV MEM.DATA TO LS[TEMP]  ; double shifted result to LS[TEMP]
XOR LS[TEMP] TO WR[1]     ; xor VA[31:19] with VA[22:10]
SHL2 WR[0]                ; shift ASN left two bits
ASHL WR[0]                ; shift ASN left another bit
XOR WR[0] TO WR[1]        ; xor the ASN into the hash
SHL2 WR[1]                ; shift result left two bits
AND LS[#FFFF] TO WR[1]    ; clear high 16 bits
BIC LS[#8] TO WR[1]       ; clear low 3 bits
```

Figure 16.6: Microcode to Compute Hash Function

16.6.3 Address-Space-Number Management

There may be many more domains than address space numbers (ASNs), because ASNs are limited to the range 0 to 511. This means that the operating system must multiplex the use of ASNs. This section describes the basic algorithms for that multiplexing.

ASNs are assigned sequentially to newly created domains. One ASN (with value 0) is reserved and never assigned to any domain. For each ASN, the kernel keeps two state bits indicating whether the ASN has been assigned and whether the ASN is *stale*. A stale ASN is one that has been assigned to a domain that is

no longer in memory. Stale ASNs may not be re-used, because their entries may still be present in the hashed page tables (in the VAX-11/730 implementation) or in the translation buffer (in an ideal SCAP processor that implemented ASNs in the translation buffer). When a domain is killed or is swapped out, its ASN is marked as stale. When a domain is swapped in, it must be assigned an ASN.

Eventually, the security kernel will run out of ASNs. At that time, the kernel flushes the translation buffer and the hashed page tables, marks all the ASNs as free, and starts over. Section 15.2.3 discusses the performance implications of running out of ASNs. Note that the complete flush of the hashed page table can be optimized by preserving those entries that belong to currently active domains. Currently active domains have their ASNs retained, and the page-table-flush algorithm scans all entries, examining their ASNs. If the ASN is marked in use, then the entry is left unmodified. However, if the ASN is marked free, then the ASN is replaced by the reserved ASN with value 0.

16.6.4 Invalidating PTEs in the Hash Table

Because the hashing algorithm uses open addressing with linear probing, invalidating entries in the hash table is not straightforward. The algorithm is designed to make lookups as fast as possible, because the table will be searched much more frequently than PTEs will be invalidated. The invalidation difficulty arises, because when collisions occur, an entry in the table may have been moved forward into the slot belonging to some other virtual address. The invalidation algorithm must be careful not to invalidate other entries that may have collided with the entry to be invalidated. The SCAP experimental implementation, described in Section 16.6.5 uses the following invalidation algorithm, based on deletion with linear probing [128, pp. 526-527].

- Compute an initial hash-table index for the virtual address. Save that index in a variable, INITIAL.
- Start at the INITIAL index and search for a match on the virtual address and ASN.
- When a match is found, invalidate the PTE, and store the index of the match in a variable, T.
- Continue searching forward, examining each hash entry until an entry is found containing the ASN reserved to terminate searches. For each examined entry, compute the hash of the virtual address and ASN that is stored to see if it resulted from a collision at the INITIAL location.
- If the entry was from a collision at the INITIAL location, then copy this entry up to location T, and set T to the old index of the copied entry. Invalidate the old copy, and continue searching forward.

- When an entry with the reserved ASN for termination is found, the procedure is finished. (Of course, the search forward must be prepared to wrap around at the end of the hash table and to terminate if the table is completely full.)

For the vast majority of cases, the algorithm terminates very quickly, because collisions are rare, and the search usually finds an exact match followed immediately by a terminate entry. The code for moving entries up is executed very rarely. However, the algorithm should handle even the rare cases in which multiple chains of collisions that hash to different locations actually overlap each other in the hash table.

16.6.5 Hashing Results

Measuring the performance benefits of hashed page tables is very difficult without an operating system that uses them. Therefore, modifications were made to Digital's experimental security kernel [145] to use hashed page tables, rather than the conventional VAX page tables.

The experimental security kernel, however, uses the conventional interpretation of VAX system space, that it is identical and shared in all address spaces. As a result, if S0 space were hashed as the process spaces were hashed, the page table entries (PTEs) for S0 space would appear in the hash table once for each address space, would waste a great deal of space, and would cause the hashing algorithm to thrash. The proper solution would be to make the operating system run in its own address space with its own ASN, but that level of modification was much too large to make to the experimental security kernel. Therefore, a compromise was adapted, and, as mentioned above in Section 16.6, only P0 space and P1 space were hashed, leaving S0 space to use the conventional page table.

A large benchmark of the experimental kernel to measure the performance differences of hashing revealed, somewhat surprisingly, that the performance differences were minimal. Hashing P0-space and P1-space page tables on a VAX-11/730 ran at about the same speed as the conventional page tables. The hashed tables did occupy somewhat less primary memory, but due to the form of the modifications, the experimental security kernel was not able to take advantage of the memory that was freed.[7]

The experiment was very limited and did not measure many of the effects that one might see in a hashed page table environment. Particularly, the effects of the operating system behaviour were not measured, and those could completely dominate. However, as a first approximation, it appears that the choice between conventional page tables and hashed page tables can be made entirely

[7]Due to the proprietary nature of Digital's experimental kernel, details of the benchmark results cannot be included, beyond noting that the differences were not significant.

on the basis of memory usage and convenience to the operating system designer. If the processor supports translation-buffer-miss handling in software, then the operating-system designer can make this choice and even switch between conventional page tables and hashed page tables in different releases of the system.

Chapter 17

Cross-Domain Call Optimization

17.1 Performance of Cross-Domain Calls

The performance of cross-domain calls is one of the most significant factors in determining the overall performance of a capability-based system. Gehringer and Colwell [77] suggest improvements to the Intel 432 cross-domain call that they estimate would save up to 54% of the call time. They suggest the addition of extra hardware-register sets to improve cross-domain-call performance. However, RISC hardware and compiler designers have developed improved register-assignment algorithms [34, 37] that can reduce the number of registers that must be saved and restored on procedure calls. Gehringer and Colwell assert that such optimizations cannot be used for separately compiled modules, but a generalization of recent work by Wall [224] should provide significant improvement in register optimization in cross-domain calls.

Wall achieves significant performance improvements (10% to 25%) by deferring register assignments until link time, although still using the Chaitin *register allocation by graph colouring* algorithms at compile time. Wall's approach saves and restores only those registers that the caller requires after the call returns and that the called procedure actually uses as intermediate values. Further, the algorithms tend to minimize the overlap of registers used by the calling and called procedures.

Wall's techniques assume that the calling and called procedures are in the same protection domain and therefore trust each other. However, with the information that Wall's compiler saves about register usage, optimizing register usage across cross-domain calls is also possible.

Optimization across cross-domain calls must be done very carefully. In particular, we must avoid the temptation to just leave values in registers, when the calling sequence of another domain claims to not disturb those values. For example, the CAP-I cross-domain call and return instructions [92] did not properly clear registers when returning from a cross-domain call. As a result, Johnson [109] was

able to penetrate Herbert's implementation [96] of the game of MOO[1] by reading values that were left in general registers after the MOO subsystem returned. In another example, Sites [197] discusses how the cost of saving and restoring 640 registers on the CRAY-1 led the designers of the CRAY-1 operating system to implement partial task switches, leaving values in some of these registers. Sites suggests that such partial task switches may lead to potential security breaches.

17.2 Multiple Register Sets

Multiple register sets could be used to speed up switching between protection domains. The earliest such use was on the Honeywell 800 processor [149] which switched automatically between 8 programs by maintaining 8 complete register sets. More recently, the S–1 computer from Lawrence Livermore Laboratory [154] has been designed to support 16 register sets to optimize switching between address spaces. Such multiple register sets allow the operating system to switch between different processes or protection domains very quickly without saving or restoring any registers. While the total number of domains to be scheduled might be larger than the number of register sets, the operating system could use a least-recently-used algorithm to minimize the amount of register saving and restoring.

If the multiple register sets were completely isolated, then no parameters could be passed in registers. The Berkeley RISC machines [172] introduced the notion of overlapping register sets (also called register windows) to optimize passing parameters between procedures of the same protection domain. Not only do overlapping register sets not help in passing parameters between different protection domains, but they also increase the number of registers to be saved and restored when moving between protection domains. Wall's approach of global register allocation at link time achieves much of the benefits of the overlapping register sets, entirely in software.

The chip area reserved for the overlapping register sets can instead be used for multiple register sets for each protection domain. Wall's techniques can be used to optimize register assignments and parameter passing on normal procedure calls, and the techniques proposed in this chapter can be used to optimize cross-domain calls. The Am29000 microprocessor [33, 110] provides just such an option. Its register file may be used either as multiple register sets for each address space, or as an overlapping-register-set scheme within a single address space, but not both at once. Furthermore, the Am29000 provides some global

[1]The game of MOO [5, 84] has become a classic example of a protected subsystem. The user should be able to write a program to solve the game, but the calling program should not be able to obtain the correct solution from the game's memory nor should the calling program be able to tamper with the game's history of high scorers.

registers that could be used for parameter passing on cross-domain calls, if they were managed as proposed here.

17.3 Argument Passing

There are three types of arguments that can be passed in a cross-domain call. Optimizing cross-domain calls requires understanding the performance implications of all three.

- capabilities for memory segments,
- capabilities for abstract data types, and
- small values that fit in processor registers

Most previous capability architectures, including particularly the Intel 432, passed arguments only by capability. Capabilities for abstract data types[2] are relatively cheap to pass, because only the capability itself must be copied from one domain to another. The called domain will either pass the abstractly-typed object to yet another domain, or it will explicitly unseal the abstract type, to gain access to its contents. In contrast, capabilities for memory segments are more expensive to pass, because both the calling and called domains must be able to address the contents of the segment. In the case of a paged machine, the cross-domain call could entail significant overhead in mapping the pages of the memory segment into the new address space. Thus, if an argument must be passed through several domains that do not all require access to the contents of the argument, then using abstract types can significantly improve the performance of the cross-domain call. (Section 17.3 shows that such use of abstract types is in fact common.)

A much greater performance gain can be achieved by passing small values in the processor registers. Indeed, the CAP-I ENTER instruction [92, Section 3.3] leaves the registers unmodified when crossing domain boundaries, and the CAP-I operating system passes many arguments in registers. As discussed in Section 17.1, attacks were possible, because the CAP-I did not properly clear the registers. CAP-I provided for up to 256 capability arguments on every ENTER call (one full capability segment). As shown in Section 17.3, most actual ENTERs passed a very small number of capabilities, making further savings possible.

The sources for Digital Equipment Corporation's research prototype security kernel for VAX processors [145] gave a rough estimate of the cost of cross-domain calls as a function of the number and types of arguments. This kernel is structured as a strictly layered architecture, such as originally proposed by

[2]Capabilities for abstract data types are implemented using sealing, as described in Section 4.4.

Janson [108]. While all layers of the prototype kernel run in the same protection domain, and layering is used only as a software-structuring tool, the relative frequency of cross-layer calls can be used to estimate the frequency of cross-domain calls, were the prototype kernel to be re-implemented on a capability machine. In one large benchmark, the prototype kernel executed 150,363,618 cross-layer calls. Although the kernel consisted of 10 layers and 168 distinct targets for cross-layer calls, 80.5% of all calls were to only three target procedures. Fourteen target procedures covered over 99% of all cross-layer calls. Of the three most frequently called procedures, one took no input arguments at all, and the other two took parameters that could be passed in registers plus one parameter each that was a data segment of less than 512 bytes in length. Of the next eleven most popular procedures, two took register parameters only, five took registers and capabilities for abstract types only, and only three took data structures that would require mapping. Again, all of these data structures were less than 512 bytes long.

A second estimate of the cost of cross-domain calls as a function of the number and types of arguments comes from Cook's evaluation [45] of the CAP-I operating system. Cook ran benchmarks that showed that calls on the spooled stream protected procedure (SSPP)[3] completely dominated all other cross-domain calls. The SSPP routines, described by Slinn in [198, page 106], take one or two integers and (sometimes) a capability as parameters. Thus, the most frequent cross-domain calls in the CAP-I operating system similarly pass most arguments in registers, rather than as capabilities to be mapped into an address space.

As further confirmation, Herbert's design of the CAP-III architecture provided the ability to pass up to five capabilities and four words of data [94, p. 24] in each message-passing cross-domain call.[4] Pardoe's current implementation [171] of an operating system for CAP-III typically passes only one capability argument along with some small data values.

Thus, trading-off performance of the relatively infrequent passing of capabilities to memory segments in favour of the frequent passing of arguments in registers should significantly improve the overall performance of cross-domain calls.

The remainder of this chapter will focus on argument passing to see what performance gains are possible without compromising security.

17.4 Categories of Trust

This section classifies the various types of procedure calls as a function of the type and level of trust between the calling and the called domains, in order to se-

[3]The spooled stream protected procedure (SSPP) provides a file-I/O interface to segments of virtual memory, much like the Multics file DIM [68].

[4]CAP-III actually implemented cross-domain calls as message-passing primitives and mapped each domain into a separate process. That distinction is not relevant here.

curely optimize the register usage in cross-domain calls. The trust relationships between two protection domains can be divided into two broad categories—trust for security and trust for integrity. These categories are analogous to the two principal models of non-discretionary access control—the Bell and LaPadula security model [14] and the Biba integrity model [18]. To trust a domain for security means that you trust the domain not to release information to unauthorized recipients. To trust a domain for integrity means that you trust the domain to not improperly modify or sabotage data structures that you may make available to the domain.

The calling domain and the called domain may trust each other for any different combination of security and integrity. For example, in a normal subroutine call, the calling and the called domains trust each other for both security and integrity, so no special protection is required between them. When a user program issues a call to the operating system, as with a VAX change-mode-to-kernel instruction, the caller trusts the operating system for both security and integrity. However, the operating system does not trust the user's program for either security or integrity and must take steps to protect itself. In a fault-tolerant system, a calling routine may not be worried about unauthorized release of information, but may be concerned that a called module could malfunction. That is an example of trusting for security, but not for integrity. The most severe cases are Schroeder's mutually-suspicious subsystems [189] in which neither the calling nor the called domain trusts the other for either security or integrity.

The discussions here about a domain trusting another domain for security or integrity only refer to trust in a discretionary sense. The register optimizations proposed in the next section are not sufficient to protect against Trojan horses in a non-discretionary sense. As discussed in Chapter 7, a pair of calling and called domains would have to execute at the same non-discretionary access class, because just the act of calling and returning constitute storage channels, regardless of what else the domains may do. The optimizations discussed here, therefore, deal only with security and integrity within a single, non-discretionary access class.

17.5 Register Optimization Based on Trust

Traditional subroutine calling sequences, such as the VAX procedure calling standard [106] typically require the called routine to save and restore those registers that it modifies. While this strategy is safe, it is far from optimal, since the called routine may save and restore many values in registers that the calling routine may never reference again. My optimization strategy divides the register usage into several categories. Some registers are used to pass arguments to the called domain. Those argument registers may be used for input only, for both input and output, or may be used to only return output values. Some registers

contain values that the caller wishes preserved across the subroutine call. The called domain may actually use some of these registers and not others. Finally, some registers contain values that the caller does not need preserved. In the optimal case with full mutual trust, the only registers that must be saved and restored are those that both the calling domain requires be preserved and the called domain actually modifies.

Table 17.1 summarizes these classes of registers and shows which registers must be either loaded with input values, cleared, saved, saved and cleared, or left untouched as part of the calling sequence. (An entry consisting of just a dash means the register is untouched during the call.) The minimum amount of register saving occurs when the calling domain trusts the called domain for both security and integrity. If the calling domain does not trust the called domain for integrity, then more registers must be saved. If the calling domain does not trust the called domain for security, then not only must registers be saved, but all registers that are not used to pass input parameters must also be cleared to prevent information leakage.

Returns from a cross-domain call must be analyzed as carefully as the calls. Table 17.2 shows the treatment of registers for the various cases of trust. The register categories are differ from those in Table 17.1, because the only divisions are between registers that the caller saved and registers the caller did not save. For argument-passing registers, the scheme assumes that all output registers are actually assigned values by the called domain.

17.6 Capability-Argument Optimization

Because the most frequently executed cross-domain calls pass either zero or one capabilities as arguments, SCAP does not use the CAP-I approach of passing an entire capability segment, but instead allows passing only a single capability. For most calls, that capability either is null or points to the single argument that must be passed as a capability. For the much less frequent calls that require more than one capability argument, then that one capability points to a capability segment that contains capabilities for the other arguments.

The capability argument is not automatically mapped into the address space of the called domain. Instead, the called domain must explicitly request that the capability be mapped, using a MAPARG instruction. This restriction, compared with the CAP strategy of mapping the calling domain's argument-passing segment (N-segment) into the called domain's argument-receiving segment (A-segment), exists to maximize the performance of cross-domain calls that pass only register arguments. It also improves performance in infrequent cases where a capability is passed to a domain, solely to be passed on to yet another domain.

Register Usage	Does Caller Trust Callee for Security(S) and Integrity(I)?			
	S and I	S only	I only	Neither
Argument Registers				
Input Arguments Used Later by Caller and Modified by Callee	loaded & saved	loaded & saved	loaded & saved	loaded & saved
Input Arguments Used Later by Caller and not Modified by Callee	loaded	loaded & saved	loaded	loaded & saved
Input Arguments not Used Later by Caller	loaded	loaded	loaded	loaded
Output Arguments	—	—	cleared	cleared
In/Out Arguments	loaded	loaded	loaded	loaded
Non-Argument Registers				
Caller Uses Later and Modified by Callee	saved	saved	saved & cleared	saved & cleared
Caller Uses Later and not Modified by Callee	—	saved	saved & cleared	saved & cleared
Other Registers	—	—	cleared	cleared

Saved registers will be restored upon Cross-Domain Return.
Registers with entries marked with a dash are untouched.

Table 17.1: Register Usage on Cross-Domain Call

Register Usage	Does Callee Trust Caller for Security(S) and Integrity(I)?			
	S and I	S only	I only	Neither
Output Arguments	loaded	loaded	loaded	loaded
In/Out Arguments	loaded	loaded	loaded	loaded
Saved by Caller	restored	restored	restored	restored
Not Saved by Caller	—	—	cleared	cleared

Registers with entries marked with a dash are untouched.

Table 17.2: Register Usage on Cross-Domain Return

The actual mechanics of passing the one capability argument must be carefully designed for maximum performance. Colwell [44, Section 3.3.5] found in the Intel 432 that every cross-domain call had to clear the memory of various access descriptors. This memory clearing added significantly to the cost of the 432's cross-domain call, and Colwell suggested improving performance by adding a special memory-clearing primitive [44, Section 5.1.2]. Such a memory-clearing primitive would still require a certain number of memory cycles to accomplish the clearing. Not having to clear the memory at all would be a much better solution.

17.6.1 Avoiding Clearing

Avoiding the need to clear the C-stack is very tricky, because the memory allocated for the C-stack will be used by many different domains as a process executes cross-domain call and return instructions. It must be impossible for a domain to improperly re-use a capability that might have been passed to some other domain in an earlier call, even though the C-stack frame for the current domain occupies the same memory location as that previous call. Always clearing the C-stack is sufficient to meet this security goal, but at extreme performance cost.

Imposing certain conventions on the use of the C-stack can eliminate the need for most clearing. First, all C-stack frames are the same size. No matter how many registers are pushed in a call, sufficient space is left for all registers. Second, with the exception of the single capability argument, no user-manipulable data is stored in the C-stack. Only the cross-domain call and return instructions can read and write the C-stack frame, and they only reference data that the user can properly see.[5] Given these assumptions, the SCAP cross-domain call need

[5]For example, the cross-domain return instruction only restores those registers that were actually pushed. See Section 17.7.2 and 17.7.3 for more details.

not clear most of the C-stack frame, either on call or return, and thereby saves a significant number of memory cycles.

The capability-argument field in the C-stack remains problematical. The need for clearing of the capability-argument field depends on whether there are subsequent cross-domain calls from the current domain, and, if there are subsequent calls, whether those calls pass capability arguments. Section 17.8 shows the solution to when to clear the capability argument field.

17.6.2 Avoiding Probing

A VAX-based capability architecture could expend significant numbers of cycles probing the C-stack prior to use, to allow restarting instructions after page faults. SCAP avoids probing the C-stack by requiring that all pages of the C-stack for the current process be locked into primary memory when that process is allowed to run. If any page of the C-stack is found to be inaccessible, then the processor takes a C-stack-not-valid abort, analogous to the VAX-architecture's kernel-stack-not-valid abort [138, p. 240], and the operating system terminates the offending process. This restriction is feasible, because the C-stack is totally under the control of the privileged operating system, and no user programs can affect it. The size of the C-stack can be determined at process-creation time, and, barring infinite cross-domain-call-recursion loops, the C-stack should never be very large. For example, the CAP-I operating system normally limits the size of the C-stack for a user process to 256 words of 32 bits each.

17.7 Implementing the Optimizations

Implementing the register optimizations for cross-domain calls is not straightforward. Wall's approach of assigning registers at link time is sufficient for calls in which the calling and called domains fully trust each other, but a cross-domain call mechanism must assume that the output of a normal linker could be maliciously modified. The code that actually performs the register saving, clearing, and restoring must be protected from unauthorized tampering. That code is generated by a *trusted cross-domain linker*. There are two alternate implementation approaches: one using a microcoded cross-domain-call instruction, such as found in CAP [231], and one using a RISC approach with specially compiled code sequences for each cross-domain call and return.

17.7.1 Trusted Linker

The trusted cross-domain linker finds the cross-domain call instructions of the calling domain. The linker then finds the called domains, and, based on the information provided by the compilers of the calling domain and all the called

domains, constructs three register masks for each cross-domain call instruction in the calling domain. These three masks are analogous to the procedure-entry masks of the VAX CALLx instructions [138, p. 89]. The VAX procedure-entry mask specifies which registers must be saved during the call and restored during the return. By contrast, the three masks for cross-domain calls specify which registers must be saved during the call and restored during the return, which registers must be cleared during the call, and which registers must be cleared during the return. The contents of the masks must be protected from tampering by both the calling and the called domains. Therefore, the cross-domain call instruction simply specify a numeric offset within a protected-linkage table that contains the masks and the actual location of the target domain.[6]

The trusted linker could be run statically, much like conventional linkers, or it could run dynamically, much like the Multics dynamic linker [168]. Unlike the Multics linker that Janson [107] removed from the security kernel, the trusted linker will be very simple, since it need only generate the register masks. (The trusted linker will do a little more in the RISC case described in Section 17.7.3.)

17.7.2 Microcoded Cross-Domain Call

This section describes one possible microcoded implementation of cross-domain call and return, as an extension to the VAX architecture. See Appendix D for a description of the various registers that make up the processor state.

To make a cross-domain call, the calling domain loads the input arguments into registers, before it issues the cross-domain call instruction. The instruction takes two operands, the index into a cross-domain linkage table, as generated by the trusted linker, and a capability to be passed to the new domain. The cross-domain linkage table is stored at a location specified in the C-stack frame, thus avoiding the need for a privileged register. To avoid the need for a length check, the size of the table is fixed at 256 entries so that the index can be specified as a single byte. This restricts the number of cross-domain call targets that any single domain can contain, but each domain has its own table.

Each entry in the cross-domain linkage table (shown in Figure 17.1) contains the three register masks and a pointer to the *domain control block (DCB)*[7] for the called domain. The domain control block is analogous to the *process resource list (PRL) entry* in CAP-I. The domain control block contains a set of memory-management registers used to map the called domain into memory and a pointer to the starting location in the domain. Just as in CAP, the microcode for the

[6]Even if a malicious caller deliberately specified an incorrect offset, no security violation could occur. All that would happen is that a different, but authorized, cross-domain call would be executed.

[7]The name, *domain control block*, is used, because of the analogy to the VAX *process control block (PCB)*.

Save/Restore Register Mask
Call Clear Mask
Return Clear Mask
Pointer to Domain Control Block (DCB)

Figure 17.1: Format of Cross-Domain Linkage Table Entry

cross-domain call pushes the necessary linkage information onto the C-stack. The processor has a new stack-pointer register, called the CSP, to point to the top of the C-stack. Figure 17.2 shows the format of a frame on the C-stack, and Figure 17.3 shows the domain control block (DCB). The order of the entries in the C-stack frame and the DCB is determined by the ease of writing the microcode to push and pop the values.

Pointer to Callee's Cross-Domain Linkage Table	
PME	P1LR
P1BR	
ASTLVL	P0LR
P0BR	
PSL	
PC	
FP through R0	
USP	
SSP	
ESP	
KSP	
Save/Restore Register Mask	
Return Clear Mask	

Figure 17.2: Format of C-stack Frame

The microcode for the cross-domain call instruction performs the following tasks:

1. Use the supplied index into the cross-domain linkage table to locate the DCB.

2. Push the call frame onto the C-stack, storing only the appropriate registers, specified in the save/restore mask.

3. Push the save-register mask and the return-clear mask onto the C-stack.

4. Clear the registers specified in the call-clear mask.

PSL	
PC	
PME	P1LR
P1BR	
ASTLVL	P0LR
P0BR	
USP	
SSP	
ESP	
KSP	
Pointer to Callee's Cross-Domain Linkage Table	

Figure 17.3: Format of Domain Control Block (DCB)

5. Load the new values into the memory-management registers from the DCB of the called domain.

6. Clear the translation buffer.

7. Transfer control to the start location.

The new domain optionally maps the capability argument into its address space as a separate operation, and eventually issues a cross-domain return instruction, after first loading the output argument registers with appropriate values. The return instruction performs the following steps:

1. Pop the memory-management registers off the top of the C-stack and load them.

2. Clear the translation buffer.

3. Pop the clear-register mask off the C-stack and clear the appropriate registers.

4. Pop the save-register mask off the C-stack and use it to reload the specified general registers.

5. Perform an REI (return from exception or interrupt) using the PC and PSL on the top of the C-stack, thus returning to the calling domain.

The performance benefits of the microcoded cross-domain-call instruction will depend heavily on the relative performance match of the processor's cycle time, the speed of primary memory references (from the cache, if it exists), and the amount of computation that can be performed in each microinstruction. More specifically, using masks to selectively save and clear registers is a performance

benefit only if the microengine can check the bits in the masks faster than it can save and clear the registers.

```
; Assumption is that the stack pointer is in LS[T3] upon entry
;
       MOV LS[#E] TO Q         ;Set up loop counter
       MOV Q TO LS[OS]         ; in the operand specifier
       MOV LS[#4] TO WR[2]     ;Get constant 4 for stack pushes
       JSR LOOP                ;Start the loop
LOOP:  MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG]
       SUB WR[2] FROM LS[T3]   ;Push the stack pointer
       DEC LS[OS],             ;Decrement the loop counter
         DT(LONG)&SET.ALU.CC   ; and set condition codes
       WRITE.MEM LS[GPR.OS],   ;Write the register value
         SKIP.IF[MEM.REF.OK]   ; and skip if no error
       JSR [WRITE.GPR.OS]      ;Jump to error subroutine
       CLR LS[GPR.OS],         ;Clear the general register
         LOOP.IF(NEQ)          ; and loop until done
```

Figure 17.4: Microcode to Save and Clear Registers Unconditionally

Figure 17.4 shows a section of VAX-11/730 microcode which saves and clears all the general purpose registers unconditionally.[8] Figure 17.5 shows a section of VAX-11/730 microcode to save and clear only those registers indicated in masks. The microcode to unconditionally save and clear all the registers is actually the faster of the two (five microinstructions per register versus eight, assuming all memory references work), because the relative speed of the VAX-11/730 memory [217] is such that checking the two masks takes longer than just performing the register pushes and clears. By contrast, it appears that the VAX-11/780 microengine [219] probably could make effective use of the masks, because it can perform more functions per memory cycle. More interestingly, it appears that the VAX 8800 processor [74], like the VAX-11/730, would be unlikely to benefit from the masks, because it can reference memory faster than it can test bit masks. The VAX 8800 is pipelined at the microcode level, and its memory cache has a special bypass path to minimize the effects of stalls. This effect is not surprising, since the principal reason [230] for the current success of RISC processors is the availability of extremely fast primary memories and cache memories.

[8] Appendix E contains a brief introduction to VAX-11/730 microcode.

```
; Assumption is that the stack pointer is in LS[T3]
; WR[0] contains the register save mask
; WR[1] contains the register clear mask
;
        MOV LS[#E] TO Q              ;Set up loop counter
        MOV Q TO LS[OS]              ; in the operand specifier
        MOV LS[#4] TO WR[2]          ;Get constant 4
        JSR LOOP                     ;Start the loop
LOOP:   SHL WR[0],                   ;Check the save mask
          DT(LONG)&SET.ALU.CC        ; and set condition codes
        SUB WR[2] FROM LS[T3] TO Q,  ;Decrement stack ptr to Q
          JMP.IF(N.CLR) TO CLEAR     ; and jump if register
                                     ; does not need saving
        MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;Start writing
        MOV Q TO LS[T3]              ;Move new stack ptr to T3
        WRITE.MEM LS[GPR.OS],        ;Write the register value
          SKIP.IF[MEM.REF.OK]        ; and skip if no error
        JSR [WRITE.GPR.OS]           ;Jump to error subroutine
CLEAR:  SHL WR[1],                   ;Check the clear mask
          DT(LONG)&SET.ALU.CC        ; and set condition codes
        DEC LS[OS],                  ;Decrement loop counter,
          DT(LONG)&SET.ALU.CC        ; set condition codes,
          SKIP.IF(N.CLR)             ; and skip on clear mask
        CLR LS[GPR.OS],              ;Clear the register
          LOOP.IF(NEQ)               ; and loop if not done
        NOP,                         ;Extra loop test, for when
          LOOP.IF(NEQ)               ; CLR is skipped
```

Figure 17.5: Microcode to Save and Clear Registers Using Masks

17.7.3 RISC Cross-Domain Call

In a RISC implementation, there is no microcode, and cross-domain call and return must be implemented as a series of simple instructions. Such simple instructions would each run much faster than complex instructions, and could be pipelined. Further, the expense of looping through register masks is likely to be excessive in a RISC implementation. Instead, trusted linker generates a series of load or store instructions to save or restore precisely the needed registers. The code sequences replace the cross-domain linkage table used in the microcoded version in Section 17.7.2. The cross-domain call instruction similarly takes an operand to identify which call is being made. The operand serves as an index into a transfer vector to locate the proper code sequence.

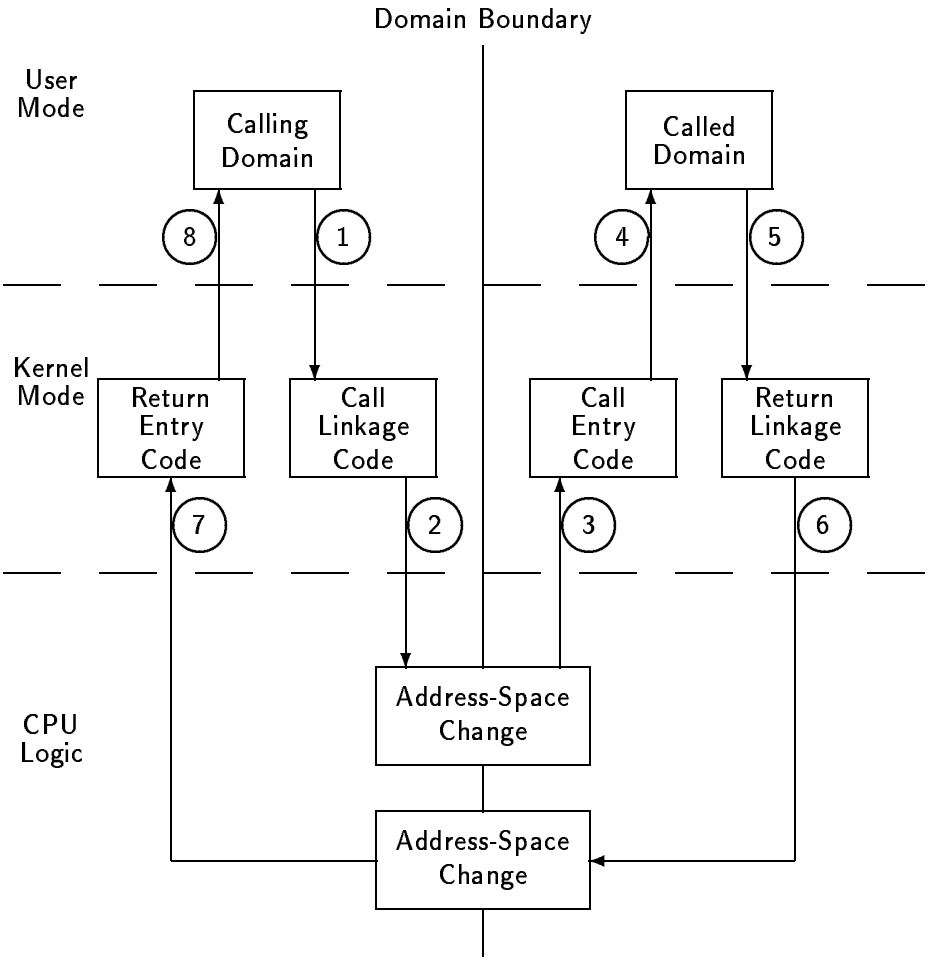

Figure 17.6: RISC Optimized Cross-Domain Call

The actual mechanism of a cross-domain call, shown in Figure 17.6, follows:

1. The calling domain loads argument values into the registers and executes a cross-domain call instruction.

2. The cross-domain call instruction traps into kernel mode and transfers control to the call-linkage code generated by the trusted linker. The call-linkage code pushes the current values of the program counter (PC) and processor status longword (PSL) onto the C-stack and then executes a series of stores and clears to push onto the C-stack those registers that must be saved and to clear those registers that must be cleared. Then, the call linkage code pushes the address of the return linkage code onto the C-stack, followed by the current values of the memory-management registers.

3. Next, the call entry code loads new values into the memory-management registers from the DCB of the called domain and then clears the translation buffer.

4. Finally, the call entry code transfers control to the start location.

Cross-domain return, also shown in Figure 17.6, uses the following algorithm:

5. The called domain loads values into the registers to be used as output arguments and executes a cross-domain return instruction.

6. The return instruction traps to kernel mode and transfers control to standard return-linkage code. The standard return linkage code pops the memory-management registers off the C-stack, loads them, and clears the translation buffer.

7. Next, the return-linkage code pops the address of the return-entry code, generated by the trusted linker, and transfers to that code to restore or clear the appropriate registers.

8. Finally, the return-entry code executes an REI (return from exception or interrupt) using the PC and PSL on the C-stack.

Leonard has suggested [137] one additional optimization that is possible with assistance from both the compiler and the trusted linker. Instead of clearing registers during both the cross-domain call and cross-domain return sequences, the compiler can provide specified values to be loaded into those registers, thereby saving some instructions in some cases. The values to be loaded are specified in tables, so that the trusted linker merely loads the specified constant rather than a zero into the register. The complexity of the trusted linker does not significantly change by this additional optimization.

17.8 Minimizing Argument Clearing

Section 17.6.1 outlined the problem of clearing the capability-argument field in the C-stack frame. The simplest solution would be to incorporate the passing of the capability as part of the cross-domain call instruction and clear the field on every return. Thus far, however, the cross-domain call instruction does not depend on capabilities at all. It only needs to reference the general purpose registers and the memory-mapping registers. It would be highly desirable not to build knowledge of the capability format into the instruction in either the microcoded case or the RISC case, because the capability format is relatively complex and subject to change as the operating system evolves. Further, if the call instruction always passed the argument, the return instruction would always have to clear the field, and many of those clears would be redundant.

Therefore, SCAP implements capability argument passing with a new technique, called *conditional clearing on return*. The cross-domain call instruction has no knowledge of capabilities at all. It only handles switching address spaces and optimizing the general-register saving, restoring, and clearing. A location is reserved in the C-stack frame to hold an optional capability argument. If a calling domain wishes to pass a capability argument, the domain must first execute a PASSARG instruction that copies a specified capability into the argument field in the next C-stack frame, that is, the C-stack frame belonging to the domain about to be called. The called domain executes a MAPARG instruction to extract the capability from the C-stack frame and loads it into a capability segment slot. PASSARG and MAPARG are both implemented as operating system calls, since many cross-domain calls do not use them at all. When a called domain returns, the capability argument field used in that call is *not* cleared. In that way, if the calling domain wishes to make a second call, using the same capability parameter, no additional memory cycles need be expended. If the calling domain wishes to make another call and pass a different capability argument, it calls PASSARG and overwrites the previous capability. If it wishes to make another call and pass no capability arguments, then the calling domain must explicitly call PASSARG with a null capability to overwrite the previous capability. In this way, no unnecessary clearing operations are done.

Eventually, the calling domain must return to its caller, and something must be done about any capability remaining in the argument field. The field could be unconditionally cleared, but that would waste memory cycles if either no calls had been made or if the last call made had passed no capabilities. Therefore, the clearing on return is made conditional. A spare bit in the C-stack frame of the calling procedure is reserved to indicated whether the capability argument field contains a valid capability. The bit should be taken from a spare bit in some other entry in the C-stack frame that would have to be referenced anyway. In SCAP, based on the VAX architecture, the clearing-on-return bit replaces one

of the bits in the processor status longword (PSL) that must otherwise be zero. The PSL must be read on every cross-domain return, so no additional memory cycles are required to check the bit.

The clearing-on-return bit is set to one every time PASSARG is called with a capability. If PASSARG is called with a null capability, then the bit is cleared. When the cross-domain return occurs, the bit is checked. If set, the cross-domain argument field is cleared, along with the bit itself. If the bit is clear, then the return proceeds immediately.

The algorithm just described uses the C-stack frames of the current domain and the next domain in a complex fashion. Figure 17.7 should clarify how the frames are used. In the figure, domain A has called domain B which is currently executing. B will call C, D, and E, which in turn will call other domains. When B executes a PASSARG instruction, the capability argument is written into the frame for C, D, and E, but the bit used for conditional clearing on return is be written into B's own frame. Note that the same storage locations are used for the calls to C, D, and E. Likewise, the domains called by C, D, or E will share the same storage locations. The conditional-clearing-on-return algorithm ensures that the capability-argument locations in those shared stack frames are cleared the precise number of times required, and no more.

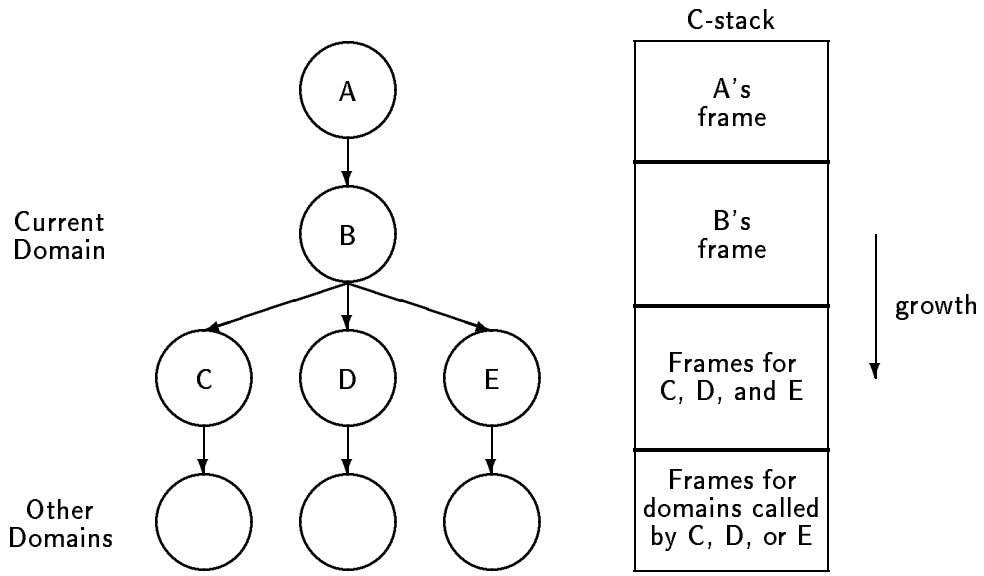

Figure 17.7: C-stack Frame Usage

No memory cycles are wasted by clearing the capability-argument field only to immediately store another capability in the field. This is achieved by recognizing that most cross-domain calls require only register arguments, and by taking advantage that testing a bit in a field that must be read and restored

is significantly less expensive than the extra memory cycles that unconditional clearing would require.

The conditional-clearing-on-return strategy just described is for an implementation of cross-domain call and return using microcode. For a RISC implementation, there is no need to check a bit in the C-stack frame. Instead, the trusted linker generates two alternate code sequences to clear or not to clear the capability-argument field. The PASSARG call patches the transfer address in the trusted linker's area to point to either one code sequence or the other, as required.

The capability-argument-passing strategy has increased the size of the C-stack frame from that shown in Figure 17.2. The revised C-stack frame format is shown in Figure 17.8. The CR bit to implement the conditional-clear-on-return operation is shown using one of the spare bits in the PSL. This minimizes memory usage in the frame, but requires that the PASSARG call perform extra work to set the bit without disturbing the rest of the PSL. Alternatively, the CR bit could appear in a longword by itself, and PASSARG would require fewer memory cycles. As memory is getting cheaper, the latter strategy is probably better.

Pointer to New Cross-Domain Linkage Table	
Received Capability Argument	
PME	P1LR
P1BR	
ASTLVL	P0LR
P0BR	
CR	PSL
PC	
FP through R0	
USP	
SSP	
ESP	
KSP	
Save Register Mask	
Return Clear Mask	

Figure 17.8: Format of Revised C-stack Frame

One may wonder why conditional clear on return is worth implementing, given that a much larger number of memory cycles are spent saving and restoring the VAX memory management registers (P0BR, P0LR, P1BR, and P1LR) and the VAX stack pointers (KSP, ESP, SSP, and USP). In fact, those costs overwhelm any benefit from conditional clear on return for a VAX-like processor. In a simpler memory management strategy, however, the benefits of conditional clearing

on return are significant. This chapter uses the VAX memory management as an example, so that the design concepts can be more closely related to existing technology. An ideal implementation would use a single memory-management register, rather than four, and would not have separate stack pointers for each protection ring.

17.9 Optimizing With Long Returns

There is a further optimization possible for cross-domain returns. This optimization, called *long return* by Stroustrup [204] depends on the observation that many procedures call other procedures as their last operation and, as a result, several return instructions may be executed sequentially. This type of behaviour is most frequently found in tail-recursive functions.[9]

If the compiler recognizes that the calling procedure will simply return to its caller after the return of the called procedure, then the compiler can generate a special kind of cross-domain call that does not save registers and a return location on the C-stack, but re-uses the frame from the previous domain. When the called procedure eventually executes its cross-domain return, the return skips over the intermediate procedure and returns directly to the caller of the calling procedure. With this optimization, the cost of both the final call and return are significantly reduced.

Opportunities for using long returns can be recognized at the time of compiling the calling procedures and need no information about the called routines. The actual change is to the code of the calling routine to omit unnecessary saves. The called routine need not know whether it is making a normal return or a long return, so separate compilation is not affected by the optimization. The use of long returns has particular payoff for cross-domain returns, because even with all the register-saving optimizations described above, the cross-domain call requires more memory references than a normal call. While long return may save a memory reference or two in a normal subroutine returns, the use of a long return in a cross-domain return can save dozens of memory references. The Honeywell DPS 8 processor implements a form of the long return optimization with the PCLIMB version of the CLIMB cross-domain call instructions. [64, page 7–88] Unfortunately, the only published discussion of performance of the DPS 8 processor [150] does not address the potential performance benefits of LTRAD. Indeed, as of the publication of that paper in 1984, the GCOS 8 operating system for the DPS 8 processor did not use the non-hierarchic-protection-domain features of the hardware.

[9] Although Stroustrup [204] coined the term long return, the idea had been previously implemented by Steele [202]. Steele [203, page 6] further suggests that the idea of long returns had been used much earlier to optimize tail recursion in LISP compilers on the PDP-10 and even the PDP-1!

The long return optimization has no impact on the conditional clearing on return optimization, described above in Section 17.8. This is because the long return optimization simply avoids pushing unnecessary C-stack frames in certain cases. The code to test the bit in the PSL or to transfer to the correct return code will continue to work correctly in such cases.

17.10 Benefits

While a capability-based security kernel for SCAP would probably be quite different from Digital's prototype kernel for VAX processors, the frequency and types of arguments of cross-domain calls will be similar. (See Appendix G for a design sketch of a capability-based security kernel.) Thus, if the costs of simple cross-domain calls with few arguments are minimized, the performance degradation of domain-structured protection should not be significantly worse than in a non-domain-structured kernel that implements cross-layer calls with the VAX CALLx instruction. Chapter 18 shows the measured performance of microcoded cross-domain calls on the VAX-11/730.

As seen above, the number of arguments that cannot be passed in registers is small. Further, the different protection domains that make up a security kernel will trust each other for security and integrity. Domain structuring is a tool to aid software development, minimize the propagation of errors, and to aid in the formal verification of the kernel as a whole. The domains are not mutually-suspicious subsystems, as viewed by Schroeder [189]. The register optimization thus can be taken from the first column of Tables 17.1 and 17.2, where the calling and the called domains trust each other. Much of the cost for register saving, clearing, and restoring can be minimized for the most frequently executed cross-domain calls and returns.

Chapter 18

Cross-Domain-Call Performance Experiments

Each cross-domain-call experiment ran stand-alone on the VAX-11/730. No operating system or user programs were present. The experiments were driven by a testing program written in VAX PL/I[1], with selected subroutines in VAX MACRO.

Each experiment had a calling domain and a called domain. The calling domain executed a loop consisting of 10,000 cross-domain calls to the called domain. The called domain simply executed a cross-domain-return instruction. This type of experiment measured the cost of the call and return instructions only. No measurements were made of the cost of translation-buffer refills, as these would be dependent on operating-system design and on specific application design.

The timings were done with the VAX interval timer that increments once each microsecond. Immediately prior to starting each experiment, the interval timer was set to zero and started. Immediately after the last cross-domain return, the interval timer was read and stopped. Each experiment consisted of 10,000 iterations in order to minimize any measurement skew due to the time required to read the interval timer. I measured the time required to start the interval timer at 0 and then immediately read its value to be 81 microseconds.[2] This value is high, because the VAX-11/730 timers are implemented in the 8085A console processor, and the communications path to the console processor is quite slow. Using 10,000 iterations of each experiment eliminates any errors from the 81 microseconds of timing overhead.

This style of timing depends on the use of the full VAX interval timer implementation. VAX subsetting rules [138, p. 287] permit implementation of an

[1]The experiments were written in VAX PL/I, because the author was familiar with its use in a stand-alone environment on a VAX. Many other higher level languages could equally well have been used.

[2]The VAX 8550 can read its timer in 4 microseconds.

interval timer that only gives 10-millisecond interrupts. Running these experiments with only the subset timer would have been possible, but more difficult, as a clock interrupt handler would have been required. The accuracy of the measurements would not have been affected, although increasing to 100,000 iterations might have been desirable.

Appendix I contains the actual code for certain of the experiments described in this section. To limit the volume of listings, not all the experiments are included in the appendix.

18.1 Experimental Results

		μseconds		
	Test Name	730 Macro	730 μcode	8550 Macro
1	JSB/RSB	7	N/A	1
2	CALLS/RET saving 0 regs	51	N/A	3
3	CALLS/RET saving 4 regs	60	N/A	3
4	CALLS/RET saving 10 regs	79	N/A	4
5	Cross-Domain Call/Return with SVPCTX and LDPCTX	803	N/A	46
6	Cross-Domain Call/Return with handcoded equivalent of SVPCTX	1255	N/A	83
7	Cross-Domain Call/Return saving and clearing all regs	1231	372	80
8	Cross-Domain Call/Return passing 3 args returning 1, saving 4, clearing others	1185	347	74
9	Cross-Domain Call/Return passing 3 args returning 1, saving & clearing none	1079	N/A	69
10	Cross-Domain Call/Return as in 9 plus skipping ESP, SSP, ASTLVL, PME	960	335	57
11	Cross-Domain Call/Return as in 10 plus simulating ASNs	355	112	31

Table 18.1: Cross-Domain-Call/Return Performance Results

Table 18.1 shows the timing results of various experiments. Where possible and meaningful, some of the experiments show implementations of the same design of cross-domain call and return in both VAX MACRO and in VAX-11/730 microcode. The MACRO numbers are shown for both the VAX-11/730 and for the VAX 8550. The entries in the table that are marked N/A had no microcode version. The purpose of doing both implementations is to show the benefits gained by microcoding frequently used complex instructions over software implementation in a machine like the VAX-11/730. All of the timing numbers were

measured for 10,000 iterations, but then normalized to 1 iteration. The time for the loop instruction has not been factored out of any of the numbers.

The first four tests measure the cost of the standard VAX subroutine calling instructions. The first test measures the Jump-to-Subroutine (JSB) and Return-from-Subroutine (RSB) instructions, these being the fastest way to call a subroutine in the VAX architecture. JSB simply pushes the PC onto the stack and transfers control to the subroutine. RSB pops the PC off the stack and returns to it. Section I.1 shows the machine code used to measure the performance of JSB and RSB.

By comparison, the full VAX calling sequence uses the Call-Procedure-with-Stack-Argument-List (CALLS) and the Return-from-Procedure (RET) instructions to create stack frames and save from 0 to 10 of the general purpose registers, based on register masks. As can be seen from the tests, the full VAX procedure call can easily be much more expensive than a simple call using JSB. Section 17.7.2 showed how unconditional saving of all the registers could be faster than selective saving of the registers. In fact, the comments in the microcode source files indicate that the VAX-11/730 CALLS instruction was optimized for space, rather than for speed. By contrast, the VAX 8550 microcode optimizes the performance of the CALLS and RET instructions, including special optimization for the case where all ten registers are saved.[3]

Test 5 is the first test with a true cross-domain call. The test shows a cross-domain call and return implemented with the VAX Save Process Context (SVPCTX) and Load Process Context (LDPCTX) instructions. SVPCTX and LDPCTX are intended for scheduling processes, rather than cross-domain calls, and are therefore less than ideal. In particular, they always save and then load all the general-purpose registers, so no register arguments are possible, and they do not conveniently follow a cross-domain call stack approach, but save in a location specified in the processor-control-block-base register (PCBB). Despite these drawbacks, it is possible to build a cross-domain call from them, and as the results of tests 6 through 8 show, a microcoded instruction that doesn't do quite the right thing is preferable to a series of simpler instructions that do exactly the right thing, but require many more instruction decodes. Section I.2 shows the actual code for test 5.

Test 6 shows a cross-domain call and return built out of simple VAX instructions that implement an algorithm identical to that in test 5. This shows that, both for the VAX-11/730 and the VAX 8550, microcoding large complex sequences of instructions is advantageous if they are executed frequently. Most of the performance gain of the microcode comes from having to decode only a single instruction, instead of a large number of instructions.

[3]The VAX calling standard never saves more than 10 registers out of the total 16 general purpose registers, because the remaining six registers are reserved for other purposes.

Test 7 is the first test that includes both a MACRO version and a microcode version. This test is a true cross-domain call in which all general registers are saved on the C-stack and then cleared prior to entering the called domain. The registers are then all restored on return. Once again, no arguments could be passed in registers, but the registers are handled efficiently. The microcode implementation takes less than half the time of the SVPCTX and LDPCTX implementation, indicating that optimizing the microcode for the precise task can give significant performance gains. The code for invoking the microcode versions of this test (7) and all subsequent microcode tests is shown in Section I.3. The actual microcode for test 7 is shown in Section I.4.

Test 8 is the first test to take advantage of register optimization based on trust, as defined in Section 17.5. This test passes three register arguments on call and returns one register value on return. Four registers are assumed to overlap between the calling and the called domains and are therefore saved and restored. It is assumed that the calling and called domains do not trust one another, so all non-argument registers are cleared. The microcoded version here is handcoded to perform exactly those operations, simulating the code that the trusted linker would generate in a RISC machine implementation. This test and subsequent tests do not measure the use of register masks, because Section 17.7.2 showed that the use of masks would always be slower than unconditional saving and restoring on the VAX-11/730.

Test 9 is identical to test 8, except the calling and called domains are assumed to trust one another. Therefore, register clearing can be avoided. Only the software version was implemented to show the small but significant performance gain possible here. The microcode performance should be about midway between the results obtained for tests 8 and 10.

Test 10 is identical to test 9, except that operations specific to the VAX architecture have been omitted. All tests, thus far, have saved and restored the executive (ESP) and supervisor (SSP) stack pointers, the ASTLVL register, and the PME register. A true RISC implementation of SCAP would not include these registers that are needed only to maintain VAX compatibility. The results show that another small but significant performance gain is possible by omitting these registers.[4]

Finally, test 11 simulates the behaviour of a translation buffer that supports ASNs. As can be seen, at least one third of the cost of the most highly optimized cross-domain call is spent in flushing the translation buffer. This test omits the flushing, but simulates the time required to load an address space number (ASN)

[4]PDP-11 compatibility mode was also going to be omitted in this test, but close examination revealed that the only savings would be in microcode space, rather than performance. Compatibility mode is entered by setting a bit in the PSL, and that bit can be tested at the same time as other bits are tested, so that the normal path through cross-domain call and return require no extra cycles for compatibility mode.

register. While this simulation of ASNs is clearly inaccurate, it does make clear that translation buffer flushing is the largest single contributor to the cost of cross-domain calls.

18.2 Comments on the Performance Results

Several interesting conclusions can be drawn from the performance numbers. First, the ratio of the most heavily optimized cross-domain call to the CALLS type of call is a little under 2:1. This says that cross-domain calls could replace CALLS calls in selected parts of the security kernel without totally destroying system performance. The ratio to the JSB type of call is not as good—roughly 16:1. If one compares that ratio to the ratios measured on CAP-I by Cook [45, Section 7.3.2][5], things do not look so dismal.

Cook computed a ratio of 114.6:1 comparing a CAP ENTER/RETURN sequence to a simple register load instruction. Cook's experiments subtracted out the cost of the looping instructions, while mine do not, and Cook measured register load instructions, rather than simple subroutine calls. The CAP-I has no exact counterpart to the JSB and RSB instructions, although SREB (subroutine entry) is similar to JSB. Cook reported times for both the BS (register load) and TCS (test and count) instructions. By counting the number of microinstructions used in the CAP-I interpreter to implement SREB, TCS, and BS, I estimated that the time for a JSB and an RSB would be roughly that of a TCS and a BS. Using this estimate, if Cook had used my experimental methodology, the ratio on CAP would have been 31:1, compared to my SCAP ratio of 16:1.[6]

This suggests that the SCAP optimization techniques gain a significant improvement over the CAP ENTER instruction. Since the CAP ENTER instruction is simpler and faster than cross-domain call instructions on other capability machines, this makes my SCAP optimizations all the more significant. However, Cook's experimental methodology is sufficiently different from mine that no stronger conclusions should be drawn without much more careful measurements on both processors.

Benchmarks, such as these, must be treated with caution, because their performance on the VAX can be changed by more than a factor of two by very subtle effects. First, the target of the loop instruction that performs the 10,000 iterations must be longword aligned to give the best performance. If the start

[5]It is interesting to note that the absolute speeds of the Cambridge CAP computer and the VAX-11/730 seem to be similar. Of course, CAP was built using early 1970s hardware technology and commissioned in 1975, while the VAX-11/730 was built in late 1970s technology and announced in 1982.

[6]The 31:1 ratio was computed by taking Cook's instruction times from Table 7.3 in his dissertation [45]. The ratio of an ENTER + RETURN + TCS instruction to a TCS + BS + TCS instruction was $240.0 + 2.8 : 2.8 + 2.14 + 2.8$ or 31:1.

of the loop is not longword aligned, performance can vary by 50% or more. Second, the VAX-11/730 uses a direct-mapped translation buffer as described in Sections E.2.1 and C.2. As a result, if the pages used by a benchmark all map to the same location in the TB, the performance results could be thrown off by large numbers of TB misses. In one experiment, the page containing the instruction loop, the page containing the C-stack, and the page containing the data stack all inadvertently mapped to the same location in the TB. This meant that every instruction in the benchmark took three TB misses for every time around the loop! This level of thrashing doubled the time for that experiment, making the results useless, until I reorganized the benchmark to avoid this inadvertent addressing clash. As a result, my benchmarks are extremely artificial and do not reflect real applications that may have such addressing clashes.

Further, the experiments may not have used the ideal alignments, so changes in instruction alignment could easily change my performance results in either direction. These alignment problems occur on all virtual memory machines and particularly on VAX computers, where instructions may start on arbitrary byte boundaries. As a result, they make realistic benchmarking a very difficult task, and they throw significant doubts on all published benchmarks that try to compare performance between different computer architectures.

The SCAP architecture contains three different optimizations for cross-domain calls. The most significant optimization is the use of ASNs in the translation buffer. That improvement is the first to choose in trying to improve cross-domain call performance. The next optimization is saving, clearing, and restoring registers selectively. This optimization has clear performance benefits, given the use of a trusted linker. The third optimization is the conditional clear on return. This optimization was not broken out in the experiments, because it involves only avoiding one extra memory write of a capability.[7] Conditional clear-on-return is clearly less significant than the other two optimizations, but once there is a trusted linker for generating call and return sequences, conditional clear-on-return provides an incremental performance improvement at no significant cost in added complexity. That incremental improvement, when spread over the millions of cross-domain calls executed per day, will become visible.

[7]It could be two writes, if capabilities are larger than the unit of memory transfer.

Chapter 19

Real-Time Issues

This chapter presents a design for interrupt handling in SCAP. The design is based on the way the VAX handles interrupts. The primary goals are to support very small interrupt handlers that respond very quickly and to support more complex interrupt handlers that may cross domain boundaries while handling an interrupt.

19.1 Idealized Interrupt Handling

Ideally, each interrupt would be handled by a full-fledged domain with its own address space. The processor would have a list of enter capabilities, one for each possible interrupt, and, upon receipt of an interrupt, the processor's interrupt priority level would be raised and a cross-domain call executed using the appropriate enter capability. Thus, the list of enter capabilities would be analogous to the System Control Block (SCB) of the VAX architecture.

The drawback of this idealized interrupt strategy is that the time needed to actually schedule an interrupt handler may be lengthy, and it is often crucial to handle an interrupt before the data associated with that interrupt is lost. Flight-control software and nuclear-reactor-control software are examples where real-time response is crucial. Rapid response to interrupts also can be important in a time-shared environment to maximize overall system throughput and to keep I/O devices busy.

Performing a cross-domain call requires saving and restoring the contents of many processor registers. Because interrupts may occur at any time, no assumptions can be made about what registers are in use at the time of an interrupt. Saving all the registers in a modern CPU can require a large number of memory cycles, quite adversely affecting the interrupt-latency time. Of course, a trusted interrupt handler might only save the registers that it actually intends to use.

One technique to reduce the interrupt latency is to dedicate a special set of registers to be used only by interrupt handlers. When the interrupt occurs, the processor simply switches to the alternate set of registers, leaving undisturbed

the registers of the currently running job.[1] A machine with multiple interrupt priority levels, such as the VAX (See Section D.6.), would need a set of registers for at least the highest priority levels. Lower priority levels could save and restore the registers they use, as is currently done for all priority levels.

19.2 SCAP Hardware Interrupt Handling

The hardware of SCAP handles interrupts just as the VAX does. There will be 32 interrupt priority levels (IPLs) numbered from 0 (lowest) to 31 (highest). When an interrupt occurs, the processor raises its IPL to the level of the interrupt, fetches the appropriate vector from the System Control Block (SCB), switches to the interrupt stack if specified in the vector, and transfers control to the specified address in kernel mode. The code that runs in the interrupt handlers specified in the SCB is intended to be quite short and simple. The code might perform simple tasks, such as updating the clock or echoing characters, but any more complex tasks should be handled in a separate domain. The buffers used by the interrupt routines are protected from normal user code, by being accessible only in kernel mode. The interrupt handers will effectively execute in a common domain and will have to trust one another. Since the interrupt handlers will have to be part of the security kernel, this seems an effective tradeoff of least privilege for improved performance. As soon as the interrupt handler issues a cross-domain call, as described below in Section 19.3, the domain separation will be reinstated.

This simple approach to interrupt handling is very different from the way most previous capability systems have handled interrupts. I have chosen this approach to support real-time computations that may require very small interrupt-latency times. Appendix F summarizes how other capability systems have dealt with the interrupt handling problem.

19.3 Cross-Domain Calls at Elevated IPL

SCAP permits cross-domain calls at elevated IPLs, but only to domains that are part of the security kernel. The VAX architecture, described briefly in Appendix D, uses an interrupt stack to provide a unique per-processor context to handle interrupts, regardless of what other software may be running at the time of the interrupt. This unique context allows the scheduler to switch to some other process as a result of the interrupt, without losing the kernel-stack context of the process that was running at the time of the interrupt.

[1]The Ferranti Atlas computer [127] dedicated 9 out of 128 index registers (called B–lines) for the exclusive use of interrupt handlers. Although other programs were free to use the registers, the contents of the registers could change during any interrupt.

For SCAP, the IPL value will be preserved across the call, but the called domain will run on its own stack, rather than either the interrupt stack or the stack of the caller. Only code in the interrupt handlers pointed to by the SCB will run on the interrupt stack.

Likewise, SCAP must provide an interrupt C-stack (or IC-stack) to allow cross-domain calls out of an interrupt handler, without interfering with the C-stack of the job that is running at the time of the interrupt. A domain called at elevated IPL may further raise and then lower IPL, but it must not lower IPL below the base level at which it was called. Lowering IPL below the base level must be forbidden to avoid incorrect stacking of device interrupts. (See [138, Chapter 5, page 229] for more details.)

For simplicity, a domain that raises IPL is required to lower IPL to the previous level before executing a cross-domain return instruction. The cross-domain return will check that the IPLs of the returning domain and of the domain to which it returns are identical. If the IPLs are not identical, then the instruction will take a reserved-operand fault. The cross-domain return could just restore the IPL of the domain to which the return is happening. However, this would needlessly complicate the instruction by requiring that it check for interrupts at intermediate IPL levels, just as the MTPR (Move to Processor Register) to IPL and REI (Return from Exception or Interrupt) instructions do. Since most cross-domain returns will be at matching IPLs, requiring the execution of an extra MTPR if they do not will result in a simpler cross-domain return instruction and better performance for most cross-domain returns.

19.4 Software Interrupts and ASTs

The REI and cross-domain-return instructions in SCAP check for pending software interrupts, just as in the VAX architecture. Software interrupts are vectored to handlers in the system-wide context, just as any other interrupts would be.

Asynchronous system traps (ASTs), however, are constructs specific to protection-ring architectures. Their first purpose is to signal asynchronous events that cannot be handled currently, but should be handled prior to returning to a less privileged protection ring. This need was first recognized in the Multics protection-ring architecture [192] that provides a ring-alarm register that can be used to request a fault prior to returning to a less privileged protection ring. The VAX ASTLVL register is a direct analog of the Multics ring-alarm register. Because there is no direct mapping of the ring-alarm concept to a non-hierarchic domain system, SCAP will not check for ASTs during cross-domain calls or returns. The REI instruction will continue to check for ASTs, so that any code within a single domain that uses the protection rings will continue to work correctly. Such code would exist only if SCAP were to provide a virtual-machine

environment for compatibility with existing VAX operating systems, as discussed in Section H.3.

The second purpose of ASTs, as used in the VAX/VMS operating system, is to simulate multi-tasking within the context of a single process [221, Chapter 5] by requesting that an AST be associated with an asynchronous event. When the event in question occurs and the process enters the operating system to await any event at all, the AST occurs and code can be executed to handle the event. This second purpose for ASTs is a reflection of the lack of light-weight processes in the VAX/VMS operating system.[2] A much less complex way to handle an asynchronous event is to schedule a light-weight process to wait for the event. SCAP processes, as defined in Section 6.1.2, are specifically designed to be cheap enough to dedicate one to every possible asynchronous event. Therefore, the SCAP operating system will have no need for ASTs for either returning across domain boundaries or for handing asynchronous events.

19.5 Operation of the Scheduler

To illustrate the use of the C-stack and IC-stack in interrupt handling, this section presents two examples of how a job might enter the scheduler, either as the result of an interrupt (from I/O completion or expiration of a scheduler quantum) or as the result of an explicit request to wait for some event.

19.5.1 Scheduling Due to Interrupts

Assume that a user job is running and has a chain of cross-domain calls nested on its C-stack. An interrupt occurs, either due to I/O completion or due to a timer runout, reflecting the end of the job's time slice.

When the interrupt occurs, the processor behaves exactly as a normal VAX CPU. The current mode is switched to kernel, the PSL (processor status long-word) is marked for the interrupt stack, and the stack pointer is saved and replaced by the value of the interrupt stack pointer (ISP).[3] The program counter (PC) is loaded from the value in the appropriate vector in the System Control Block (SCB). To improve interrupt response, the interrupt-handler code must be mapped into the address space of every domain at the same location. Those pages will be mapped only for kernel-mode access, to prevent tampering. Note that the VAX architecture's concept of system space does this form of universal mapping, but not just for interrupt handlers.

[2] Multics had an analogous mechanism, called interprocess signals (IPS), to deal with the same problem of multi-tasking in an single, expensive process.

[3] Actually, switching to the interrupt stack is controlled by a bit in the vector in the System Control Block.

The code of the interrupt handler stores any data associated with the interrupt, and for this example, decides that it must invoke the scheduler.[4] The interrupt handler does a special cross-domain call to the scheduler. The special call saves the state of the current domain on the C-stack of the current job, but then switches to the IC-stack before entering the called domain.[5] The scheduler domain performs whatever functions are required to select the next process to run, possibly including saving the entire process state. The scheduler then switches to the C-stack of the new job and does a cross-domain return to restore the state of the domain of the new job. Note that by switching to the IC-stack, the interrupt handler may call any series of domains before the scheduler domain is actually invoked to switch jobs. Of course, when the final cross-domain return is executed on the IC-stack, the instruction will have to switch back to the C-stack, based on a flag in the IC-stack frame.

19.5.2 Scheduling Due to Explicit Calls

The scheduler can also be entered by an explicit cross-domain call to wait for some event. In this case, the scheduler itself would issue the special cross-domain call that would save the state of the current job on the C-stack and switch to the IC-stack. Once on the IC-stack, the scheduler would operate in the same way as for the case of entry on interrupt.

19.6 Summary of Stack Usage

To summarize the stack usage in SCAP, each domain will have a stack corresponding to each of the access modes. On a modified VAX, there would be four stacks, one each for kernel, executive, supervisor, and user modes. On a RISC processor, there would be only kernel and user modes. Each job will have a C-stack to record cross-domain activations. For each processor, there will be an interrupt stack for use only in interrupt handlers and an interrupt C-stack (IC-stack) to record cross-domain activations from interrupt handlers.

[4]For simplicity, this example has ignored the distinctions between hardware and software interrupt priority levels. In fact, the hardware interrupt handler would simply store some data, request a software interrupt at lower IPL, and do an REI instruction. Later, the software interrupt handler would actually invoke the scheduler, after allowing other hardware interrupts to be handled.

[5]Entering the scheduler from an interrupt handler is a very frequent event in most operating systems. Additional specialized optimizations may be required here. These optimizations will be dependent on the precise implementation of the scheduler.

19.7 Real-Time Processing

This chapter has discussed various techniques for reducing interrupt-latency times, yet still allowing cross-domain calls in interrupt handlers. Actually supporting real-time applications requires much more than just good interrupt-latency times. Low-overhead paths through the scheduler and other critical parts of the operating system, direct control over main-memory residence, and many other factors all contribute to a good real-time system. Achieving all of these goals within a capability-based operating system is a research topic outside the scope of this dissertation. SCAP is designed to improve the performance of capability systems, but it cannot claim to provide full real-time solutions.

Chapter 20

Conclusions

20.1 Major Research Results

This dissertation contains several significant research results about the design of capability systems that are more secure and provide better performance than existing capability designs.

The most significant result is that highly-secure capability systems can be constructed from conventional computer systems. The principal hardware requirements are a large virtual address space and good support in the virtual-address-translation hardware for rapid and frequent changes of address space.

Highly specialized hardware and microcode for supporting capabilities are likely to make the system slower and make security verification more difficult. The strong arguments for hardware support for capabilities came from systems, such as Hydra [238] and CAL [130], that did not have either large virtual address spaces or good ways to switch address spaces quickly.

Application of good compiler optimization techniques can eliminate the need for the typical hardware capability features, such as memory tagging and segment-bounds checking. This dissertation describes cross-domain-call optimization techniques, using a trusted linker, that significantly improve the context-switching performance with no hardware support required beyond address space numbers in the translation buffer.

The second major result is that programming generality can easily become a performance trap. A security system must be designed to meet the actual requirements, rather than to conform to abstract standards of elegance. It is better to make security boundaries explicitly visible to the programmers, so that they can design for the performance limitations, rather than make security completely transparent and end up with unacceptably bad performance.

Third, there is a resolution to the long-standing debate between advocates of access-control-list systems and capability systems. By changing the definition of a capability from necessary and sufficient to gain access to only necessary to gain access, the new secure capabilities can be used to solve the confinement

problem and the traceability-of-access problem and to better implement immediate revocation. These problems, for which capability systems were frequently criticized, can be solved without giving up the major advantages of capabilities for constructing small domains of protection for use as abstract type managers.

20.2 Problem Resolution

It is important to recognize that not all the problems raised in this dissertation have been solved. While many have been solved, others remain and may well be unsolvable.

For example, effective security enforcement cannot be achieved at no cost in performance. The cost of changing protection domains will always be higher than that of remaining in the same protection domain. This dissertation proposes several ways to reduce that cost and to make that cost tolerable, compared to the benefits gained.

Likewise, complete software compatibility can only be achieved at some cost in either performance or security effectiveness, or both. Many existing operating systems have features that are inherently insecurable, and maintaining full compatibility with such features is impossible in a highly secure system. Maintaining full compatibility with certain software features may have a very high cost in performance. In such cases, engineering trade-offs will be required to achieve a capability system that can be commercially successful.

20.3 The Next Step

This dissertation has developed the basic SCAP architecture and has shown prototypical performance results for some of its key components. There is still much research and development needed before SCAP can be considered a proven technology. In particular, the issues of software compatibility between the models of conventional operating systems and the capability, object-oriented models have not been resolved. Appendix H suggests techniques for achieving at least some compatibility.

The next step to take is the construction of a domain-structured security kernel to run on either a RISC processor or on a VAX processor with higher performance than the VAX-11/730. Unlike previous security kernel development projects, however, the development of the SCAP kernel must include development of compiler and linker tools, such as those described in Chapter 17 to gain the necessary level of performance. New programming languages are not required, but the code generation and linkage conventions must be specially tailored to make context switching as fast as possible.

References

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tukubo, and D. A. Webb. *Security Analysis and Enhancements of Computer Operating Systems.* NBSIR 76-1041, The RISOS Project, Lawrence Livermore Laboratory, Livermore, CA, USA, April 1976. Published by the Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, DC, USA.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: a new kernel foundation for UNIX development. In *Proceedings of Summer USENIX*, July 1986.

[3] W. B. Ackerman and W. W. Plummer. An implementation of a multiprocessing computer system. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Gatlinburg, TN, USA, October 1967.

[4] Katherine Addison, Larry Baron, Mark Copple, Don Cragun, Keith Hospers, Patricia Jordan, Mikel Lechner, Michael Manley, and Case Schaufler. Computer security at Sun Microsystems, Inc. In *Proceedings of the* 10^{th} *National Computer Security Conference*, pages 216–219, National Bureau of Standards, Gaithersburg, MD, USA, 21–24 September 1987.

[5] $\aleph_0$. Computer recreations. *Software—Practice and Experience*, 1(2):201–204, April—June 1971.

[6] James P. Anderson. *Computer Security Technology Planning Study.* Technical Report ESD–TR–73–51, Vols. I and II, James P. Anderson and Co., Fort Washington, PA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, October 1972.

[7] M. Anderson, R. D. Pose, and C. S. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, February 1986.

[8] Poul Anderson. Sam Hall. *Astounding Science Fiction*, 51(6):9+, August 1953. Reprinted in Isaac Asimov, Martin H. Greenberg, and Charles G. Waugh, editors, *Computer Crimes and Capers*, pages 144–172, Academy Chicago Publishers, Chicago, IL, USA, 1983.

[9] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[10] C. Richard Attanasio, Peter W. Markstein, and Ray J. Phillips. Penetrating an operating system: a study of VM/370 integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[11] Özap Babaoğlu and William Joy. Converting a swap-based system to do paging in an architecture lacking page referenced bits. *Operating Systems Review*, 15(5):78–86, December 1981. Proceedings of the Eighth Symposium on Operating Systems Principles, Asilomar Conference Grounds, Pacific Grove, CA, USA, 14–16 December 1981.

[12] D. W. Barron, A. G. Fraser, D. F. Hartley, B. Landy, and R. M. Needham. File handling at Cambridge University. In *AFIPS Conference Proceedings, Volume 30, 1967 Spring Joint Computer Conference*, pages 163–167, Thompson Books, Washington, DC, USA, 1967.

[13] D. E. Bell, R. S. Fiske, M. Gasser, and P. S. Tasker. *Secure On–Line Processing Technology—Final Report.* Technical Report ESD–TR–74–186, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, August 1974.

[14] David E. Bell and Leonard J. LaPadula. *Computer Security Model: Unified Exposition and Multics Interpretation.* Technical Report ESD–TR–75–306, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, June 1975.

[15] David E. Bell and Leonard J. LaPadula. *Secure Computer Systems: Mathematical Foundations.* Technical Report ESD–TR–73–278, Vol. I, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, November 1973.

[16] David E. Bell and Leonard J. LaPadula. *Secure Computer Systems: A Mathematical Model.* Technical Report ESD–TR–73–278, Vol. II, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, November 1973.

[17] David E. Bell. *Secure Computer Systems: A Refinement of the Mathematical Model.* Technical Report ESD–TR–73–278, Vol. III, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, April 1974.

[18] Kenneth J. Biba. *Integrity Considerations for Secure Computer Systems.* Technical Report ESD–TR–76–372, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, April 1977.

[19] Joel S. Birnbaum and William S. Worley, Jr. Beyond RISC: high-precision architecture. *Hewlett-Packard Journal*, 36(8):4–10, August 1985. Also published in *Digest of Papers, Compcon Spring 86*, IEEE Computer Society, San Francisco, CA, USA, 3–6 March 1986, pages 40–47.

[20] Andrew D. Birrell. *System Programming in a High Level Language.* Ph. D. dissertation, Technical Report No. 6, Computer Laboratory, University of Cambridge, Cambridge, England, December 1977.

[21] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[22] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* Ph. D. thesis, Department of Electrical Engineering and Computer Science, MIT/LCS/TR–178, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, May 1977.

[23] Steven Blotcky, Kevin Lynch, and Steven Lipner. SE/VMS: implementing mandatory security in VAX/VMS. In *Proceedings of the* 9^{th} *National Computer Security Conference*, pages 47–54, National Bureau of Standards, Gaithersburg, MD, USA, 15–18 September 1986.

[24] W. E. Boebert. On the inability of an unmodified capability machine to enforce the $*$–property. In *Proceedings of the* 7^{th} *DoD/NBS Computer Security Conference*, pages 291–293, National Bureau of Standards, Gaithersburg, MD, USA, 24–26 September 1984.

[25] W. E. Boebert and C. T. Ferguson. A partial solution to the discretionary Trojan horse problem. In *Proceedings of the* 8^{th} *National Computer Security Conference*, pages 141–144, National Bureau of Standards, Gaithersburg, MD, USA, 30 September – 3 October 1985.

[26] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the* 8^{th} *National Computer Security Conference*, pages 18–27, National Bureau of Standards, Gaithersburg, MD, USA, 30 September – 3 October 1985.

[27] W. E. Boebert, R. Y. Kain, and W. D. Young. Secure computing: the Secure Ada Target approach. *Scientific Honeyweller*, 6(2):1–27, July 1985.

[28] W. E. Boebert, W. D. Young, R. Y. Kain, and S. A. Hansohn. Secure Ada target: issues, system design, and verification. In *Proceedings of the 1985 Symposium on Security and Privacy*, pages 176–183, IEEE Computer Society, Oakland, CA, USA, 22–24 April 1985.

[29] Richard G. Bratt, Gerald F. Clancy, Craig J. Mundie, Stephen I. Schleimer, and Steven J. Wallach. *Data Processing System Having a Memory Using*

Object-Based Information and a Protection Scheme for Determining Access Rights to Such Information. United States Patent No. 4,525,780, 25 June 1985.

[30] Thierry Breton and Denis Beneich. *Softwar.* Futura Publications, a division of Macdonald & Co., London, England, 1987. Mark Howson, English translator. Originally published by Editions Robert Laffont S. A., Paris, France, 1984.

[31] B. R. S. Buckingham. *The SWARD Command Language (CL/SWARD).* Technical Report TR-73-011, IBM Systems Research Institute, New York, NY, February 1981.

[32] Steve Bunch. The SETUID feature in UNIX and security. In *Proceedings of the 10^{th} National Computer Security Conference*, pages 245–253, National Bureau of Standards, Gaithersburg, MD, USA, 21–24 September 1987.

[33] Brian Case. 32-bit microprocessor opens system bottlenecks. *Computer Design*, 26(7):79–86, 1 April 1987.

[34] G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, Boston, MA, USA, 23–25 June 1982.

[35] Albert Chang and Mark F. Mergen. 801 storage: architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[36] Maureen Harris Chehyl, Morrie Gasser, George A. Huff, and Jonathan K. Millen. Verifying security. *ACM Computing Surveys*, 13(3):279–339, September 1981.

[37] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. *SIGPLAN Notices*, 19(6):222–232, June 1984. Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, Montreal, Quebec, Canada, 17–22 June 1984.

[38] David D. Clark. *Ancillary Reports: Kernel Design Project.* Technical Memorandum MIT/LCS/TM–87, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1977.

[39] David D. Clark, Robert M. Graham, Jerome H. Saltzer, and Michael D. Schroeder. *The Classroom Information and Computing Service.* Project MAC TR–80, Massachusetts Institute of Technology, Cambridge, MA, USA, 11 January 1971.

[40] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194, IEEE Computer Society, Oakland, CA, USA, 27–29 April 1987.

[41] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: simulation and measurement. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[42] *CLIPPER 32-Bit Microprocessor: User's Manual.* Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1987.

[43] Fred Cohen. Computer viruses—theory and experiments. In *Proceedings of the 7^{th} DoD/NBS Computer Security Conference*, pages 240–263, National Bureau of Standards, Gaithersburg, MD, USA, 24–26 September 1984.

[44] Robert P. Colwell. *The Performance Effects of Functional Migration and Architectural Complexity in Object-Oriented Systems.* Ph. D. thesis, Department of Computer Science, CMU-CS-85-159, Carnegie-Mellon University, Pittsburgh, PA, USA, August 1985.

[45] Douglas John Cook. *The Evaluation of a Protection System.* Ph. D. dissertation, Computer Laboratory Technical Report No. 9, University of Cambridge, Cambridge, England, April 1978.

[46] Hugo Cornwall. *The Hacker's Handbook.* Century Communications, Ltd., London, England, 1985.

[47] Hugo Cornwall. *Hacker's Handbook III.* Century Hutchinson, Ltd., London, England, 1988.

[48] D. C. Cosserat. A capability oriented multi-processor system for real-time applications. In *Proceedings of the International Conference on Computer Communications*, pages 282–289, Washington, DC, USA, 24–26 October 1972.

[49] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *AFIPS Conference Proceedings, Volume 27, Part I, 1965 Fall Joint Computer Conference*, pages 213–229, Spartan Books, Washington, DC, USA, 1965.

[50] C. J. Date. *An Introduction to Database Systems.* Volume I, Addison-Wesley Publishing Company, Reading, MA, USA, fourth edition, 1986.

[51] C. J. Date. *An Introduction to Database Systems.* Volume II, Addison-Wesley Publishing Company, Reading, MA, USA, 1983.

[52] L. DeLashmutt. Trusted computing research at Data General Corporation. In *Proceedings of the Fourth Seminar on the DoD Computer Security Initiative*, pages J–1 – J–21, National Bureau of Standards, Gaithersburg, MD, USA, 10–12 August 1981.

[53] Carl Nigel Robert Dellar. *The Distribution of Operating System Functions.* Ph. D. dissertation, Computer Laboratory, University of Cambridge, Cambridge, England, September 1980.

[54] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[55] M. DeMoney, J. Moore, and J. Mashey. Operating system support on a RISC. In *Digest of Papers, Compcon Spring 86*, pages 138–143, IEEE Computer Society, San Francisco, CA, USA, 3–6 March 1986.

[56] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[57] Jack B. Dennis. Programming generality, parallelism and computer architecture. In *IFIP Congress 68, Booklet C, Software 2*, pages C1–C7, North Holland Publishing Company, Amsterdam, The Netherlands, August 1968.

[58] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[59] *Department of Defense Trusted Computer System Evaluation Criteria.* DOD 5200.28-STD, Department of Defense, Washington, DC, USA, December 1985.

[60] Yvo Desmedt. Is there an ultimate use of cryptography? In Andrew M. Odlyzko, editor, *Advances in Cryptology—CRYPTO '86*, pages 459–463, Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science, Volume 263.

[61] Daniel W. Dobberpuhl, Robert M. Supnik, and Richard T. Witek. The MicroVAX 78032 chip, a 32-bit microprocessor. *Digital Technical Journal*, (2):12–23, March 1986.

[62] R. W. Doran. *Computer Architecture: A Structured Approach.* Academic Press, London, England, 1979.

[63] Deborah D. Downs, Jerzy R. Rub, Kenneth C. Kung, and Carole S. Jordan. Issues in discretionary access control. In *Proceedings of the 1985 Symposium on Security and Privacy*, pages 208–218, IEEE Computer Society, Oakland, CA, USA, 22–24 April 1985.

[64] *DPS 8 & DPS 88 Assembly Instructions.* Order Number DH03-01, Honeywell Information Systems Inc., Waltham, MA, USA, June 1984.

[65] D. M. England. Architectural features of system 250. In *Operating Systems: Infotech State of the Art Report 14*, pages 395–427, Infotech Information Ltd., Maindenhead, England, 1972.

[66] Deborah Lynn Estrin. *Access to Inter-Organization Computer Networks.* Ph. D. thesis, Department of Electrical Engineering and Computer Science, MIT/LCS/TR–345, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, August 1985.

[67] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[68] R. J. Feiertag and E. I. Organick. The Multics input/output system. *Operating Systems Review*, 6(1,2):35–41, June 1972. Proceedings of the Third Symposium on Operating Systems Principles, Stanford University, Palo Alto, CA, 18–20 October 1971.

[69] Eduardo B. Fernández, Rita C. Summers, and Christopher Wood. *Database Security and Integrity.* Addison-Wesley Publishing Company, Reading, MA, USA, 1981.

[70] Harry C. Forsdick and David P. Reed. Patterns of security violations: multiple references to arguments. In David D. Clark, editor, *Ancillary Reports: Kernel Design Project*, pages 34–49, MIT/LCS/TM-87, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1977.

[71] Tryggve Fossum, James B. McElroy, and William English. An overview of the VAX 8600 system. *Digital Technical Journal*, (1):8–23, August 1985.

[72] J. M. Foster, I. F. Currie, and P. W. Edwards. *Flex: A Working Computer with an Architecture Based on Procedure Values.* RSRE Memorandum No. 3500, Royal Signals & Radar Establishment, Malvern, England, July 1982.

[73] A. G. Fraser. User control in a multi-access system. *The Computer Journal*, 11(1):12–16, 1968.

[74] John Fu, James B. Keller, and Kenneth J. Haduch. Aspects of the VAX 8800 C box design. *Digital Technical Journal*, (4):41–51, February 1987.

[75] *GEC 41XX Series Nucleus.* DD1622, Issue 2, GEC Computers Limited, Borehamwood, Herts, England, August 1983.

[76] Edward F. Gehringer. *Capability Architectures and Small Objects.* UMI Research Press, Ann Arbor, MI, USA, 1982.

[77] Edward F. Gehringer and Robert P. Colwell. Fast object-oriented procedure calls: lessons from the Intel 432. *Computer Architecture News*, 14(2):92–101, June 1986. The 13th Annual International Symposium on Computer Architecture Conference Proceedings, Tokyo, Japan, 2–5 June 1986.

[78] C. G. Girling. Object representation on a heterogeneous network. *Operating Systems Review*, 16(4):49–59, October 1982.

[79] Virgil D. Gligor, C. S. Chandersekaran, Robert S. Chapman, Leslie J. Dotterer, Matthew S. Hecht, Wen–Der Jiang, Abhai Johri, Gary L. Luckenbaugh, and N. Vasudevan. Design and implementation of secure Xenix. *IEEE Transactions on Software Engineering*, SE-13(2):208–221, February 1987.

[80] B. D. Gold, R. R. Linde, R. J. Peeler, M. Schaefer, J. F. Scheid, and P. D. Ward. A security retrofit of VM/370. In *AFIPS Conference Proceedings, Volume 48, 1979 National Computer Conference*, pages 335–344, AFIPS Press, Montvale, NJ, USA, 1979.

[81] G. Scott Graham and Peter J. Denning. Protection—principles and practice. In *AFIPS Conference Proceedings, Volume 40, 1972 Spring Joint Computer Conference*, pages 417–429, AFIPS Press, Montvale, NJ, USA, 1972.

[82] Frederick T. Grampp and Robert H. Morris. *UNIX* operating system security. *AT&T Bell Laboratories Technical Journal*, 63(8):1649–1672, October 1984.

[83] J. N. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, chapter 3.F, pages 393–481, Springer-Verlag, Berlin, 1979.

[84] Jerrold M. Grochow. MOO in Multics. *Software—Practice and Experience*, 2(3):303–304, July—September 1972.

[85] *Guide to VAX/VMS File Applications.* Order No. A1–Y508B–TE, Digital Equipment Corporation, Maynard, MA, USA, April 1986.

[86] *Guide to VAX/VMS System Security.* Order No. AA–Y510A–TE, AA–Y510A–T1, Digital Equipment Corporation, Maynard, MA, USA, July 1985.

[87] D. Halton. Hardware of the system 250 for communication control. In *International Switching Symposium*, Massachusetts Institute of Technology, Cambridge, MA, USA, 6-9 June 1972.

[88] Paul M. Hansen, Mark A. Linton, Robert N. Mayo, Marguerite Murphy, and David A. Patterson. A performance evaluation of the Intel iAPX 432. *Computer Architecture News*, 10(4):17–26, June 1982.

[89] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[90] B. Hebbard, P. Grosso, T. Baldridge, C. Chan, D. Fishman, P. Goshgarian, T. Hilton, J. Hoshen, K. Hoult, G. Huntley, M. Stolarchuk, and L. Warner. A penetration of the Michigan terminal system. *Operating Systems Review*, 14(1):7–20, January 1980.

[91] Andrew J. Herbert, editor. *CAP Operating System Manual.* University of Cambridge Computer Laboratory, Cambridge, England, 13 January 1978.

[92] Andrew J. Herbert, editor. *CAP System Programmers' Manual.* University of Cambridge Computer Laboratory, Cambridge, England, 13 January 1978.

[93] Andrew J. Herbert. A hardware-supported protection architecture. In D. Lanciaux, editor, *Operating Systems: Theory and Practice*, pages 293–306, North-Holland Publishing Co., Amsterdam, The Netherlands, 1979. Proceedings of the Second International Symposium on Operating Systems Theory and Practice.

[94] Andrew J. Herbert. *Microcode Kernel Specification.* CAP-III Technical Memorandum, Computer Laboratory, University of Cambridge, Cambridge, England, 6 September 1982.

[95] Andrew J. Herbert. *A Microprogrammed Operating System Kernel.* Ph. D. dissertation, Computer Laboratory, University of Cambridge, Cambridge, England, November 1978.

[96] Andrew J. Herbert. MOO on the CAP computer. *Software—Practice and Experience*, 7(6):797–798, November—December 1977.

[97] Andrew J. Herbert and Roger M. Needham. Sequencing computation steps in a network. *Operating Systems Review*, 15(5):59–63, December 1981. Proceedings of the Eighth Symposium on Operating Systems Systems Principles, Asilomar Conference Grounds, Pacific Grove, CA, USA, 14–16 December 1981.

[98] Phillip D. Hester, Richard O. Simpson, and Albert Chang. The IBM RT PC ROMP and memory management unit architecture. In *IBM RT Personal Computer Technology*, pages 48–56, SA23–1057, IBM Engineering Systems Products, Milford, CT, USA, 1986.

[99] T. H. Hinke and M. Schaefer. *Secure Data Management System.* Technical Report TM-(L)-5407/007/00, System Development Corporation, Santa Monica, CA, USA, June 1975.

[100] M. E. Houdek and G. R. Mitchell. Translating a large virtual address. In *IBM System/38 Technical Developments*, pages 22–24, G580–0237–1, IBM General Systems Division, Atlanta, GA, USA, 1980.

[101] *iAPX 432 General Data Processor Architecture Reference Manual.* Intel Corporation, Aloha, OR, USA, 1981.

[102] *IBM System/38 Functional Concepts Manual.* GA21-9330-5, International Business Machines Corporation, Rochester, MN, USA, November 1986.

[103] *IBM System/38 Functional Reference Manual—Volume 1.* GA21-9331-5, International Business Machines Corporation, Rochester, MN, USA, June 1984.

[104] *IBM System/38 Technical Developments.* G580–0237–1, IBM General Systems Division, Atlanta, GA, USA, 1980.

[105] *Introduction to KeyKOS Concepts.* KL004-06, Key Logic, Inc., Santa Clara, CA, USA, November 1986.

[106] *Introduction to VAX/VMS System Routines.* Order Number: AA–Z500A–TE, Digital Equipment Corporation, Maynard, MA, USA, September 1984.

[107] Philippe A. Janson. *Removing the Dynamic Linker from the Security Kernel of a Computing Utility.* S.M. and E.E. thesis, Department of Electrical Engineering, MAC TR–132, Project MAC, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1974.

[108] Philippe A. Janson. *Using Type Extension to Organize Virtual Memory Mechanisms.* Ph. D. thesis, Department of Electrical Engineering and Computer Science, MIT/LCS/TR–167, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1976.

[109] Martyn A. Johnson. Computer Laboratory, University of Cambridge, Cambridge, England. Private Communication, January 1988.

[110] Mike Johnson. *Am29000 Streamlined Instruction Processor User's Manual.* TD-WCT 10 K 2/87, PID 08996A, Advanced Micro Devices, Sunnyvale, CA, USA, 1987.

[111] Jay Jonekait. GNOSIS: a secure capability based 370 operating system. In *Proceedings of the Third Seminar on the DoD Computer Security Initiative*, pages G–1 – G–16, National Bureau of Standards, Gaithersburg, MD, USA, 18–20 November 1980.

[112] Clifford E. Kahn. Digital Equipment Corporation, Littleton, MA, USA. Private Communication, 3 December 1987.

[113] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, SE–13(2):202–207, February 1987.

[114] Paul A. Karger. Authentication and discretionary access control in computer networks. *Computer Networks and ISDN Systems*, 10(1):27–37, 1985.

[115] Paul A. Karger. Implementing commercial data integrity with secure capabilities. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 130–139, IEEE Computer Society, Oakland, CA, USA, 18 – 20 April 1988.

[116] Paul A. Karger. The lattice security model in a public computing network. In *ACM 78: Proceedings 1978 Annual Conference*, pages 453–459, Association for Computing Machinery, Washington, DC, USA, 4–6 December 1978.

[117] Paul A. Karger. Limiting the damage potential of discretionary trojan horses. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 32–37, IEEE Computer Society, Oakland, CA, USA, 27 – 29 April 1987.

[118] Paul A. Karger. *Non-Discretionary Access Control for Decentralized Computing Systems.* S.M. thesis, Department of Electrical Engineering and Computer Science, MIT/LCS/TR-179, Massachusetts Institute of Technology, Cambridge, MA, USA, May 1977.

[119] Paul A. Karger and Andrew J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 2–12, IEEE Computer Society, Oakland, CA, USA, 29 April – 2 May 1984.

[120] Paul A. Karger and Steven B. Lipner. Digital's research activities in computer security. In *Proceedings of EASCON'82*, pages 29–32, IEEE, Washington, DC, USA, September 1982.

[121] Paul A. Karger and Roger R. Schell. *Multics Security Evaluation: Vulnerability Analysis.* Technical Report ESD–TR–74–193, Vol. II, HQ Electronic Systems Division, Hanscom AFB, MA, USA, June 1974.

[122] B. Jeffrey Katz, Stephen P. Morse, William B. Pohlman, and Bruce W. Revenel. 8086 microcomputer bridges the gap between 8- and 16-bit designs. *Electronics*, 51(4):99–104, 16 February 1978.

[123] R. H. Katz, S. J. Eggers, D. A. Wood, , C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. *Computer Architecture News*, 13(3):276–283, June 1985. The 12th Annual International Symposium on Computer Architecture Conference Proceedings, Boston, MA, USA, 17–19 June 1985.

[124] Lawrence J. Kenah, Ruth E. Goldenberg, and Simon F. Bate. *VAX/VMS Internals and Data Structures.* Order No. EY-8264E-DP, Digital Press, Bedford, MA, USA, 1988.

[125] Tracy Kidder. *The Soul of a New Machine.* Little, Brown and Company, Boston, MA, USA, 1981.

[126] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC–11(2):223–235, April 1962.

[127] T. Kilburn, R. B. Payne, and D. J. Howarth. The Atlas supervisor. In *Computers—Key to Total Systems Control, Proceedings of the Eastern Joint Computer Conference*, pages 279–294, American Federation of Information Processing Societies, Macmillan Company, New York, NY, USA, 12–14 December 1961.

[128] Donald E. Knuth. *Sorting and Searching.* Volume 3 of *The Art of Computer Programming*, Addison-Wesley Publishing Company, Reading, MA, USA, 1973.

[129] Steven Kramer. Linus IV—an experiment in computer security. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 24–32, IEEE Computer Society, Oakland, CA, USA, 29 April – 2 May 1984.

[130] B. W. Lampson and H. E. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–265, May 1976.

[131] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[132] Butler W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, January 1974. Proceedings of the Fifth Princeton Conference on Information Sciences and Systems, Princeton University, Princeton, NJ, USA, March 1971, pp. 437–443.

[133] Bill Landreth. *Out of the Inner Circle: A Hacker's Guide to Computer Security.* Microsoft Press, Bellvue, WA, USA, 1985.

[134] Carl E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[135] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. In D. Lanciaux, editor, *Operating Systems: Theory and Practice*, pages 371–384, North-Holland Publishing Co., Amsterdam, The Netherlands, 1979. Proceedings of the Second International Symposium on Operating Systems Theory and Practice, IRIA, Rocquencourt, France, 2–4 October 1978.

[136] T. M. P. Lee. Using mandatory integrity to enforce "commercial" security. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Oakland, CA, USA, 18–21 April 1988.

[137] Timothy E. Leonard. Digital Equipment Corporation, Boxborough, MA, USA. Private Communication, 3 September 1987.

[138] Timothy E. Leonard, editor. *VAX Architecture Reference Manual.* Digital Press, Bedford, MA, USA, 1987.

[139] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, MA, USA, 1983.

[140] Henry M. Levy and Douglas W. Clark. On the use of benchmarks for measuring system performance. *Computer Architecture News*, 10(6):5–8, December 1982.

[141] Henry M. Levy and Richard H. Eckhouse, Jr. *Computer Programming and Architecture: The VAX–11.* Digital Press, Bedford, MA, USA, 1980.

[142] S. B. Lipner, S. R. Beckhardt, and D. F. Stork. *A UNIX Executive for Use with the PDP–11/45 Security Kernel.* Working Paper 20056, The MITRE Corporation, Bedford, MA, USA, 5 December 1974.

[143] Steven B. Lipner. A comment on the confinement problem. *Operating Systems Review*, 9(5):192–196, November 1975. Proceedings of the Fifth Symposium on Operating Systems Principles, Unversity of Texas at Austin, Austin, TX, USA, 19–21 November 1975.

[144] Steven B. Lipner. Non-discretionary controls for commercial applications. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 2–10, IEEE Computer Society, Oakland, CA, USA, 26–28 April 1982.

[145] Steven B. Lipner. Secure system development at Digital Equipment: Targetting the needs of a commercial and government customer base. In *Proceedings of the 8th National Computer Security Conference*, pages 120–123, DoD Computer Security Center and National Bureau of Standards, Gaithersburg, MD, USA, 30 September – 3 October 1985.

[146] Jerome Lobel. *Foiling the System Breakers.* McGraw-Hill Book Company, New York, NY, USA, 1986.

[147] T. Mark A. Lomas. Computer Laboratory, University of Cambridge, Cambridge, England. Private Communication, 14 January 1988.

[148] Loopholes in IBM System/38. *Computer Fraud & Security Bulletin*, 5(6):1–5, April 1983.

[149] N. Lourie, H. Schrimpf, R. Reach, and W. Kahn. Arithmetic and control techniques in a multiprogram computer. In *Proceedings of the Eastern Joint Computer Conference*, pages 75–81, Boston, MA, USA, 1–3 December 1959.

[150] G. A. Mann. Software design implications of a domain architecture. In *Proceedings of the Third Annual International Conference on Computers and Communications*, pages 144–150, IEEE Computer Society, Phoenix, AZ, USA, 19–21 March 1984.

[151] Andrew H. Mason. *A Layered Virtual Memory Manager.* S.M. and E.E. thesis, Department of Electrical Engineering and Computer Science, MIT/LCS/TR–177, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, May 1977.

[152] Mark McCain. Beating off the hacking trojans. *The Times* (London, England), (62,778):26, 29 May 1987.

[153] E. J. McCauley and P. J. Drongowski. KSOS—The design of a secure operating system. In *AFIPS Conference Proceedings, Volume 48, 1979 National Computer Conference*, pages 345–351, AFIPS Press, Montvale, NJ, USA, 1979.

[154] Thomas M. McWilliams and L. Curtis Widdoes, Jr. S–1 multiprocessor architecture. In *Advanced Digital Computing Technology Base Development for Navy Applications: The S–1 Project*, Technical Report UCID-18038, Lawrence Livermore Laboratory, University of California, Livermore, CA, USA, 30 September 1978.

[155] *MICRO2 User's Guide.* Order No. AA–H531A-TE, Digital Equipment Corporation, Maynard, MA, USA, June 1979.

[156] James G. Mitchell, William Maybury, and Richard Sweet. *Mesa Language Manual: Version 5.0.* Technical Report CSL–79–3, Xerox Palo Alto Research Center, Systems Development Department, Palo Alto, CA, USA, April 1979.

[157] Warren A. Montgomery. Measurements of sharing in Multics. *Operating Systems Review*, 11(5):2–12, November 1977. Proceedings of the Sixth ACM Symposium on Operating Systems Principles, Purdue University, West Lafayette, IN, USA, 16–18 November 1985.

[158] D. Morris and G. D. Detlefsen. An implementation of a segmented virtual store. In *Conference on Computer Science and Technology*, pages 63–71, University of Manchester Institute of Science and Technology, Institution of Electrical Engineers, London, England, 30 June – 3 July 1969. IEE Conference Publication 55.

[159] Derrick Morris and Roland N. Ibbett. *The MU5 Computer System.* Springer-Verlag, New York, NY, USA, 1979.

[160] Robert Morris. Scatter storage techniques. *Communications of the ACM*, 11(1):38–44, January 1968.

[161] J. Moussouris, L. Crudele, D. Freitas, C. Hansen, E. Hudson, R. March, S. Przybylski, T. Riodan, C. Rowen, and D. Van't Hof. A CMOS RISC processor with integrated system functions. In *Digest of Papers, Compcon Spring 86*, pages 126–131, IEEE Computer Society, San Francisco, CA, USA, 3–6 March 1986.

[162] Sape J. Mullender. *Principles of Distributed Operating System Design.* Ph. D. dissertation, Vrije Universiteit te Amsterdam, Amsterdam, The Netherlands, 31 October 1985. Published by Mathematisch Centrum, Amsterdam.

[163] W. H. Murray. Data integrity in a business data processing system. In *Proceedings of the Workshop in Integrity Policy in Computer Information Systems (WIPCIS)*, ACM SIGSAC and Bentley College, Waltham, MA, USA, 27–29 October 1987.

[164] G. Myers and B. R. S. Buckingham. A hardware implementation of capability-based addressing. *Operating Systems Review*, 14(4):13–25, October 1980.

[165] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System.* Addison-Wesley Publishing Company, London, England, 1982.

[166] Peter G. Neumann, Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson. *A Provably Secure Operating System: The System, Its Applications, and Proofs.* Computer Science Laboratory Report CSL–116, SRI International, Menlo Park, CA, USA, May 7 1980.

[167] Elliott I. Organick. *Computer System Organization: The B5700/B6700 Series.* Academic Press, New York, NY, USA, 1973.

[168] Elliott I. Organick. *The Multics System: An Examination of Its Structure.* The MIT Press, Cambridge, MA, USA, 1972.

[169] Elliott I. Organick. *A Programmer's View of the Intel 432 System.* McGraw-Hill Book Company, New York, NY, USA, 1983.

[170] Ronald Paans and Guus Bonnes. Surreptitious security violation in the MVS operating system. In Viiveke A. Fåk, editor, *Security, IFIP/Sec'83*, North-Holland Publishing Company, Amsterdam, The Netherlands, 1983. Proceedings of the IFIP First Security Conference, Stockholm, Sweden, 16–18 May 1983.

[171] J. B. D. Pardoe. Computer Laboratory, University of Cambridge, Cambridge, England. Private Communication, July 1987.

[172] David A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[173] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[174] Gerald J. Popek, Mark Kampe, Charles S. Kline, Allen Stoughton, Michael Urban, and Evelyn J. Walton. UCLA secure UNIX. In *AFIPS Conference Proceedings, Volume 48, 1979 National Computer Conference*, pages 355–364, AFIPS Press, Montvale, NJ, USA, 1979.

[175] Publisher blamed for computer virus. *Lafayette (IN, USA) Journal & Courier*, A–12, 16 March 1988.

[176] Michael O. Rabin and J. D. Tygar. *An Integrated Toolkit for Operating System Security.* Technical Report TR–05–87, Aiken Computational Laboratory, Harvard University, Cambridge, MA, USA, May 1987.

[177] S. A. Rajunas, N. Hardy, A. C. Bomberger, W. S. Frantz, and C. R. Landau. Security in KeyKOS. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 78–85, IEEE Computer Society, Oakland, CA, USA, 7–9 April 1986.

[178] Justin Rattner and William W. Lattin. Ada determines architecture of 32-bit microprocessor. *Electronics*, 54(4):119–126, 24 February 1981.

[179] David D. Redell. *Naming and Protection in Extendible Operating Systems.* Ph. D. thesis, University of California, Berkeley, CA, USA, published as Project MAC TR-140, Massachusetts Institute of Technology, Cambridge, MA, USA, November 1974.

[180] David P. Reed. *Processor Multiplexing in a Layered Operating System.* S.M. thesis, Department of Electrical Engineering and Computer Science, MIT/LCS/TR–164, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, July 1976.

[181] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

[182] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[183] A. R. Saxena. *A Verified Specification of a Hierarchical Operating System.* Technical Report 107, Digital Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA, USA, January 1976.

[184] O. Sami Saydjari, Joseph M. Beckman, and Jeffrey R. Leaman. Locking computers securely. In *Proceedings of the* 10^{th} *National Computer Security Conference*, pages 129–141, National Bureau of Standards, Gaithersburg, MD, USA, 21–24 September 1987.

[185] Marvin Schaefer, Rich Neely, Luke Dion, Hilda Faust Mathieu, Larry Robinson, Mike Soleglad, Peter Neumann, and Les Fraim. Kernel performance issues. In *Proceedings of the 1981 Symposium on Security and Privacy*, pages 162–178, IEEE Computer Society, Oakland, CA, USA, 27–29 April 1981.

[186] Roger R. Schell. Computer security: the Achilles' heel of the electronic Air Force? *Air University Review*, 30(2):16–33, January–February 1979.

[187] Roger R. Schell. A security kernel for a multiprocessor microcomputer. *Computer*, 16(7):47–53, July 1983.

[188] W. L. Schiller. *The Design and Specification of a Security Kernel for the PDP–11/45.* Technical Report ESD–TR–75–69, The MITRE Corporation, Bedford, MA, USA, HQ Electronic Systems Division, Hanscom AFB, MA, USA, May 1975.

[189] Michael D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility.* Ph. D. thesis, Department of Electrical Engineering, Project MAC TR–104, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1972.

[190] Michael D. Schroeder. The Multics kernel design project. *Operating Systems Review*, 11(5):43–56, November 1977. Proceedings of the Sixth Symposium on Operating Systems Principles, Purdue University, West Lafayette, IN, USA, 16–18 November 1985.

[191] Michael D. Schroeder. Performance of the GE–645 associative memory while Multics is in operation. In *ACM SIGOPS Workshop on System Performance Evaluation*, pages 227–245, Harvard University, Cambridge, MA, USA, 5–7 April 1971.

[192] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, March 1972.

[193] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[194] Lawrence J. Shirley. *Non-Discretionary Security Validation by Assignment.* Master's thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, USA, June 1981.

[195] Lawrence J. Shirley and Roger R. Schell. Mechanism sufficiency validation by assignment. In *Proceedings of the 1981 Symposium on Security and Privacy*, pages 26–32, IEEE Computer Society, Oakland, CA, USA, 27–29 April 1981.

[196] William R. Shockley. *Implementing the Clark/Wilson Integrity Policy Using Current Technology.* Technical Report GCI–88-6-01, Gemini Computers, Inc., P.O. Box 222417, Carmel, CA, USA, February 1988.

[197] R. L. Sites. An analysis of the CRAY–1 computer. *Computer Architecture News*, 6(7):101–106, April 1978. The 5th Annual Symposium on Computer Architecture Conference Proceedings.

[198] Christopher John Slinn. *Aspects of a Capability Based Operating System.* Ph. D. dissertation, Computer Laboratory, University of Cambridge, Cambridge, England, February 1977.

[199] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(2):473–530, September 1982.

[200] Leroy Smith. *Architectures for Secure Computing Systems.* Technical Report MTR-2772, The MITRE Corp., Bedford, MA, USA, June 1974.

[201] Margaret Stanley. *The Use of Values Without Names in a Programming Support Environment.* RSRE Memorandum 3901, Royal Signals and Radar Establishment, Malvern, Worcs., England, November 1985.

[202] Guy Lewis Steele Jr. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: the ultimate GOTO. In *Proceedings of the 1977 Annual Conference*, pages 153–162, Association for Computing Machinery, Seattle, WA, USA, 16–19 October 1977.

[203] Guy Lewis Steele Jr. *LAMBDA: The Ultimate Declarative.* AI Memo 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA, November 1976.

[204] Bjarne Stroustrup. *Communication and Control in Distributed Computer Systems.* Ph. D. dissertation, Computer Laboratory, University of Cambridge, Cambridge, England, February 1979.

[205] H. E. Sturgis. *A Postmortem for a Time Sharing System.* Technical Report CSL 74–1, Xerox Palo Alto Research Center, Palo Alto, CA, January 1974.

[206] Daniel Tabak. *Reduced Instructions Set Computer —RISC— Architecture.* Research Studies Press, Ltd., Letchworth, Hertfordshire, England, 1987. Distributed by John Wiley & Sons, Inc., New York, NY, USA.

[207] John D. Tangney. *Minicomputer Architectures for Effective Security Kernel Implementations.* Technical Report MTR-3531, The MITRE Corp., Bedford, MA, USA, May 1978.

[208] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. *Computer Architecture News*, 15(5):164–172, October 1987.

[209] Shreekant S. Thakkar and Alan E. Knowles. A high-performance memory management scheme. *Computer*, 19(5):8–22, May 1986.

[210] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[211] Ken Thompson and Dennis M. Ritchie. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[212] *ULTRIX-32 Programmer's Manual: Sections 1 and 6.* Order No. AA-BG53C-TE, Digital Equipment Corporation, Merrimack, NH, USA, 1986.

[213] *ULTRIX-32 Programmer's Manual: Sections 2 and 3.* Order No. AA-BG54C-TE, Digital Equipment Corporation, Merrimack, NH, USA, 1986.

[214] *VAX DBMS Database Security Guide.* Order No. AA–Y312A-TE, Digital Equipment Corporation, Maynard, MA, USA, January 1984.

[215] *VAX DEC/MMS User's Guide.* Order No. AA–P119B–TE, Digital Equipment Corporation, Maynard, MA, USA, August 1984.

[216] *VAX Hardware Handbook.* Volume 1, EB 25949 46/85 12 04/43, Digital Equipment Corporation, West Concord, MA, USA, 1986.

[217] *VAX–11/730 Central Processing Unit Technical Description.* EK–KA730–TD–001, Digital Equipment Corporation, Maynard, MA, USA, May 1982.

[218] *VAX–11/730 Memory System.* EK–MS730–TD–001, Digital Equipment Corporation, Maynard, MA, USA, May 1982.

[219] *VAX 11/780 Data Path Description.* AA–H307A–TE, Digital Equipment Corporation, Maynard, MA, USA, February 1979.

[220] *VAX/VMS Command Definition Utility Reference Manual.* Order No. AA–Z408A–TE, Digital Equipment Corporation, Maynard, MA, USA, September 1984.

[221] *VAX/VMS System Services Reference Manual.* Order No. AA-Z501B-TE, AD-Z501B-T1, Digital Equipment Corporation, Maynard, MA, USA, April 1986.

[222] D. E. Waldecker, C. G. Wright, M. S. Schmookler, T. G. Whiteside, R. D. Groves, C. P. Freeman, and A. Torres. ROMP/MMU implementation. In *IBM RT Personal Computer Technology*, pages 57–65, SA23–1057, IBM Engineering Systems Products, Milford, CT, USA, 1986.

[223] R. D. H. Walker. *The Structure of a Well–Protected Computer.* Ph. D. dissertation, Computer Laboratory, University of Cambridge, Cambridge, England, December 1973.

[224] David W. Wall. Global register allocation at link time. *SIGPLAN Notices*, 21(7):264–275, July 1986. Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, CA, 25–27 June 1986.

[225] K. G. Walter, W. F. Ogden, W. C. Rounds, F. T. Bradshaw, S. R. Ames, and D. G. Shumway. *Primitive Models for Computer Security.* Technical Report ESD–TR–74–117, Case Western Reserve University, Cleveland, OH, HQ Electronic Systems Division, Hanscom AFB, MA, USA, 23 January 1974.

[226] Desmond John Watson. *An Approach to Protection Through Capabilities.* Ph. D. dissertation, Computer Laboratory, University of Cambridge, Cambridge, England, July 1978.

[227] Clark Weissman. Security controls in the ADEPT–50 time sharing system. In *AFIPS Conference Proceedings, Volume 35, 1969 Fall Joint Computer Conference*, pages 119–133, AFIPS Press, Montvale, NJ, USA, 1969.

[228] *WFL Reference Manual.* Form No. 5011794, Burroughs Corporation, Detroit, MI, USA, March 1981.

[229] J. Whitmore, A. Bensoussan, P. Green, D. Hunt, A. Kobziar, and J. Stern. *Design for Multics Security Enhancements.* Technical Report ESD–TR–74-176, Honeywell Information Systems, Inc., HQ Electronic Systems Division, Hanscom AFB, MA, USA, December 1973.

[230] Maurice V. Wilkes. Unpublished lectures on reduced instruction set computer (RISC) design. Computer Laboratory, University of Cambridge, Cambridge, England., 1985–1987.

[231] Maurice V. Wilkes and Roger M. Needham. *The Cambridge CAP Computer and Its Operating System.* Elsevier North Holland, Inc., New York, NY, USA, 1979.

[232] A. L. Wilkinson, D. H. Anderson, D. P. Chang, Lee Hock Hin, A. J. Mayo, I. T. Viney, R. Williams, and W. Wright. A penetration analysis of a Burroughs large system. *Operating Systems Review*, 15(1):14–25, January 1981.

[233] J. H. Wimbrow. A large-scale interactive administrative system. *IBM Systems Journal*, 10(4):260–282, 1971.

[234] Simon Wiseman. A secure capability computer system. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 86–94, IEEE Computer Society, Oakland, CA, USA, 7–9 April 1986.

[235] Ian H. Witten. Computer (in)security: infiltrating open systems. *Abacus*, 4(4):6–25, Summer 1987.

[236] J. P. L. Woodward and G. H. Nibaldi. *A Kernel-Based Secure UNIX Design.* Technical Report MTR-3499, The MITRE Corporation, Bedford, MA, USA, November 1977.

[237] J. B. Wordsworth. Formal methods in the development of CICS. *Computer Bulletin*, 3(4):6–7, December 1987.

[238] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill, New York, NY, USA, 1981.

Appendix A

Computer Security Evaluation Criteria

This appendix is a brief summary of the computer security evaluation criteria, promulgated by the National Computer Security Center [59]. The criteria divide computer security systems into four major divisions, with classes within those divisions. Computer vendors submit their operating systems to the National Computer Security Center for design assistance and ultimately evaluation against the criteria. A number of commercially available systems have been successfully evaluated against the criteria. At least one commercial system has been evaluated in each of the four major divisions.

- **Division D: Minimal Protection**
  This division contains only one class. It is reserved for those systems that have been evaluated but that fail to meet the requirements for a higher evaluation class.

- **Division C: Discretionary Protection**
  Classes in this division provide for discretionary (need-to-know) protection.

  - **Class (C1): Discretionary Security Protection**
    Class (C1) systems provide a minimal set of security features to separate users and their data. Most conventional time-sharing systems fall into this class.
  - **Class (C2): Controlled Access Protection**
    Class (C2) systems require a finer grained control system than class (C1) systems. For example, simple owner/group/world protection schemes would be unacceptable at class (C2). Class (C2) systems must also provide improved audit trails and login control procedures.

- **Division B: Mandatory Protection**
  Classes in this division provide an implementation of the non-discretionary lattice security model.

- **Class (B1): Labeled Security Protection**
  Class (B1) systems must label all storage objects and enforce the lattice security model on those objects. However, covert channels are not addressed in this class.
- **Class (B2): Structured Protection**
  Class (B2) systems must label all system resources (as opposed to only storage objects), and must show that covert channels have either been eliminated or bandwidth limited. Also, a trusted communications path between the user and the system must provide two-way authentication.
- **Class (B3): Security Domains**
  Class (B3) systems are required to isolate the security functions from the rest of the operating system, typically into some form of security kernel. At this class, access control lists are explicitly required. An informal descriptive top level specification (DTLS) of the design is required.

- **Division A: Verified Protection**
  Division A systems are characterized by the use of formal mathematical methods to assure correctness of design and implementation.

  - **Class (A1): Verified Design**
    Class (A1) systems require the preparation and verification of a mathematically formal top level specification (FTLS) of the security kernel design. Informal techniques must be used to show correspondence between the FTLS and the implemented software.
  - **Beyond Class (A1)**
    Classes beyond A1 will probably require formal verification of the code of the security kernel and some considerations of microcode and hardware correctness. However, constructing systems at this level of security is still beyond current technology, so requirements have not yet been stated.

Appendix B

Tutorial on Paging

In the earliest processors with virtual memory, such as the Atlas [126], page tables could simply be stored in consecutive words of physical memory. Starting with the first Multics processor, the GE-645 [168], the size of the virtual address spaces have been sufficiently large that the entire page table cannot be stored in primary memory at once. The traditional solution to the problem of large page tables has been to construct hierarchies of tables, such as in the GE-645 or the VAX architecture. This appendix contains a brief review of the evolution of paging structures, as a tutorial backup to Chapter 16.

B.1 Atlas

The structure of the Atlas page table was very simple [126]. It mapped a virtual address space of 2^{18} 48-bit words consisting of 512-word pages. Thus, a page table would need at most 512 entries, as shown in Figure B.1.

Page Table Entry	0
Page Table Entry	1
...	
Page Table Entry	510
Page Table Entry	511

Figure B.1: Atlas Page Table

Atlas did not automatically consult the full page table. Instead, it had 32 Page Address Registers (PARs), one for each page of physical memory. On each memory reference it compared the virtual address against the 32 PARs and generated a fault, if no match was found. Software in the Atlas operating system was responsible for loading the PARs from the page table. Thus, the PARs formed a fully-associative translation buffer, with a software TB-miss handler. See Section 15.3 for a more complete discussion of translation-buffer miss handling.

B.2 Multics

By comparison, the Multics virtual address space was 2^{36} 36-bit words in 1,024-word pages. As a result, mapping a full Multics address space required 2^{26} page table entries, five orders of magnitude more than Atlas. Even with today's rapidly declining memory prices, one could not dedicate that much memory to page tables. Instead, the Multics processor provided a scheme for multiple levels of address-translation tables, so that only a small number of page table entries need be in primary memory at any one time. Figure B.2 shows how address translation was accomplished on the Multics processor.

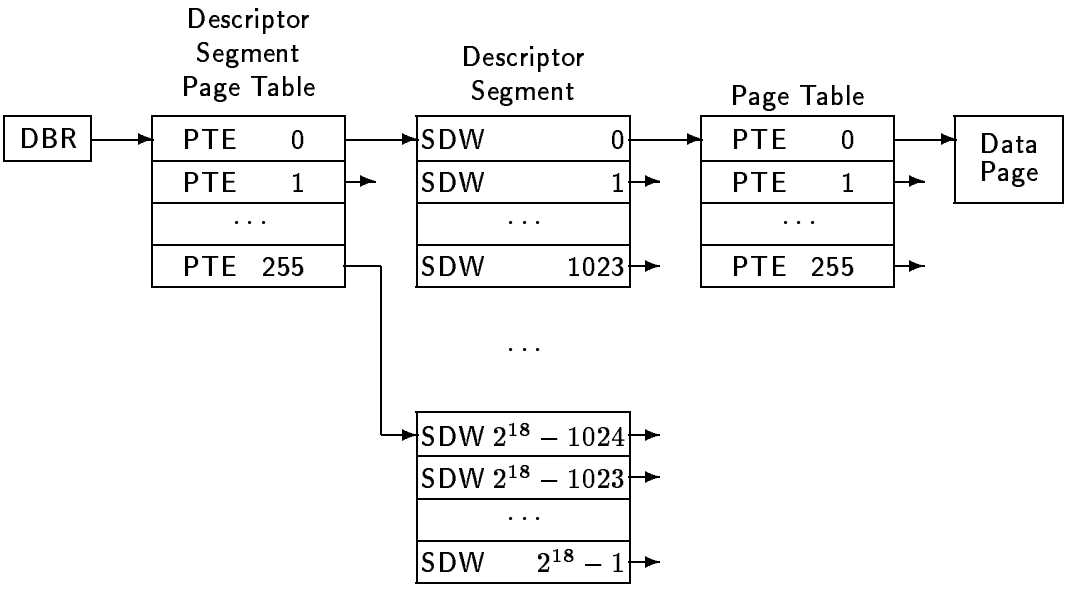

Figure B.2: Multics Address Translation

To translate a virtual address, Multics started with the descriptor base register (DBR) to locate the page table of the descriptor segment. From there, the processor stepped through the descriptor segment, and the page table of the target segment to find the physical address of the desired page of data. At each stage, the processor checked for exception conditions. As a result, the hardware and microcode needed to translate an address were quite complex.

B.3 VAX

The VAX architecture [138] supports a virtual address space of 2^{32} bytes in 512-byte pages. Like Multics, mapping a full VAX virtual address space requires a very large number of page table entries, 2^{23}. The VAX strategy for dealing with the proliferation of page table entries is different from Multics' strategy. To

avoid so many levels of tables, the address space is divided into regions, called system space, P0 space, and P1 space. The P0 and P1 spaces are collectively termed process space. Each of the three spaces may grow to 2^{30} bytes each. A fourth space is reserved for future use. The system-space page table resides in contiguous physical memory. The P0 and P1 page tables reside in system space, and are thus paged[1]. Figure B.3 shows how the page tables map the three regions of the VAX address space. In practice, only a fraction of system space is ever used so as to limit the amount of contiguous physical memory that would have to be devoted to page tables.

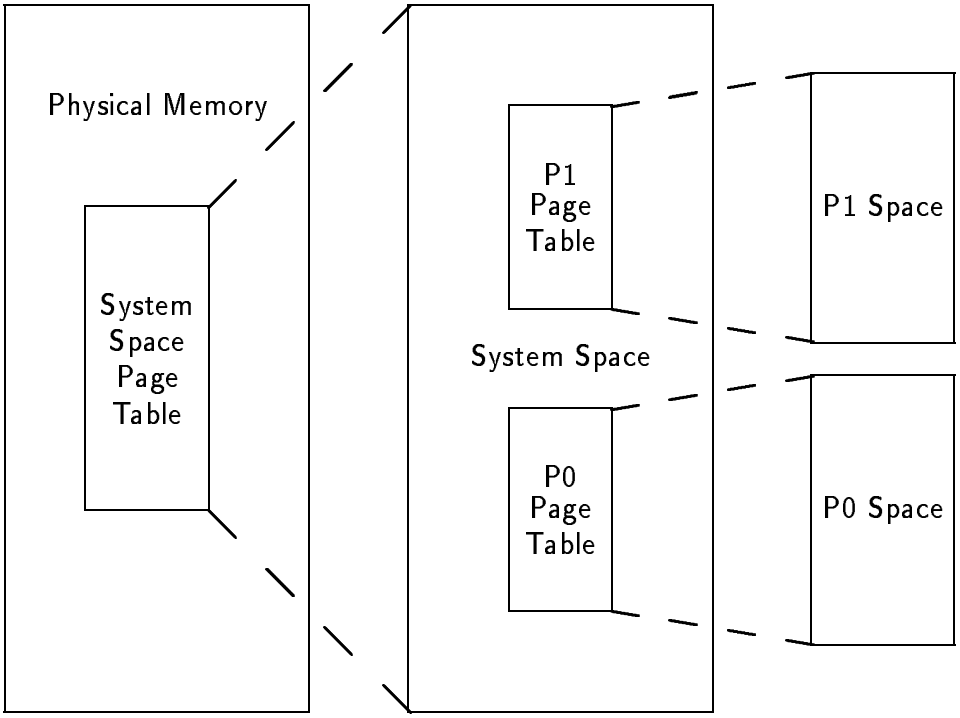

Figure B.3: VAX Address Space Mapping

B.4 Additional Levels of Page Tables

To support larger address spaces than found in either Multics or the VAX, one could add additional levels of page tables. The CLICS (Classroom Information and Computing Service) [39] design proposed an extra level of page tables,

[1]This description omits the complications related to P1 space growing in the negative direction.

beyond that of Multics, to support an address space of 2^{40} words. All such schemes for multiple levels of page tables suffer from two problems. First, it takes additional memory references to perform the address translation. The use of translation buffers can minimise the cost of extra memory references by resolving most addresses in the translation buffer, and only going through the page tables when the address is missing from the buffer. Second and more important, the multiple levels of page tables make the address translation hardware and microcode significantly more complex. At any level of translation, one or more of several exception conditions may be encountered. (These exceptions include translation not valid, access violation, and non-existent physical memory.) The address-translation mechanism must be prepared to deal with all of these. Further, the operating system must allocate space for the various page tables and must make policy decisions on how many page tables should be allowed in primary memory at any one time. Implementing those decisions can require quite complex software algorithms.

Appendix C

Translation Buffer Associativity

One of the most significant aspects of translation buffer design is the tradeoff of size against level of associativity. Most translation buffers are organised either as direct mapped, n-way set associative, or fully associative.[1] For the examples in this section, we shall assume that a virtual address is divided into a virtual-page-frame number and an offset within that page, as shown in Figure C.1.

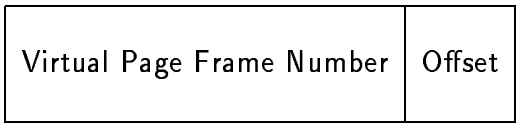

Figure C.1: Fields of a Virtual Address

C.1 Fully Associative

A fully associative translation buffer, as shown in Figure C.2, allows any entry in the TB to map any page in primary memory. The look-up hardware must compare the virtual page frame number with the tags of every entry in the translation buffer. To make the look-up fast, the comparisons must be done in parallel, and there must be logic to implement a replacement algorithm. Most frequently, a least-recently-used (LRU) algorithm is implemented with a usage count associated with each entry, although some machines use other replacement algorithms. Since both the parallel-look-up and the LRU replacement algorithms significantly increase in cost and complexity as the number of entries goes up, fully associative translation buffers are in practice quite small (typically 8 to 32 entries). For example, the MicroVAX 78032 chip [61] has an 8-entry, fully associative translation buffer with LRU replacement.

[1] Other translation-buffer organizations are sometimes used, but they do not directly affect the security questions of this chapter.

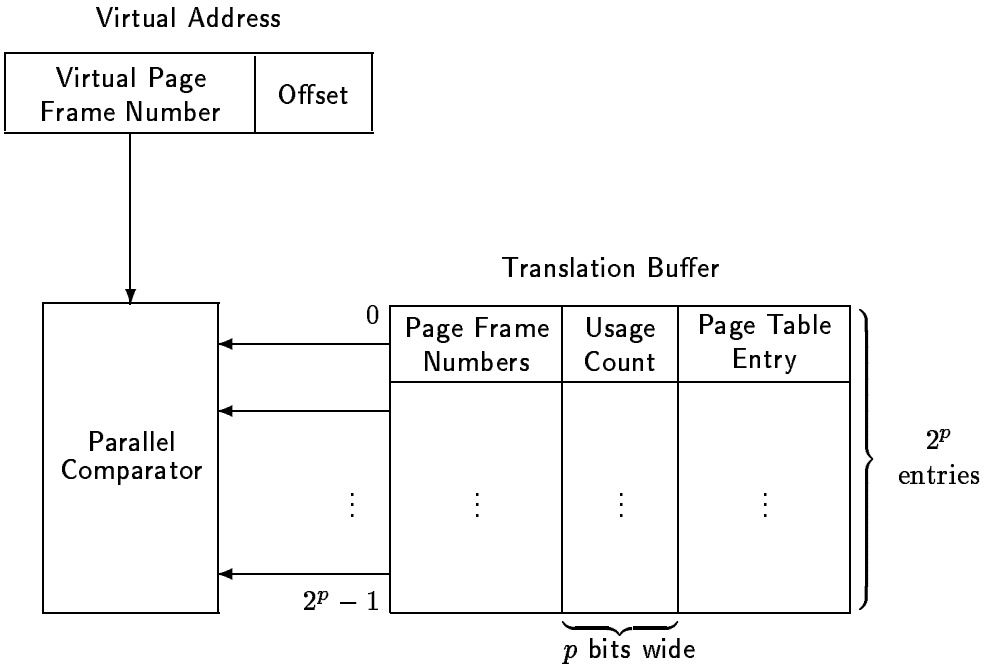

Figure C.2: Fully-Associative Translation Buffer

C.2 Direct Mapped

In a direct-mapped translation buffer, as shown in Figure C.3, the page descriptor for a particular virtual address can only be loaded into a corresponding translation-buffer entry. If the direct-mapped translation buffer contains 2^p entries, then p bits of the virtual page-frame number are used as an index into the buffer. The remaining bits of the virtual page-frame number are stored in the tag field of the TB entry and a single comparator checks whether the selected entry is a match. Thus, look-up and replacement hardware can be extremely simple and fast, and the incremental cost of additional entries is quite small. The low-end VAX-11/730 [218] has a 128-entry, direct-mapped translation buffer, and the high-performance VAX 8600 [71] has a 512-entry, direct-mapped translation buffer.[2]

Because more than one virtual address may map to a single TB entry, the CPU may have a very difficult time using a direct-mapped TB. In particular, if an instruction and its memory operand both map to the same TB entry, then the CPU cannot reference both simultaneously. Thus, the simplest TB-miss

[2]Most VAX translation-buffer implementations use the high order bit as part of the index, effectively dividing the translation buffer into two pieces, one for system space addresses and one for process space addresses.

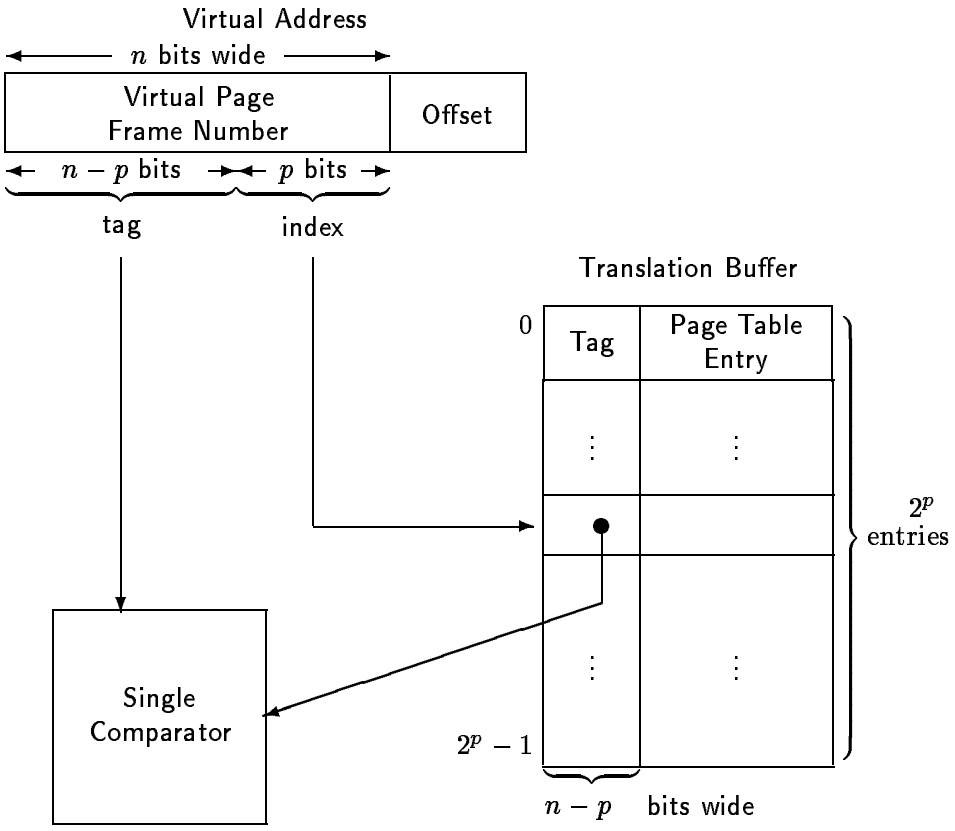

Figure C.3: Direct-Mapped Translation Buffer

strategy, restarting the instruction from scratch after filling the TB, will cause an infinite loop with a direct-mapped TB. Instead, the CPU (or its microcode) must maintain a copy of the instruction, to avoid re-fetching from memory after the operand TB miss has been resolved. If the instruction is allowed to reference many operands in different pages, then the CPU must be capable of buffering either all the operands or the intermediate results.

In a RISC machine, each instruction will reference at most a single memory operand.[3] However, some RISC architectures, such as the MIPS Computer Systems chip [55], implement the TB miss entirely in software. (See Section 15.3 for more detail.) If the return from the fault handler simply restarts the instruction fetch, then a direct-mapped translation buffer cannot be used.[4] Alternatively, the CPU could push the actual faulting instruction, rather than just the value of the program counter onto the exception handler's stack frame.[5] When the soft-

[3]The presence of vector instructions could complicate this issue.

[4]The MIPS chip in fact has a 64-entry, fully-associative translation buffer.

[5]The GE-600 and Honeywell H6000 series processors store the actual faulting instruction in their saved machine conditions.

ware handler returned, the CPU could use the stored instruction, rather than retrying the instruction fetch.

C.3 Set Associative

An N-way set-associative translation buffer represents a compromise between the direct-mapped and the fully-associative translation buffers. Essentially, an N-way set-associative TB consists of N direct-mapped TBs in parallel. Just as in the direct-mapped case, a set of bits from the address are used as an index into the TB. However, fewer bits are used for the look-up and the balance of the address bits are used in an associative comparison on the N entries selected by the index.

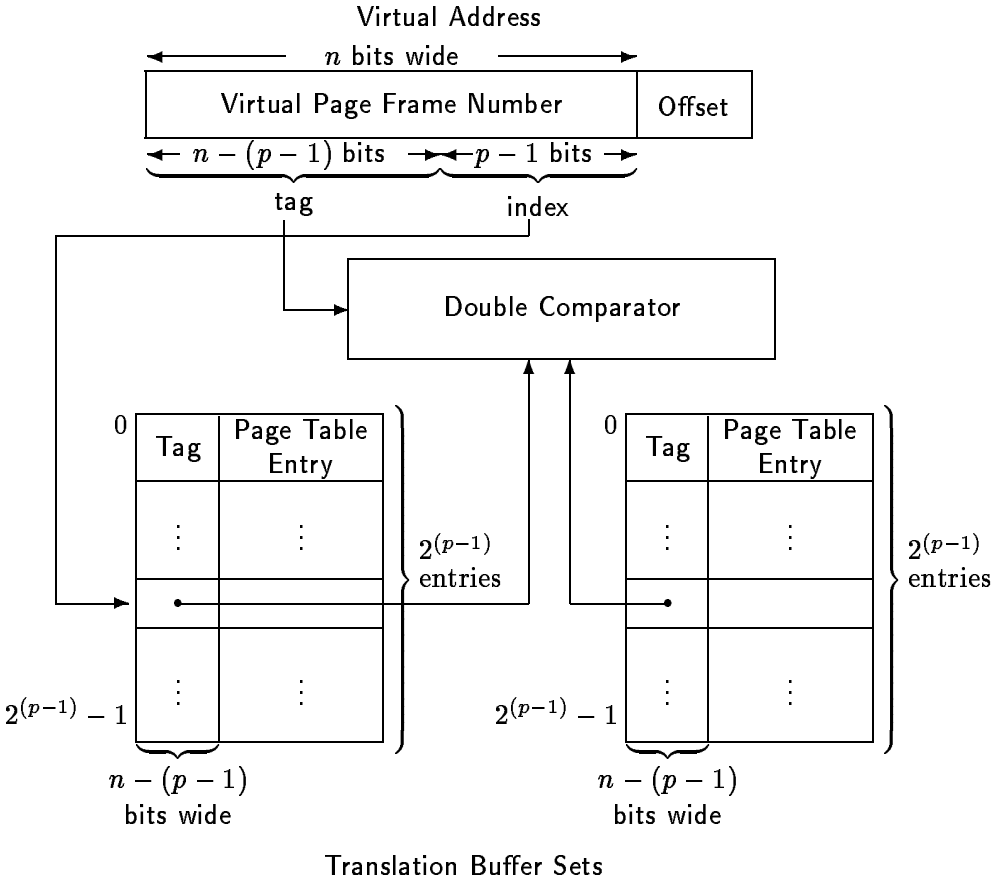

Figure C.4: Two-Way Set-Associative Translation Buffer

A 2-way set-associative translation buffer, as shown in Figure C.4, of 2^p entries would use $p-1$ bits from the virtual page-frame number to index the translation buffer, and then do a parallel comparison of the two entries, looking for a match

on the remaining bits of the virtual page-frame number. For example, the VAX-11/780 and the VAX-11/785 [216] have 128-entry and 512-entry, 2-way, set-associative translation buffers, respectively.

The set-associative organization reduces the frequency of address clashes, such as we saw in the direct-mapped translation buffer, but without the hardware cost and complexity of a fully-associative translation buffer. For a RISC processor in which the only memory referencing instructions are load and store of a single location, a 2-way set-associative translation buffer can completely avoid the infinite-loop problem of software-filling a direct-mapped translation buffer.[6]

[6]Operands or data structures that cross page boundaries can complicate this issue.

Appendix D

VAX Processor Architecture

This appendix briefly summarizes the VAX processor architecture. For the definitive specification of the architecture, see Leonard [138]. For a tutorial on assembly level programming of the VAX, see Levy and Eckhouse [141].

D.1 Data Types

The VAX processor supports a large number of data types, including integers, floating point numbers, variable-length bit fields, queues, character strings, and decimal strings. The machine is fundamentally byte-addressed, with each byte containing 8 bits, numbered from 0 to 7, right to left. Table D.1 summarizes the various data types.

D.2 Registers

The VAX architecture supports both programmer-visible registers and internal processor registers (IPRs) that are visible only to the operating system.

D.2.1 Programmer Visible Registers

The programmer of a VAX computer sees 16 general-purpose registers (GPRs), and a processor status longword (PSL). The GPRs are each 32 bits in length, and adjacent registers can be combined to represent quadwords, octawords, or D, G, or H floating point numbers. In addition, variable length (0 to 32-bit) bit fields may cross register boundaries. The following GPRs have special meanings:

- R15 is the program counter (PC). The PC contains the address of the next instruction byte of the program.
- R14 is the stack pointer (SP). The SP contains the address of the top of the current stack.

Type	Description
Integers (both signed and unsigned)	
Bytes	8 bits long
Words	16 bits long (two bytes)
Longwords	32 bits long (four bytes)
Quadwords	64 bits long (eight bytes)
Octawords	128 bits long (sixteen bytes)
Floating Point	
F_floating	32 bits long (8-bit exponent, 24-bit fraction)
D_floating	64 bits long (8-bit exponent, 56-bit fraction)
G_floating	64 bits long (11-bit exponent, 53-bit fraction)
H_floating	128 bits long (15-bit exponent, 113-bit fraction)
Queues	
absolute queues	circular, doubly linked lists using absolute virtual addresses as links
self-relative queues	use displacements from queue entries as links. can be interlocked for simultaneous access from multiple CPUs or I/O controllers.
Decimal Strings	
Trailing numeric	0 to 31 decimal digits as bytes, with the sign at the end of the string.
Leading separate numeric	0 to 31 decimal digits, two to a byte, with the sign at the beginning of the string.
Packed decimal	0 to 31 decimal digits, two to a byte, with the sign at the end of the string.
Other	
Character strings	contiguous sequences of from 0 to 65,535 bytes.
Variable-length bit fields	allow addressing 0 to 32 contiguous bits starting at any arbitrary bit position.

Table D.1: VAX Data Types

- R13 is the current frame pointer (FP). The FP contains the address of the base of the current stack frame, as used by the VAX procedure-calling convention.

- R12 is the argument pointer (AP). The AP is used by the subroutine call and return instructions (CALLx and RET) to point to argument lists.

The PSL is a 32-bit longword that defines the current processor status. Table D.2 briefly summarizes the contents of the PSL. See the *VAX Architecture Reference Manual* [138, pp. 20-23] for a more detailed description of the PSL.

Extent	Name	Mnemonic	Meaning
<31>	Compatibility Mode	CM	Indicates the CPU is in PDP–11 compatibility mode.
<30>	Trace Pending	TP	Force a trace fault at the start of the next instruction.
<29:28>	Reserved		Reserved to Digital. Must be 0.
<27>	First Part Done	FPD	First part of an interruptable instruction done.
<26>	Interrupt Stack	IS	Processor is executing on the interrupt stack.
<25:24>	Current Access Mode	CUR_MOD	Access mode of the current process.
<23:22>	Previous Access Mode	PRV_MOD	Access mode before the last exception or CHMx instruction.
<21>	Reserved		Reserved to Digital. Must be 0.
<20:16>	Interrupt Priority Level	IPL	The current processor priority.
<15:8>	Reserved		Reserved to Digital. Must be 0.
<7>	Decimal Overflow Enable	DV	Enable decimal overflow traps.
<6>	Floating Underflow Enable	FU	Enable floating underflow exceptions.
<5>	Integer Overflow Enable	IV	Enable integer overflow traps.
<4>	Trace Enable	T	When set at the beginning of an instruction, cause TP to be set.
<3:0>	Condition Codes	NZVC	Negative, Zero, Overflow, and Carry condition code bits.

Table D.2: Processor Status Longword Fields

D.2.2 Internal-Processor Registers

The VAX architecture also supports a large number of internal processor registers (IPRs) that are referenced with the move to/from processor register instructions (MTPR and MFPR). Both of those instructions are privileged, so that only the operating system can reference the IPRs. Table D.3 briefly summarizes the architecturally defined IPRs. In addition to those IPRs, each VAX CPU may define additional IPRs that are specific to that processor alone.

Name	Mnemonic	Hex
kernel stack pointer	KSP	0
executive stack pointer	ESP	1
supervisor stack pointer	SSP	2
user stack pointer	USP	3
interrupt stack pointer	ISP	4
P0 base register	P0BR	8
P0 length register	P0LR	9
P1 base register	P1BR	A
P1 length register	P1LR	B
system base register	SBR	C
system length register	SLR	D
process control block base	PCBB	10
system control block base	SCBB	11
interrupt priority level	IPL	12
AST level	ASTLVL	13
software interrupt request register	SIRR	14
software interrupt summary register	SISR	15
interval clock control	ICCS	18
next interval count register	NICR	19
interval count register	ICR	1A
time-of-year register	TODR	1B
console receiver status	RXCS	20
console receiver data buffer	RXDB	21
console transmit status	TXCS	22
console transmit data buffer	TXDB	23
memory management enable	MAPEN	38
translation buffer invalidate all	TBIA	39
translation buffer invalidate single	TBIS	3A
performance monitor enable	PME	3D
system identification	SID	3E
translation buffer check	TBCHK	3F

Table D.3: Architecturally Defined Internal Processor Registers (IPRs)

D.3 Instruction Formats

Machine instructions in the VAX architecture are of varying lengths, aligned on byte boundaries. An instruction consists of an opcode followed by 0 to 6 operand specifiers. Opcodes may be either one or two bytes long, and operand specifiers are between one and eighteen bytes long. There are more than 300 native mode instructions and dozens of addressing modes. (The exact numbers depend on how one counts.) The instruction set is highly orthogonal; that is, the instruction set allows operations, data types, and addressing modes (to a great extent) to be chosen independently.

D.4 Memory Management

The VAX address space is 2^{32} bytes in length, divided into four regions of 2^{30} bytes each. The lower-addressed half of the address space contains two regions, called P0 space and P1 space, are unique to each process. P0 space is for program text and static data. P1 space is for the program stack and grows in the negative direction. P0 and P1 spaces are collectively termed process space.

The upper-addressed half of the address space also contains two regions—system space (S0 space) and an unused region, sometimes called S1 space. Although the process space regions are changed whenever a process is scheduled, the system space region normally remains the same in all processes. System space is normally used only by the operating system.

The actual layout of the page tables is described in Section B.3.

D.5 Protection

The VAX architecture implements protection by providing four concentric rings of protection, called access modes. The four access modes are numbered from 0 to 3 and are named kernel, executive, supervisor, and user, respectively. Access mode protection is implemented by comparing the CUR_MOD field of the PSL with the four-bit protection field of the page-table entry (PTE) that maps the virtual address in question. Table D.4 shows the possible values the PTE protection field can take. The VAX uses a more efficient encoding of the possible value of read and write protection combined with access modes than other architectures, such as the Honeywell H6180 Multics processor [192].

The VAX processor allows software to change access modes by executing one of the four change mode instructions, (CHMK, CHME, CHMS, and CHMU). These instructions cause the processor to branch to a known address in the target access mode. Code in the new mode can then determine what operation has actually been requested. When a CHMx instruction is executed, the access

Name	Mnemonic	Value	Accessibility			
			Kernel	Exec	Super	User
no access	NA	0	none	none	none	none
reserved		1	UNPREDICTABLE			
kernel write	KW	2	write	none	none	none
kernel read	KR	3	read	none	none	none
user write	UW	4	write	write	write	write
exec write	EW	5	write	write	none	none
exec read, kernel write	ERKW	6	write	read	none	none
exec read	ER	7	read	read	none	none
super write	SW	8	write	write	write	none
super read, exec write	SREW	9	write	write	read	none
super read, kernel write	SRKW	10	write	read	read	none
super read	SR	11	read	read	read	none
user read, super write	URSW	12	write	write	write	read
user read, exec write	UREW	13	write	write	read	read
user read, kernel write	URKW	14	write	read	read	read
user read	UR	15	read	read	read	read

Table D.4: PTE Protection Codes

mode of the caller is stored in the PRV_MOD field of the PSL, and the CUR_MOD field is set to the new access mode.

The VAX accomplishes argument validation with special PROBE instructions and the PRV_MOD field. The probe instructions test the access rights to an argument of not only the current access mode, but also the previous access mode to prevent any kind of Trojan horse pointer attack.

D.6 Interrupts and Exceptions

When an interrupt or exception occurs, the VAX processor transfers to an address specified in the System Control Block (SCB). The SCB consists of an array of vectors that contain addresses of the exception handlers. Each interrupt or exception has its own SCB vector.

Most interrupts and exceptions cause the processor to switch into kernel mode, although certain arithmetic exceptions are handled in the access mode in which they occurred. The CHMx exceptions cause the processor to switch into the access mode specified by the CHMx instruction that was executed. The SCB vector also contains a flag to control whether the exception is handled on the kernel stack of the current process or on the per-CPU interrupt stack.

The architecture defines thirty-two interrupt priority levels (IPLs) to mediate between conflicting I/O devices. Interrupt levels 16 to 31 are reserved for

hardware devices, while interrupt levels 0 to 15 are reserved for software use. Software interrupts can be requested using the SISR and SIRR internal processor registers. When the processor is executing at a given IPL, interrupts at a lower or equal IPL are held pending until the processor lowers its priority level.

The VAX architecture also supports a mechanism for asynchronous system traps (ASTs). See Section 19.4 for more detail.

Appendix E

Microarchitecture of the VAX-11/730

This appendix briefly summarizes the microarchitecture of the VAX-11/730 processor that I have used for experiments in this dissertation. For more detailed information on the processor, consult the CPU Technical Description [217] and the Memory System Technical Description [218].

E.1 System Overview

The VAX-11/730 computer system was announced in 1982 as the third member of the VAX family. It was marketed at the time as a low-cost entry-level system, with performance of approximately one third that of a VAX-11/780. The primary design goal of the system was minimum cost, rather than high performance, and the influence of that goal can be seen in the hardware and microcode. Figure E.1 shows the major components of a VAX-11/730 system. The KA730 CPU consists of three major modules: the data-path module (DAP), the writable-control store module (WCS), and the memory-controller module (MCT). The three modules of the CPU are connected by the memory-control bus (MC bus). An optional FP730 Floating-Point Accelerator (FPA) can be attached to the CPU to improve the performance of floating-point instructions. Up to five megabytes of memory can be attached to the MCT.

The VAX-11/730 uses a UNIBUS as its peripheral I/O bus. An optional RB730 Integrated-Disk Controller (IDC) provides a low cost way to attach disks. Alternatively, conventional disk controllers can be attached directly to the UNIBUS.

The WCS module includes a console processor that controls the system's console terminal and a pair of TU58 DECtape II tape cartridge drives.

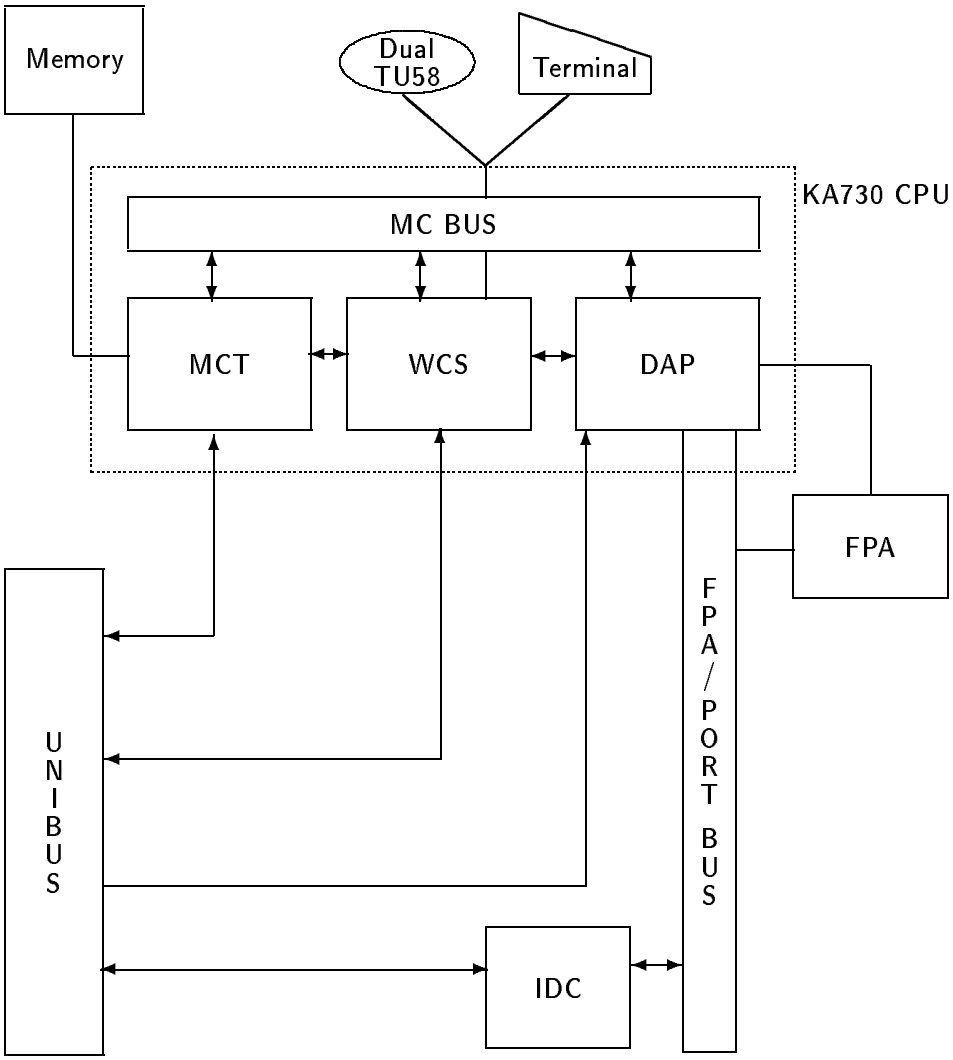

Figure E.1: VAX-11/730 System

E.2 CPU Description

The major components of the KA730 CPU are the memory controller (MCT), the the writable-control store (WCS) and the CPU data path (DAP). Figure E.2 is a simplified block diagram of these three modules.

E.2.1 Memory Controller (MCT)

The memory controller module (MCT) is a microprogrammed device to control one to five memory cards of one megabyte each. The MCT microinstructions are stored in 512-entry PROM consisting of 72-bit microwords. My research did not require any modifications to the MCT microcode.

The MCT contains the translation buffer (TB), used for virtual to physical-address translation and for UNIBUS-address mapping. The TB contains 1,024 entries of 23 bits each. Of those entries, 128 are used as a direct-mapped translation buffer. The 128 translation buffer entries are organized as 64 for system-space addresses and 64 for process-space addresses. (See Section C.2 for a discussion of direct-mapped translation buffers.) Of the remaining TB entries, 512 are used for UNIBUS mapping, and 384 are unused. While I considered attempting to use the 384 unused entries to store entries from more than one address space simultaneously, such a change would have required major modification to the PROM and possibly to the hardware of the module. Such modifications would have been quite difficult and would have invalidated the service contract on the system.

The MCT also contains a data rotator that is used for handling unaligned memory references and for certain shift and rotate operations in implementing VAX instructions. The data rotator is not a full barrel shifter and can only rotate data one byte right, one byte left, or two bytes right.

E.2.2 Writable-Control Store (WCS)

The writable-control store module (WCS) contains either 16K or 20K entries of 24-bit microwords that are used to implement the VAX instruction set. (The optional 4K is used to implement functions required by the Integrated-Disk Controller IDC.) The micromemory is read-only to the CPU data path, but the console processor, an Intel 8085A 8-bit microprocessor loads the microprograms from a TU58 tape at the time the system is powered-up. The console processor also implements the VAX console-command language and the interval timer and time-of-year clock functions for the system. The WCS module also includes the UNIBUS data transceivers for moving data to and from UNIBUS I/O devices.

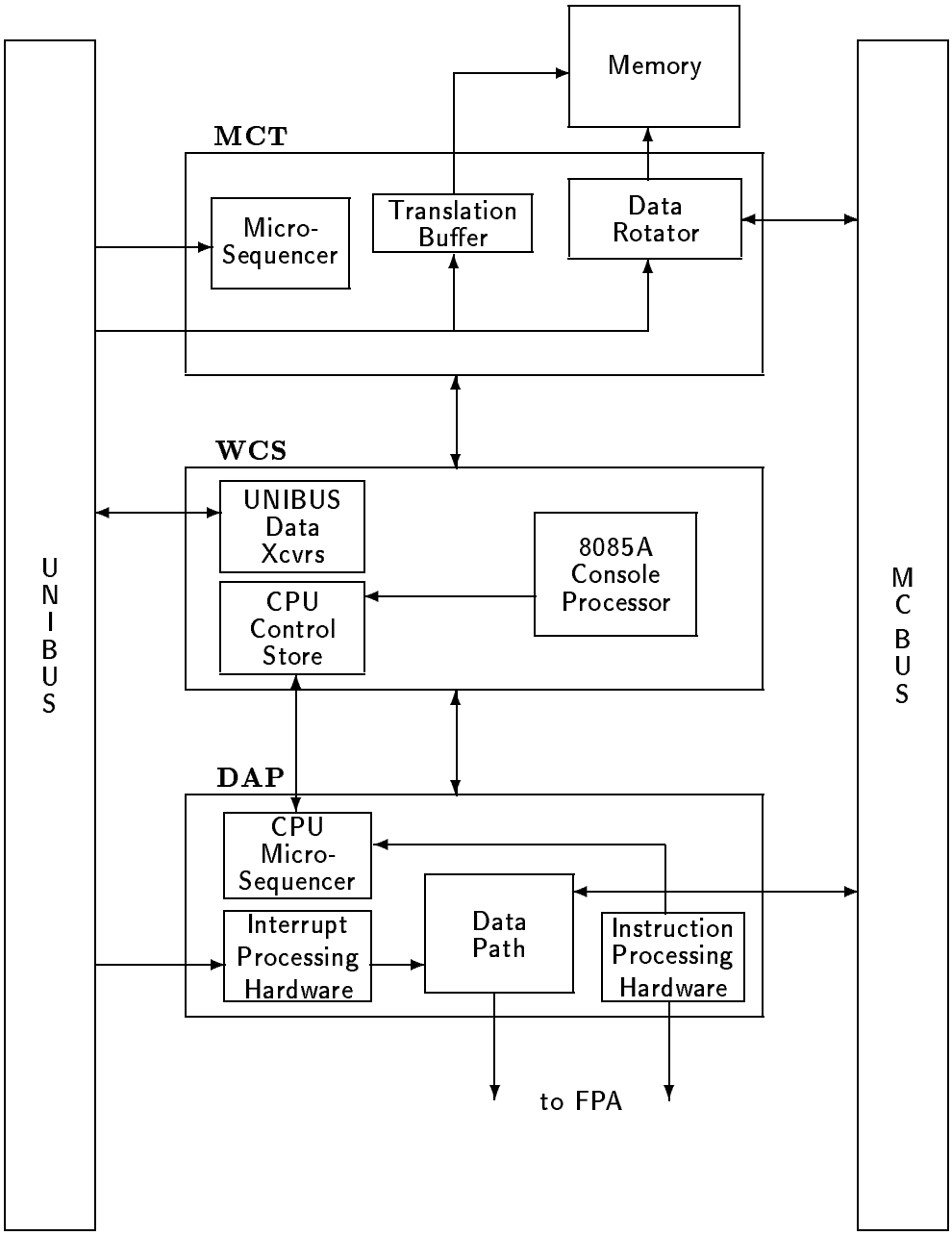

Figure E.2: KA730 Block Diagram

E.2.3 CPU Data Path (DAP)

The CPU data path module (DAP) consists of a microsequencer, interrupt-processing hardware, instruction-decoding hardware, and the data path itself. Figure E.3 contains a block diagram of the basic data path.

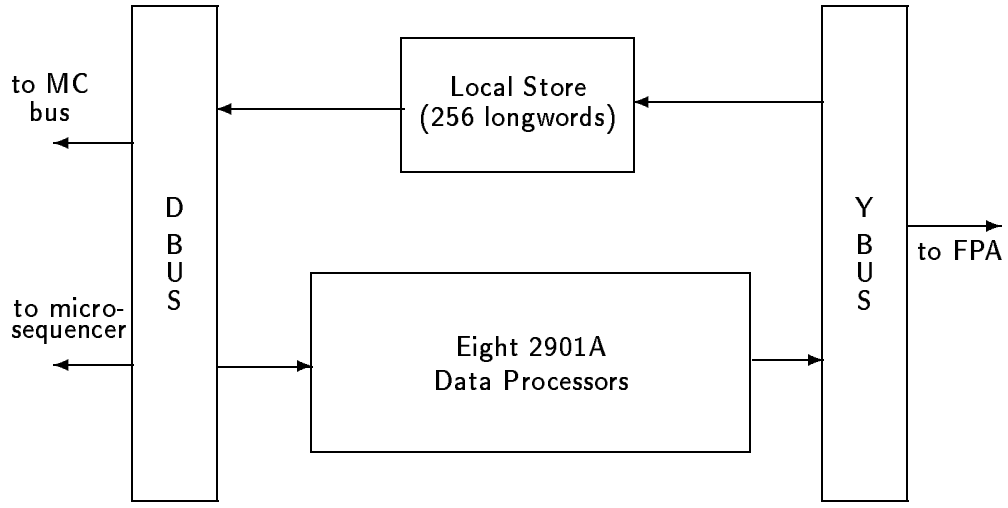

Figure E.3: Data Path Block Diagram

The data path is implemented by eight cascaded Am2901A 4-bit microprocessor slices, configured for carry look-ahead and external shift control. Data enters the 2901As from the D bus, and output data is transmitted on the Y bus. The 2901As contain sixteen 32-bit RAM locations and a special 32-bit Q register. The VAX-11/730 uses only six of the sixteen RAM locations as working registers in the microprogram.

The data path contains a 256-entry local store (LS), each entry consisting of 32 bits. Local store contains the VAX general-purpose registers (GPRs) and internal processor registers (IPRs), and various constants and temporaries used by the microprogram.

E.2.4 Micro Instruction Set

The VAX-11/730 microcode is organized vertically. Therefore, it is relatively easy to write microprograms. Each microinstruction resembles an assembler-level instruction of a simple minicomputer. There is relatively little parallelism, unlike in a horizontally-microprogrammed machine where each microinstruction would specify many functions in parallel. The principal difficulties in writing microcode for the VAX-11/730 come from the non-orthogonal microinstruction set, the delayed-branch instructions, and the delayed-memory-reference instructions.

The microinstruction set is not at all orthogonal. That is, not all operations can be performed on all sources and destinations. Some microinstructions can only be used with certain of the working registers and/or the Q register. However, the local store organization is the most serious cause of non-orthogonality.

Local store is organized in two halves of 128 entries each. The lower half of local store can be used as the source or destination address of a wide variety of microinstructions, permitting arithmetic to be done directly to or from a local store address. However, the upper half of local store can only be used in move (MOV) microinstructions. As a result, operations on values stored in the upper half of local store can require one or two extra micro instructions. Further, the microprogrammer can easily forget whether a named location is in the upper or lower half of local store, resulting in a large number of microassembly-time errors.

The microinstruction set has a very limited repertoire of shift and rotate instructions, as shown in Table E.1. The single and double bit shifts and rotates are implemented by the DAP, but the 9-bit and 15-bit rotates are implemented by the data rotator in the MCT and require delays, just as memory references do.

- Shifts
  - 1 bit right, sign extended
  - 1 bit left, zero filled
  - 2 bits left, zero filled
- Rotates
  - 1 bit right
  - 1 bit left
  - 9 bits right (using data rotator)
  - 15 bits left (using data rotator)

Table E.1: VAX-11/730 Microcode Shifts and Rotates

E.3 Microprogram Organization

The microprogram is organized into 12 major modules, as shown in Table E.2.[1] The two modules, FPWARM and FPHOT, are of the most interest, because they

[1]The instructions are not strictly divided between the modules, as Table E.2 would suggest. For example, the BITFLD module does not contain only bit field instructions.

are where the most unused microwords can be found. FPWARM contains the microcode implementation of all the VAX floating point instructions. FPHOT contains microcode to invoke the Floating-Point Accelerator (FPA) to implement a large number of the VAX floating-point instructions. At power-up time, the system checks to see if an FPA is present and loads either FPWARM or FPHOT from the TU58 tape cartridge. This is important, because FPHOT contains a large block of free microwords, and it was these microwords that I used to implement the SCAP microcode. Thus, SCAP microcode can only run on VAX-11/730s that have an FPA installed.

Module Name	Start Address	End Address	Function
DEFINE	-	-	macro definitions
CONSLE	0000	07FF	console functions, MFPR, MTPR
MMIE	0800	0DFF	memory mgmt, interrupts & exceptions
FPWARM	0E00	19FF	warm floating point
FPHOT	0E00	19FF	hot floating point
BITFLD	1A00	1CFF	bitfield instructions
CM	1D00	21FF	PDP–11 compatibility mode
BASIC	2200	3AFF	basic instruction set
QUEUE	3B00	3FFF	queue instructions
IDC	4000	43FF	integrated disk controller
IRDFPA	6200	6614	basic instruction set with FPA
POWER	8E00	8FFF	power-up code

Table E.2: VAX-11/730 Microprogram Modules

E.4 Microprogramming Tools

The microprogramming tools available for the VAX-11/730 are relatively primitive. There is no linker, so the microcode must be assembled as a single block by the MICRO2 assembler[155]. The output of MICRO2 is a listing file and a ULD file. A ULD file contains a text representation of the microcode to be loaded into the micromemory or U-memory. The ULD file is then processed by a program, called ULDTOBIN, that produces binary images suitable for writing to the TU58 tape cartridge. The microprogram is over 85,000 lines in length and a microassembly combined with the necessary housekeeping operations to prepare the TU58 tape and cross-reference listings can take over 2 1/2 hours on a VAX-11/730 or over one hour on a MicroVAX-II. Writing the TU58 tape takes another 10 minutes, and loading the microcode into the WCS takes another 10 minutes. Thus, correcting a minor microcode error can be very time-consuming.

Appendix F

Interrupt Handling in Capability Systems

This appendix summarizes how hardware interrupts were handled in a number of earlier capability-based processors. The information is presented here to contrast with the SCAP approach to interrupt handling, presented in chapter 19.

F.1 CAP

The CAP computer handles interrupts in microcode by forcing an entry to the master-coordinator process [223, Section 3.15]. The master coordinator then schedules the appropriate process to run in response to the interrupt. The expense of the Coordinator Entry instruction plus the cost of making a scheduling decision makes interrupt latency relatively long in the CAP. Actual device control was not carried out in the CAP processor itself, but rather in a front-end computer. Originally, a CTL Modular One computer served as the front-end processor to which all I/O devices were attached. As the Cambridge ring was developed, the CAP operating system was converted to use only a single peripheral, the ring, for all I/O operations [53]. The CAP microprogram was modified to include five orders for driving the ring which force coordinator entries as above. As with the Modular One, the actual terminal and disk control is done, not by the CAP processor, but by the terminal concentrators and file servers attached to the ring [165]. While the CAP operating system had no real-time response requirements in either configuration, the modified CAP microprogram had to respond to interrupts from the ring every twelve microseconds. Ring interrupts were handled entirely in the microprogram, and no attempt was made to execute cross-domain calls in response to interrupts.

F.2 Intel 432

The Intel 432 consists of the 432 General Data Processor (GDP) chip [101] and one or more front-end processors that handle I/O. The front-end processors are conventional processors and are not capability-based. The GDP communicates with the front-end processors using a message-passing protocol described in [169, Chapter 7]. The message-passing protocol, including the process-wakeup functions, is implemented entirely in microcode. While this architecture eliminates the need for interrupts in the capability-based processor, the actual I/O device drivers must be written in a totally conventional manner with none of the benefits of protection domains to assist in assuring correctness and security. Further, the cost of passing messages to the GDP means that any time-critical functions must be performed in the front-end processor, rather than in the capability-based environment.

F.3 IBM System/38

The IBM System/38 handles I/O interrupts with a complex microcode mechanism that turns the interrupts into signals in a general-purpose *event* system [103, Chapter 15]. Machine instructions are provided to allow programs to declare event monitors, test events, enable and disable events, etc. Events can be caused by I/O operations, timer runouts, or explicit program action. Because the System/38 event mechanism is so high-level, the amount of microcode needed to handle an I/O interrupt is quite large. However, the System/38 is not sold as a real-time processor, so the relatively long times needed to turn interrupts into events are probably not a serious concern.

F.4 Honeywell DPS 88

The Honeywell DPS 88 processor [64] responds to faults and interrupts by executing a cross-domain call instruction to a particular domain as specified by an enter capability stored in a processor-specific fault vector.[1] While the translation buffers of the DPS 88 are optimized to not require flushing on a cross-domain call, the number of registers that must be saved and loaded at the time of an interrupt is sizable (up to sixty four 36-bit words). As a result, the interrupt latency is likely to be quite high. The DPS 88 is sold as a mainframe computer and is not intended to do real-time processing. It has a separate channel processor, called an Input-Output Multiplexor (IOM), that directly handles the peripherals.

[1] In DPS 88 terminology [64], a cross-domain call is an ICLIMB instruction, and an enter capability is an entry descriptor.

F.5 Plessey System 250

The Plessey System 250 [87] has a different approach to handling I/O interrupts in a capability system. Rather than using a true interrupt system, the System 250 uses a polling system to detect I/O events. A location in store, the system interrupt word, is accessible to all processors and I/O channels through a special capability. I/O channels (or processors) can set bits in the system interrupt word. Each CPU polls the interrupt word periodically (typically every 100 μs).[2] Such a polling approach is particularly suitable for the intended applications of the System 250 in telephone-switching systems. The only conventional interrupts [65, pp. 401–402] in the System 250 are for timer runouts (to implement the polling) and for fault conditions.

[2] This type of scheduled device polling is common in many real-time applications.

Appendix G

Possible Kernel Design

This appendix contains a brief sketch of what the SCAP security kernel might look like. The research leading up to this dissertation has focused primarily on the security model and on the processor architecture. The purpose of this sketch is merely to suggest how the kernel might be designed. It does not include any detailed design or implementation, so major portions of the sketch may prove to be inaccurate.

The design is based heavily on the Multics security kernel [190], the Naval Postgraduate School security kernel [187], and the earlier Digital Equipment Corporation security kernel [120] designs. As part of the Multics security kernel design, Janson [108] proposed breaking the kernel into a hierarchic set of type managers. By forbidding dependency loops between the type managers, the overall design of the kernel becomes much clearer, and one can easily debug lower-level components of the kernel with higher-level components not present at all. All of the previous efforts to create layered kernel designs have used type managers only to structure their code. No enforcement of type-manager-boundaries was ever done. All of the layers of the kernel actually resided in a single domain of protection, because the processors in question, Multics, the Zilog Z8000, and the VAX, were all based on protection ring architectures and could not support large numbers of small domains. As seen in the author's experience during the development of the VAX kernel, malfunctions in one layer of the kernel can easily damage other layers. The major new feature of the SCAP kernel is the use of small protection domains to isolate the different layers of the kernel. This strong isolation should produce a more robust kernel implementation that should be easier both to debug and to formally specify and verify.

G.1 The Major Type Managers

Figure G.1 shows the lower-level type managers of the SCAP kernel. The design is based heavily on Janson's design for Multics, and this choice of layers is still very preliminary. When the full design of the SCAP kernel is done, I expect that

the particular set of layers will change somewhat. In particular, the design shown in Figure G.1 does not include disk quota or removable disk packs. These areas will likely be quite different from Janson's design. The figure also omits many of the higher levels of the kernel including network interfaces, terminal handlers, and the secure-server functions.

The translation-buffer manager is the lowest layer of the kernel. It is responsible for loading PTEs into the translation buffer. It uses the inform capability segments and the page tables to determine what to load. As part of its operation, it must check for capabilities that have not been validated and generate exceptions to higher layers. It must also check the eventcount values in the capability to detect revocations, based on the strategy described in Chapter 11. The translation-buffer manager may be implemented in software or in microcode, depending on how the CPU is implemented. (See Section 15.3.)

The kernel-segment manager is responsible for allocating the primary memory used by the security kernel itself. Since the cost of memory is dropping so dramatically, this layer can be made very simple by pre-allocating the kernel's memory at boot time. As a result, page faults occur only on non-kernel pages, and the overall memory-management strategy can be much simpler. The idea of a separate memory-management layer to support the kernel itself was first proposed by Saxena. [183]

The lower-level scheduler, based on the work of Reed [180], creates the abstraction of SCAP processes. SCAP processes are analogous to Reed's level-one virtual processors (vp1s), and are described in more detail in Section 6.1.2. The lower-level scheduler handles I/O interrupts and converts them into wakeups for the appropriate SCAP process. SCAP processes synchronize with each other by using level-one eventcounts [181]. All code above the lower-level scheduler runs in the context of a SCAP process.

The primary-memory page manager is responsible for allocating all pages of primary memory that are not used by the kernel-segment manager. The pages are kept on one of four lists - the used list, the free list, the modified list and the bad list. Higher layers are responsible for moving pages between the various lists.

The disk driver is responsible for reading and writing pages on the disks. This abstraction will actually be somewhat more complex, as there is a general I/O manager that handles all I/O requests and dispatches requests to various I/O drivers. Drivers will run in dedicated SCAP processes so that I/O can be performed asynchronously with the rest of the security kernel.

The page-table manager is responsible for page-table entries for each page of non-kernel objects. It handles page faults, issues the appropriate disk I/O requests, etc. The exact form of the page-table structures is yet to be determined.

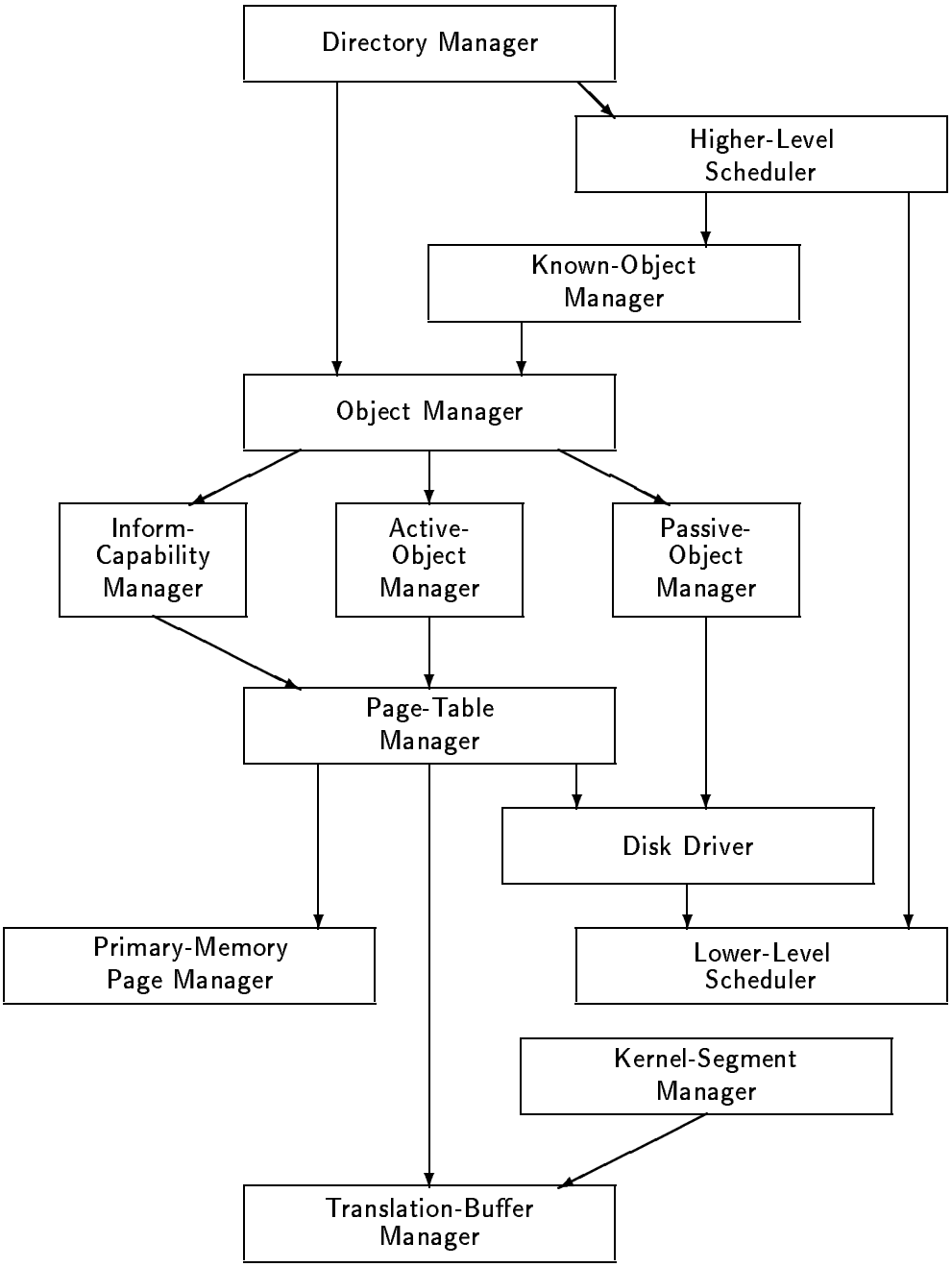

Figure G.1: Lower-Level Type Managers for SCAP Security Kernel

The inform-capability manager is responsible for the capability segments associated with a domain. It is the analog of Janson's [108] connected-segment manager that manages the Multics descriptor segments.

The passive-object manager maintains a flat file system on the disk for storing objects. It is called by the object manager to swap objects in and out.

The active-object manager is responsible for the active-object table (AOT), as described in Section 11.3.

The object manager is responsible for objects, whether they are active or passive. As a result, it calls on both the active- and passive-object managers to move objects in and out. In particular, the object manager is responsible for maintaining consistency of objects on disk and in primary memory. The final implementation of the SCAP security kernel will have to support an atomic transaction protocol on objects, to avoid the problems that Multics suffered that directories could be left inconsistent after a system crash. The security kernel cannot afford anything of the complexity and unreliability of the Multics salvager. Instead, the object manager and the directory manager will implement some form of two-phase commit protocol to ensure object consistency. Detailed design of such a reliable storage system is beyond the scope of this dissertation.

The known-object manager maintains a per-job known-object table (KOT). Each job has a known-object table that holds all the outform capabilities for all domain instances within the job. The inform capabilities have pointers back into the KOT for use when objects are deactivated. The KOT is somewhat analogous to the process resource list (PRL) in the CAP I system [231], but indirection to the PRL is only used to find an outform capability, and no further indirection to other KOTs is ever required.

The higher-level scheduler (HLS) creates the SCAP domain abstraction. As discussed in Section 6.5, SCAP domains are analogous to Reed's level two virtual processors (vp2s). Processes move from domain to domain by issuing cross-domain-call instructions that are logically the equivalent of Reed's bind operations. The HLS is responsible for creating and destroying domains and for gross level scheduling of the system, including, in particular, scheduling of access to primary memory. The lower-level scheduler, by contrast, tries to keep the CPU busy from the set of runnable processes. Together, the higher-level and lower-level schedulers implement the Multics notion of eligibility [168, p. 273], that only processes from as many domains as will fit into primary memory should be eligible to run on the CPU. Scheduling additional domains would only cause the demand paging system to thrash.

The directory manager is responsible for maintaining the directory hierarchy on disk. All objects have directory entries, so that access control decisions can be made. The directory manager and the object manager have to implement an atomic transaction protocol to ensure object consistency, even in the presence of system crashes.

G.2 Performance Considerations

The design presented so far has used protection domains quite freely. As a result, the number of cross-domain calls within the security kernel will be very high. As Janson points out, however, such a large number of cross-domain calls will likely make the overall system performance unacceptably bad, no matter how fast the actual cross domain call can be made. He recommends that calls on lower-level domains be in-line expanded by the compiler. Such a recommendation is quite reasonable in the context of a Multics security kernel in which all the type managers are actually running in a single domain of protection (ring 0). However, the primary purpose of SCAP domains is to provide structure and protection within the security kernel to provide additional confidence that the kernel is correctly implemented. In-line code expansion would eliminate those benefits.

This author's personal experience with security-kernel implementation in a protection-ring environment [120] indicates that only certain cross-domain calls within the kernel occur frequently. The SCAP kernel should initially be implemented with all type managers in separate domains. Then, the kernel's performance should be carefully measured, and based on those measurements, certain layers should be collapsed by in-line code expansion. As a result, it is essential that the kernel's implementation language support optional in-line expansion of procedure calls.

Appendix H

SCAP Software Compatibility

H.1 General Issues

Merely solving the security and performance problems of capability systems will not make them universally popular. The other major, and perhaps most crucial, problem to solve is software compatibility. Users of computer systems have made major investments in existing software that is designed to run on existing operating systems. Those users are not going to give up that software, simply because a new and secure operating system has been introduced. Similarly, computer vendors and independent software vendors have made major investments in software packages to sell to the users, and those vendors would incur high costs to convert those packages to a new operating system.

Capability-based operating systems have thus far been quite different from conventional operating systems and have maintained little, if any, compatibility with existing software. As a result, the costs of converting to a capability system have been quite high and, combined with the reputation for poor performance, have contributed to capability systems being unpopular.[1]

The history of security kernels has demonstrated the same compatibility problem. The original security kernel for the PDP-11/45, developed by Lee Schiller [188] at the MITRE Corporation, suffered from a lack of compatibility with other PDP–11 operating systems. That incompatibility motivated the development of kernelized versions of UNIX, as originally proposed by Lipner, Beckhardt, and Stork [142].

[1]The one exception to this lack of popularity has been the IBM System/38. While software conversion costs have been significant, the System/38 operating system is much more powerful than the simple operating systems of its immediate predecessors, the IBM System/3, System/32, and System/34. As a result, users have been willing to convert to the System/38 from those systems, but would be less willing to convert from more powerful systems, such as IBM's MVS or VM/370 or AT&T's UNIX.

Integrating the SCAP secure capability architecture with existing operating systems remains a difficult problem, and a full solution is well beyond the scope of this dissertation.

One approach is to build a totally new capability-based operating system. However, the investment required would be massive, and few existing applications could be run on the new system. The SCAP architecture has been designed to be used on VAX processors with only minimal modifications to the processor itself. Therefore, it would be desirable to run existing VAX operating systems, such as ULTRIX-32 (Digital's version of the UNIX operating system) or VAX/VMS.

This appendix briefly examines two ways that SCAP could support existing operating-system code. One is by replacing the UNIX setuid mechanism by SCAP protected subsystems and domains. The other is by creating a virtual machine monitor for compatibility purposes.

H.2 Replacing Setuid

One could build a security kernel on the SCAP architecture that provided an interface that looked much like UNIX. This appears possible, because UNIX already has a limited protected subsystem mechanism based on setuid. This section suggests how one might replace setuid programs with SCAP domains. The proposal here does not address the ideas of light-weight processes, described in Section 6.1.2. It is only a sketch of a design intended as a possible direction for future research.

H.2.1 Setuid Protected Subsystems

The current UNIX protected-subsystem mechanism is based on the setuid and setgid bits in the access mode of a file. When the UNIX shell invokes a program, it uses *fork*(2) [213, p. 2-45] to create a sub-process in which the program is to run. The sub-process then issues the *execve*(2)[213, pp. 2-32 – 2-34] system call to start the program. *Execve* checks the setuid and setgid bits on the file containing the program, and if either is set, changes the effective user ID or group ID of the process to that of the owner of the program. Thus, the owner of the program can create objects that can only be manipulated by the protected subsystem and not by other programs that the user might run. The user who runs a setuid program does not receive reciprocal protection from malicious actions that the program might take. Specifically, the setuid program has full access to the objects that belong to the caller.[2] Thus, setuid is not sufficient to solve the mutually-suspicious subsystem problem [189].

[2]Gould has developed some limitations on setuid programs in its secure UNIX product, UTX/32S [32].

Further, the UNIX setuid mechanism does not provide a convenient method for repeatedly calling a protected subsystem, such as in CAP-I [231]. If the setuid program returns to its caller, the sub-process is destroyed and a subsequent invocation would incur the expense of recreating the sub-process. The parent process could use interprocess communication to pass information to the setuid program, but there is no counterpart of CAP-III's ability to pass capabilities as interprocess messages [93].

H.2.2 SCAP Protected Subsystems in UNIX

Although setuid is clearly insufficient, the concept of running a protected subsystem in a separate address space forms the basis for supporting SCAP protected subsystems and domains.

A protected subsystem is a collection of capabilities and files stored on the disk. A domain is an instance of a protected subsystem in execution. A domain runs at the non-discretionary access class of its creator. A domain in execution bears a strong resemblance to a UNIX process. It has its own address space and registers and will be scheduled independently from other processes or domains. Unlike a UNIX process that is created by the *fork*(2) system call, a domain is created by a new system call that combines the effects of *fork*(2) and *execve*(2) into one call. A process created with *fork*(2) is an exact copy of its parent. A domain, however, always runs a program different from its caller and must be protected from its caller throughout its existence. The new domain-creation call combines the *fork* and *execve* operation into a single, atomic call. The resulting domain will be running in a separate process with the domain's program already loaded. Unlike a child process, a created domain does not inherit opened file descriptors. Instead, the domain receives arguments via a cross-domain call, as described in Chapter 17.

H.3 A Virtual Machine Monitor for SCAP

The other approach to achieving software compatibility without major development expense is to provide a virtual machine monitor as part of the SCAP operating system. If SCAP were built as a set of extensions to the VAX architecture, and if the protection-ring features of the VAX (that is, the four access modes: kernel, executive, supervisor, and user) were retained, then one could implement a virtual VAX CPU in a single domain of protection. Other domains would contain the software that constituted the virtual machine monitor, and that software would simulate the operation of VAX privileged instructions that were trapped by the hardware when the virtual VAX CPU attempted to execute them in an unprivileged domain. With such a virtual machine monitor, one could continue to run unmodified versions of both VAX/VMS and ULTRIX-32 in

parallel with the new capability-based operating system. Further, one could add cross-domain calls to the run-time libraries of the existing operating systems, to allow existing applications to make calls on programs running in different domains, outside the virtual machine monitor.

The principal difficulty with building such a virtual machine monitor is that the VAX architecture is not virtualizable, as defined by Popek and Goldberg [173]. The principal problem is that the Move Processor Status Longword (MOVPSL) instruction is sensitive but not privileged. A sensitive instruction is one that reveals information about the protection state of the processor. To be virtualizable, all such instructions must be privileged, so that they can be trapped and simulated by the virtual machine monitor. This problem (and other related problems) would have to be resolved, before a SCAP operating system could support virtual VAX processors that ran unmodified VAX operating systems. Solving these problems is well beyond the scope of this dissertation.

Appendix I

Annotated Code Sequences

This appendix contains the actual code for some of the experiments described in Chapter 18. Not all of the experiments are included here, because they would be far too voluminous. The intent is to give a flavour of the code involved, rather than complete listings. Only the inner loops of the performance experiments are shown. The driver routines involved in booting the processor, setting up page tables, and interfacing with the clock are not included. The routines are the actual code used, with only minor editing for clarity.

I.1 Call with JSB

Figure I.1 shows the code for test 1 of Table 18.1 on page 184. Test 1 measures the performance of calling and returning with the JSB and RSB instructions. Register 2 is chosen arbitrarily to keep the loop count. The loop count is incremented and tested by the AOBLEQ (add one and branch if less than or equal) instruction. Note that the targets of both the JSB and the AOBLEQ instructions are forced to be aligned on a longword boundary. All VAX branch instructions suffer a performance penalty if their targets are not longword aligned. Forc-

```
        .entry  jsbtest,^m<r2>
        movl    #1,r2           ; start loop at 1
        .align  long,^x01       ; align on longword with NOPs
1$:     jsb     2$              ; jump to the subroutine
        aobleq  #10000,r2,1$    ; do the loop 10000 times
        ret                     ; return to caller

        .align  long
2$:     rsb                     ; simply return from subroutine
        .end
```

Figure I.1: JSB Measurement Code

ing alignment in this and all other experiments makes the performance results independent of any alignment penalties.

I.2 Cross-Domain Call with SVPCTX

Figures I.2 and I.3 show the code for test 5 of Table 18.1 on page 184. Test 5 measures the performance of cross-domain call and return using the SVPCTX and LDPCTX instructions. This test is somewhat more complex, because the calling and called domains run in user mode, while the code sequences that implement cross-domain call and return run in kernel mode.

Figure I.2 contains the code for the calling and called domains and for the kernel-mode dispatcher. The test begins at the entry `svpctx_test`, executing in kernel mode. A PC/PSL pair is pushed onto the stack, and the test executes an REI instruction to begin executing at the label `caller_code` in user mode. The calling domain executes a loop for 10,000 times, just as in Section I.1. The cross-domain instruction (CCALL) is actually a reserved instruction that causes an exception to be handled in kernel mode. Register 0 contains the number of the routine to be called. The SCB entry for reserved-instruction exceptions has been set to point to the label `res_inst`. The code at that point determines whether the reserved instruction was a cross-domain call (CCALL), a cross-domain return (CRET), a test completed (DONE), or some other unexpected reserved instruction. Assuming that the instruction is a CCALL, the code branches to label `cross_call` in figure I.3.

At `cross_call`, the code pushes a frame onto the C-stack and executes the SVPCTX instruction to save the state of the calling domain. As the SVPCTX instruction does not save the entire state, the code executes a series of MFPR instructions to save the memory management registers, ASTLVL, and PME. (See Section D.2.2 for a definition of these internal-processor registers.) The code next range tests the parameter in register 0, loads the new state with the LDPCTX instruction, and executes an REI instruction to label `callee_code` in Figure I.2.

At `callee_code`, a cross-domain return (CRET) instruction is executed in user mode causing a reserved-instruction exception, handled at label `res_int`. From there, the code branches to label `cross_ret` in Figure I.3. The cross-domain return sequence is simpler. The C-stack is popped, and an LDPCTX instruction is executed to load the saved state of the calling domain. An REI instruction is executed to return to user mode and the AOBLEQ instruction continues the loop. At the completion of 10,000 iterations, the code executes a DONE instruction that takes a reserved-instruction exception to enter kernel mode and complete the experiment at the label `all_done`.

```
     .entry  svpctx_test,^m<r2,r9>

ccall_op = ^x01FE
cret_op = ^x02FE
done_op = ^x03FE
     .opdef  CCALL ccall_op,
     .opdef  CRET cret_op,
     .opdef  DONE done_op,

     pushl   #^x03C00000       ; push a PSL with IPL=0,
                               ; CUR_MOD=PRV_MOD=3,
                               ; and all other fields 0
     pushal  caller_code       ; push PC of caller code
     rei                       ; rei to user mode and start

caller_code:                   ; this executes in user mode
     movl    #1,r9             ; start loop at 1
     .align  long,^x01         ; longword align with NOPs
$1:  movl    #1,r0             ; routine number goes in r0
     ccall
     aobleq  #10000,r9,$1      ; loop for 10000 times
     done

     .align  long
callee_code:
     cret

     .align  long
res_inst::                     ; SCB vector branches here
     cmpw    @0(sp),#ccall_op  ; test for cross-call
     beql    cross_call
     cmpw    @0(sp),#cret_op   ; test for cross-return
     beql    cross_return
     cmpw    @0(sp),#done_op   ; test for all-done
     beql    all_done
     halt                      ; halt if something else

all_done:
     addl2   #8,sp             ; pop the frame off the stack
     ret                       ; and return
```

Figure I.2: SVPCTX/LDPCTX Measurement Code, Part 1

```
 .align  long
cross_call:
 subl2   #pcb_length,csp          ; push the CSP
 mtpr    csp,#pr$pcbb             ; point PCBB at the C-stack
 svpctx                           ; save the current context
 movl    csp,r1                   ; now we can use registers
 mfpr    #pr$p0br,pcb.p0br(r1)    ; save p0br
 mfpr    #pr$p0lr,pcb.p0lr(r1)    ; save p0lr
 mfpr    #pr$p1br,pcb.p1br(r1)    ; save p1br
 mfpr    #pr$p1lr,pcb.p1lr(r1)    ; save p1lr
 mfpr    #pr$astlvl,r2            ; get ASTLVL into R2
 insv    r2,#24,#3,pcb.p0lr(r1)   ; and insert into p0lr field
 mfpr    #pr$pme,r2               ; get PME into R2
 ashl    #31,r2,r2                ; shift into high bit
 bisl2   r2,pcb.p1lr(r1)          ; or in PME
 addl2   #2,pcb.pc(r1)            ; increment saved PC to jump
                                  ; over the reserved instruction
; At this point, the caller's state has been saved.
; The next code loads the callees state.
; The experiment uses a single PCB but simulates two PCBs.
; Remember that R0 contains the routine number being called.
 tstl    r0                       ; test r0 for negative values
 blss    error_case               ; reject if negative
 cmpl    r0,#10                   ; compare against max value
 bgtr    error_case               ; reject if greater than max
 movl    pcb.ksp(r0),r0           ; just use up some time
 mtpr    csp,#pr$pcbb             ; put proper address into PCBB
 ldpctx                           ; load the new context
 moval   callee_code,0(sp)        ; fix up new PC
 rei                              ; and REI to user mode

cross_return:
 mtpr    csp,#pr$pcbb             ; load PCBB
 addl2   #pcb_length,csp          ; pop CSP
 ldpctx                           ; load the old context back
 rei                              ; REI back

error_case:
 halt                             ; error cases
 .end
```

Figure I.3: SVPCTX/LDPCTX Measurement Code, Part 2

I.3 Microcode Invoker

Figure I.4 shows the test routine that invokes the microcoded cross-domain call and cross-domain returns instructions used in tests 7 through 11 of Table 18.1 on page 184. The code is very straightforward, compared to the invoking code in Section I.2. There is no requirement to switch between kernel and user modes, because the CCALL and CRET instructions are available in all modes. The instructions use unassigned opcodes that the modified microcode interprets as cross-domain call and cross-domain return instructions. As noted in the comments, the driving test code must have set up a cross-domain linkage table, such that entry one contains the address of `micro$target`.

```
        .entry  microtest,^m<r7>

micro_ccall_op = ^x31FD
micro_cret_op = ^x30FD
        .opdef  CCALL micro_ccall_op,ab
        .opdef  CRET micro_cret_op,

        movl    #1,r7               ; start loop at 1
        .align  long,^x01           ; align to longword with NOPs
$1:     ccall   #1                  ; call to routine #1
        aobleq  #10000,r7,$1        ; loop for 10000 times
        ret

; The cross-domain linkage table has been initialized,
; such that the start address of routine #1 is micro$target
        .align  long
micro$target::
        cret                        ; cross-domain return

        .end
```

Figure I.4: Microcode Invoker

I.4 Cross-Domain Call Microcode

This section contains part of the microcode for test 7 of Table 18.1 on page 184. Test 7 implements cross-domain call and return in microcode, saving and restoring all the general registers every time. It includes none of the optimizations described in Chapter 17. The microcode for test 7 is the best example to include here, because all of the other versions of the microcode are derived from this ver-

sion, with certain functions removed. Appendix E contains a brief summary of the microarchitecture of the VAX-11/730 and should be read before this section.

Only the microcode for cross-domain call is described here. The cross-domain return microcode is omitted, as it does not display any additional interesting features. The microcode is broken down into a series of blocks, each with an accompanying brief description.

The cross-domain call instruction has a two-byte opcode (hex 31FD) and takes one argument—an entry number that specifies which enter capability in the cross-domain call table should be used. The instruction assumes that all of the relevant data structures, such as the C-stack, and the domain-control block, have been paged in and are accessible. If any of the critical data structures is not accessible, the instruction generates a kernel-stack-not-valid abort.

The first block of microcode shows how the instruction decodes its operand, finds the C-stack, and saves the stack pointer and the PSL.

```
.TOC    "   CROSS DOMAIN CALL INSTRUCTION   31FD"
5802:   ; This puts the routine in the middle of the FPA code.
    ; Location is 1802 in FPWARM.MIC
HOT.CROSSCALL:
    DECODE.SPEC ADRS[SRC1.FLT],     ;FETCH 1ST OPERAND
    JSR.IF(IB VALID)                ;TEST FOR IB ERRORS
    JSR [FILL.IB&DECODE.SPEC.ADRS(SRC1.FLT)]

; AT THIS POINT, THE PARAMETER IS IN WR[0]

    MOV WR[0] TO LS[T2]     ;SAVE PARAMETER IN T2
    JSR [IE.SAVE.PC&PSL]    ;SAVE PC AND PSL IN T1 & T0
    MOV LS[PSL] TO WR[0]    ;GET OLD PSL
    AND LS[#1F0000] TO WR[0]    ;NEW PSL IS ALL 0,
                                ;WITH OLD VALUE OF IPL
    JSR [REI.SAVE.OLD.SP]   ;SAVE THE OLD SP REGISTER
    MOV LS[CSP] TO WR[1]    ;GET POINTER TO C-STACK

; NOW LOAD A NEW PSL, SO THAT WE CAN TOUCH THE C-STACK AND OTHER
; THINGS WITH KERNEL MODE ACCESS RIGHTS.

    MOV WR[0] TO LS[PSL.HW] ;LOAD NEW HARDWARE PSL
    SWAP WR[0] WITH LS[PSL] ; AND SAVE OLD ONE
    MOV WR[1] TO LS[T3]     ;GET ADDRESS INTO LOCAL STORE
```

The next block of microcode shows how the entry number is converted into a pointer into the call table on the stack and is checked for validity. This block also illustrates a typical performance optimization for the VAX-11/730. The first microinstruction initiates a longword read from primary memory, but the data is not

available until the third microinstruction. Therefore, the second microinstruction is inserted between the MEM.REQ and the MOV MEM.DATA microinstructions to avoid stalling the processor.

```
    MEM.REQ[READ.V.RCHK] ADRS[T3] DT[LONG]  ;GET PTR TO CALL TABLE
    MOV LS[#4] TO WR[2]          ;GET A CONSTANT 4
    MOV MEM.DATA TO WR[1],       ;READ THE VALUE
     SKIP.IF[MEM.REF.OK]         ; SKIP IF NO PROBLEMS
    JSR [READ.TO.WR1(ERROR)]     ;FIX PROBLEMS OR FAULT
    SUB WR[2] FROM LS[T3],       ;INCREMENT POINTER
     SKIP                        ; AND SKIP
    JMP [CSTACK_INVALID]         ;SERIOUS PROBLEM WITH C-STACK

; WR[1] NOW HAS A POINTER TO THE CALL TABLE

    MEM.REQ[READ.V.RCHK] ADRS[T3] DT[LONG]  ;GET LEN OF CALL TABLE
    MOV LS[T2] TO WR[0]          ;GET CALL NUMBER
    MOV MEM.DATA TO WR[2],       ;READ THE VALUE
     SKIP.IF[MEM.REF.OK]         ; SKIP IF NO PROBLEMS
    JSR [READ.TO.WR2(ERROR)]     ;FIX PROBLEMS OR FAULT
    SHL2 WR[0],                  ;MULTIPLY CALL NUMBER BY 4
     SKIP                        ; AND SKIP
    JMP [CSTACK_INVALID]         ;SERIOUS PROBLEM WITH C-STACK

; WR[2] NOW HAS LENGTH OF CALL TABLE IN BYTES

    CMP WR[0] WITH WR[2],        ;CHECK CALL NUMBER IN RANGE
     DT(LONG)&SET.ALU.CC
    JMP.IF[GEQU] TO [RESERVED.OPERAND.FAULT];JMP IF OUT OF RANGE
    ADD WR[0] TO WR[1]           ;COMPUTE ADDRESS OF CALL VECTOR
```

Next, the microcode locates the domain-control block (DCB), by reading the address from the enter capabilities in the cross-domain call table. Then the microcode reads the new PSL.

```
    MOV WR[1] TO LS[T3]
    MEM.REQ[READ.V.RCHK] ADRS[T3] DT[LONG]  ;GET ADDRESS OF DCB
    MOV LS[CSP] TO WR[0]     ;GET C-STACK POINTER
    MOV MEM.DATA TO WR[2],   ;READ THE VALUE
     SKIP.IF[MEM.REF.OK]     ; SKIP IF NO PROBLEMS
    JSR [READ.TO.WR2]        ;FIX PROBLEMS OR FAULT

; AT THIS POINT, WE HAVE A PTR TO THE DOMAIN CONTROL BLOCK (DCB)
; IN WR[2] WE GOT IT BY READING THE ENTER CAPABILITY STORED IN THE
; CALL TABLE THE FIRST TIME, THE READ WILL FAULT, AND THE O/S WILL
; MAP THE DCB INTO THE VIRTUAL ADDRESS IN KERNEL MODE - SEE THE
```

```
; MEM.REQ INSTRUCTION ABOVE

; NOW WE START ON THE DCB

    MOV WR[2] TO LS[T2]      ;SAVE DCB ADDRESS IN T2
    ADD LS[#20] PLUS WR[2] TO Q     ;MOVE TO END OF DCB
    ADD LS[#C] TO Q          ; CONSTANT 2C CAN'T BE USED
    MOV Q TO LS[T3]          ;AND SAVE IN T3
    MEM.REQ[READ.V.RCHK] ADRS[T3] DT[LONG]  ;START READ OF NEW PSL
    MOV LS[T0] TO WR[1]      ;GET OLD IPL INTO WR[1] FOR USE LATER
    MOV MEM.DATA TO WR[0],   ;READ THE NEW PSL
     SKIP.IF[MEM.REF.OK]     ;SKIP IF NO PROBLEMS
    JSR [READ.TO.WR0]        ;FIX PROBLEMS OR FAULT
```

In this block, the microcode validates the new PSL. The requirements are that the new IS must be zero, the new IPL must be equal to the old IPL, the MBZ fields must contain zeros, and, if the new IPL is greater than zero, then the new current mode must be kernel. The code also checks for entry into PDP-11 compatibility mode.

This block also illustrates the VAX-11/730's delayed branching. The first microinstruction performs a test and sets the condition codes. Before the condition codes are available, there is room for an intervening microinstruction, in this case an AND operation. The SKIP condition is specified as part of the same microword as the AND.

```
; NOW WE MUST VALIDATE THE NEW PSL

    BIT LS[BIT26] WITH WR[0],   ;TEST FOR PSL<IS>
     DT(LONG)&SET.ALU.CC        ; SET THE CONDITION CODES
    AND WR[0] WITH LS[#1F0000] TO Q,    ;GET NEW IPL INTO Q
     SKIP.IF[BITS.CLR]          ; SKIP IF PSL<IS> = 0
    JMP [RESERVED.OPERAND.FAULT];NEW PSL<IS> MUST BE 0

    AND LS[#1F0000] TO WR[1]    ;GET OLD IPL INTO WR[1]
    CMP WR[1] WITH Q,           ;COMPARE IPL VALUES
     DT(LONG)&SET.ALU.CC        ; AND SET CONDITION CODES

;NOW TEST THAT CURRENT MODE = 0 IF IPL > 0

    TST Q,                      ;SEE IF IPL > 0
     DT(LONG)&SET.ALU.CC,       ; AND SET CONDITION CODES
     SKIP.IF[EQL]               ; AND SKIP IF IPLs ARE EQUAL
    JMP [RESERVED.OPERAND.FAULT];IPL CHANGE IS FORBIDDEN
    MOV LS[#B020FF00] TO WR[1], ;GET MASK FOR MBZ AND CM FIELDS
     SKIP.IF[BITS.CLR]          ; SKIP IF IPL > 0
    AND WR[0] WITH LS[#3000000] TO Q,   ;MASK OUT ALL BUT CUR MODE
```

```
     DT(LONG)&SET.ALU.CC        ; AND SET CONDITION CODES
    SKIP.IF[BITS.CLR]           ;SKIP IF IPL > 0 & CUR > 0
                                ; OR IF IPL = 0 (FALL THRU)
    JMP [RESERVED.OPERAND.FAULT];IPL > 0 AND CUR > 0 FORBIDDEN

    AND WR[0] TO WR[1],         ;ISOLATE MBZ AND CM FIELDS
     DT(LONG)&SET.ALU.CC        ; AND SET CONDITION CODES
    BIC LS[BIT31] TO WR[1],     ;MASK ALL BUT MBZ FIELDS
     DT(LONG)&SET.ALU.CC,       ; AND SET CONDITION CODES
     SKIP.IF[NEQ]               ; AND SKIP IF MBZ AND CM NOT 0
    JMP [CSTACK.PUSH]           ;GO START PUSHING REGS
;
; The new PSL had compatibility mode bit set, or MBZ was nonzero.
; Check for PSL<MBZ> all clear, and if so, validate the REI into
; compatibility mode.
;
    MOV LS[#C0000E0] TO WR[1],  ; Get -11 mode MBZ bits.
     SKIP.IF[EQL]               ; Skip if PSL<MBZ> clear.
    JMP [RESERVED.OPERAND.FAULT]; If PSL<MBZ> nonzero, fault.

    AND WR[0] TO WR[1],         ; Mask out all but compatibility
     DT(LONG)&SET.ALU.CC        ; mode PSL<MBZ> bits, & test them.
    AND WR[0] WITH LS[#3000000] TO Q, ; Get new PSL<CUR> in Q,
     SKIP.IF[EQL]               ; skip if -11 mode and MBZ
                                ; fields are clear
    JMP [RESERVED.OPERAND.FAULT]; jump if they are nonzero.

    CMP LS[#3000000] WITH Q,    ; Compare new PSL<CUR> with User,
     DT(LONG)&SET.ALU.CC        ; as -11 mode must be in user mode
    JMP.IF[NEQ] TO [RESERVED.OPERAND.FAULT] ; If not in user mode,
                                            ; fault.

    SET.STATE.ZERO              ; Set "entering -11 mode"
```

Here, the microcode begins to save the state of the calling domain by pushing the memory management registers, PME and ASTLVL onto the C-stack.

```
; NOW START PUSHING REGISTERS ONTO THE C-STACK

CSTACK.PUSH:
    MOV LS[CSP] TO WR[0]    ;GET POINTER TO C-STACK
    MOV WR[0] TO LS[T3]     ;T3 WILL POINT TO THE C-STACK
    MOV LS[#4] TO WR[2]     ;WR[2] WILL CONTAIN CONSTANT 4

; PUSH PME AND P1LR

    SUB WR[2] FROM LS[T3]       ;SKIP OVER 1st 2 ENTRIES
```

```
    SUB WR[2] FROM LS[T3]       ;...
    MOV LS[CONSOLE.CSR.IES] TO WR[1]    ;GET PME
    BIT LS[BIT6] WITH WR[1],    ;FROM BIT 6
     DT(LONG)&SET.ALU.CC
    MOV LS[P1LR] TO WR[1],      ;GET P1LR INTO WR[1]
     SKIP.IF[BITS.CLR]          ; AND SKIP IF PME IS OFF
    BIS LS[BIT31] TO WR[1]      ;SET THE PME BIT
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    MOV WR[1] TO LS[T7]         ;GET VALUE INTO LS
    WRITE.MEM LS[T7],           ;DO THE PUSH
     SKIP.IF[MEM.REF.OK]        ; AND SKIP IF OK
    JSR [WRITE.T7(ERROR)]       ;FIX PROBLEM OR FAULT

; PUSH P1BR

    SUB WR[2] FROM LS[T3],      ;GET NEXT LONGWORD ON C-STACK
     SKIP                       ; AND SKIP
    JMP [CSTACK_INVALID]        ;IF PREVIOUS WRITE FAILED
    MOV LS[P1BR] TO WR[1]       ;GET P1BR INTO WR[1]
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    MOV WR[1] TO LS[T7]         ;GET VALUE INTO LS
    WRITE.MEM LS[T7],           ;DO THE PUSH
     SKIP.IF[MEM.REF.OK]        ; AND SKIP IF OK
    JSR [WRITE.T7(ERROR)]       ;FIX PROBLEM OR FAULT

; PUSH ASTLVL AND POLR

    SUB WR[2] FROM LS[T3],      ;GET NEXT LONGWORD ON C-STACK
     SKIP                       ; AND SKIP
    JMP [CSTACK_INVALID]        ;IF PREVIOUS WRITE FAILED
    MOV LS[POLR] TO WR[1]       ;GET POLR TO WR[1]
    MOV LS[ASTLVL] TO WR[3]     ;OR IN THE AST LEVEL
    BIS WR[3] TO WR[1]          ; USING WR3 DUE TO MICRO LIMITS
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    MOV WR[1] TO LS[T7]         ;GET VALUE INTO LS
    WRITE.MEM LS[T7],           ;DO THE PUSH
     SKIP.IF[MEM.REF.OK]        ; AND SKIP IF OK
    JSR [WRITE.T7(ERROR)]       ;FIX PROBLEM OR FAULT

; PUSH P0BR

    SUB WR[2] FROM LS[T3],      ;GET NEXT LONGWORD ON C-STACK
     SKIP                       ; AND SKIP
    JMP [CSTACK_INVALID]        ;IF PREVIOUS WRITE FAILED
    MOV LS[P0BR] TO WR[1]       ;GET P0BR INTO WR[1]
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
```

```
    MOV WR[1] TO LS[T7]          ;GET VALUE INTO LS
    WRITE.MEM LS[T7],            ;DO THE PUSH
     SKIP.IF[MEM.REF.OK]         ; AND SKIP IF OK
    JSR [WRITE.T7(ERROR)]        ;FIX PROBLEM OR FAULT
```

In the next block, the microcode continues to save the state of the calling domain by pushing the PSL, PC, and the general registers onto the C-stack.

```
; PUSH PSL
    SUB WR[2] FROM LS[T3],       ;GET NEXT LONGWORD ON C-STACK
     SKIP                        ; AND SKIP
    JMP [CSTACK_INVALID]         ;IF PREVIOUS WRITE FAILED
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    MOV LS[#E] TO WR[0]          ;INITIALIZE THE LOOP COUNTER
                                 ; FOR THE GPR SAVE&CLEAR LOOP
    WRITE.MEM LS[T0],            ;WRITE SAVED VALUE OF PSL
     SKIP.IF[MEM.REF.OK]         ; AND SKIP IF OK
    JSR [WRITE.T0(ERROR)]        ;FIX PROBLEM OR FAULT

; PUSH PC

    SUB WR[2] FROM LS[T3],  ;GET NEXT LONGWORD ON C-STACK
     SKIP                   ; AND SKIP
    JMP [CSTACK_INVALID]    ;IF PREVIOUS WRITE FAILED
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    MOV WR[0] TO LS[OS]     ;GET LOOP COUNTER INTO OS
                            ;FOR THE GPR SAVE&CLEAR LOOP
    WRITE.MEM LS[T1],       ;WRITE SAVED VALUE OF PC
     SKIP.IF[MEM.REF.OK]    ; AND SKIP IF OK
    JSR [WRITE.T1(ERROR)]   ;FIX PROBLEM OR FAULT

; NOW PUSH AND CLEAR REGISTERS FP THROUGH R0

    SUB WR[2] FROM LS[T3],  ;GET ADDRESS FOR NEXT PUSH
     SKIP                   ; AND SKIP
    JMP [CSTACK_INVALID]    ;IF PREVIOUS WRITE FAILED
    JSR [CROSS.REGISTER.SAVE.LOOP]  ;START THE LOOP

CROSS.REGISTER.SAVE.LOOP:
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    DEC LS[OS],             ;DECREMENT LOOP COUNTER
     DT(LONG)&SET.ALU.CC    ; AND SET CONDITION CODES
    WRITE.MEM LS[GPR.OS],   ;WRITE THE REGISTER VALUE
     SKIP.IF[MEM.REF.OK]    ; AND SKIP IF OK
    JSR [WRITE.GPR.OS(ERROR)];FIX PROBLEM OR FAULT
    SUB WR[2] FROM LS[T3],  ;GET ADDRESS FOR NEXT PUSH
     SKIP
```

```
    JMP [CSTACK_INVALID]
    CLR LS[GPR.OS],          ;CLEAR THE REGISTER
     LOOP.IF(NEQ)            ; AND LOOP UNTIL REGISTER 0
```

The next block of microcode completes saving the state of the calling domain by pushing five stack pointers (USP, SSP, ESP, KSP, and CSP) onto the C-stack.

```
; NOW PUSH THE STACK POINTERS
;   (REI.SAVE.OLD.SP ALREADY SAVED CURRENT SP)

; PUSH USP
    MOV LS[USP] TO WR[1]     ;GET THE STACK POINTER FROM LS
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    MOV WR[1] TO LS[T7]      ;GET THE STACK POINTER INTO T7
    WRITE.MEM LS[T7],        ;WRITE THE VALUE
     SKIP.IF[MEM.REF.OK]     ; AND SKIP IF OK
    JSR [WRITE.T7(ERROR)]    ;FIX PROBLEM OR FAULT

; PUSH SSP
    SUB WR[2] FROM LS[T3],   ;GET ADDRESS FOR NEXT PUSH
     SKIP                    ; AND SKIP
    JMP [CSTACK_INVALID]     ;IF PREVIOUS WRITE FAILED
    MOV LS[SSP] TO WR[1]     ;GET THE STACK POINTER FROM LS
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    MOV WR[1] TO LS[T7]      ;GET THE STACK POINTER INTO T7
    WRITE.MEM LS[T7],        ;WRITE THE VALUE
     SKIP.IF[MEM.REF.OK]     ; AND SKIP IF OK
    JSR [WRITE.T7(ERROR)]    ;FIX PROBLEM OR FAULT

; PUSH ESP
    SUB WR[2] FROM LS[T3],   ;GET ADDRESS FOR NEXT PUSH
     SKIP                    ; AND SKIP
    JMP [CSTACK_INVALID]     ;IF PREVIOUS WRITE FAILED
    MOV LS[ESP] TO WR[1]     ;GET THE STACK POINTER FROM LS
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    MOV WR[1] TO LS[T7]      ;GET THE STACK POINTER INTO T7
    WRITE.MEM LS[T7],        ;WRITE THE VALUE
     SKIP.IF[MEM.REF.OK]     ; AND SKIP IF OK
    JSR [WRITE.T7(ERROR)]    ;FIX PROBLEM OR FAULT

; PUSH KSP
    SUB WR[2] FROM LS[T3],   ;GET ADDRESS FOR NEXT PUSH
     SKIP                    ; AND SKIP
    JMP [CSTACK_INVALID]     ;IF PREVIOUS WRITE FAILED
    MOV LS[KSP] TO WR[1]     ;GET THE STACK POINTER FROM LS
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
    MOV WR[1] TO LS[T7]      ;GET THE STACK POINTER INTO T7
```

```
    WRITE.MEM LS[T7],        ;WRITE THE VALUE
     SKIP.IF[MEM.REF.OK]     ; AND SKIP IF OK
    JSR [WRITE.T7(ERROR)]    ;FIX PROBLEM OR FAULT

; NOW STORE THE NEW VALUE OF THE CSP REGISTER
    MOV LS[#C] TO WR[1],     ;GET 12 INTO WR[1]
     SKIP                    ; AND SKIP
    JMP [CSTACK_INVALID]     ;IF PREVIOUS WRITE FAILED
    SUB WR[1] FROM LS[T3]    ;LEAVE ROOM FOR THE CAPAB ARG
    MOV LS[T3] TO WR[1]      ;GET VALUE INTO WR[1]
    MOV WR[1] TO LS[CSP]     ;AND FROM THERE TO LS[CSP]
```

Next, the microcode locates the call table of the new domain from the domain-control block and pushes the address and length onto the C-stack.

```
; FIND THE ADDRESS OF THE CALL TABLE OF THE NEW DOMAIN
; AND PUSH IT ONTO THE C-STACK

    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG]  ;START THE MEMORY READ
    ADD WR[2] TO LS[T2]      ;GET POINTER TO NEXT ENTRY
    MOV MEM.DATA TO LS[T7], ;READ THE NEW POINTER
     SKIP.IF[MEM.REF.OK]     ; AND SKIP IF OK
    JSR [READ.TO.T7(ERROR)] ;FIX PROBLEM OR FAULT
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG],;START THE WRITE
     SKIP                    ; AND SKIP
    JMP [CSTACK_INVALID]     ;IF PREVIOUS READ FAILED
    ADD WR[2] TO LS[T2]      ;GET POINTER TO NEXT ENTRY
    WRITE.MEM LS[T7],        ;WRITE THE POINTER TO THE
     SKIP.IF[MEM.REF.OK]     ; NEW CALL TABLE INTO THE
                             ; FIRST LONGWORD OF THE NEW
                             ; C-STACK FRAME
    JSR [WRITE.T7(ERROR)]    ;FIX PROBLEM OR FAULT

; FIND THE LENGTH OF THE CALL TABLE OF THE NEW DOMAIN
; AND PUSH IT ONTO THE C-STACK

    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG], ;START THE MEMORY READ
     SKIP                    ; AND SKIP
    JMP [CSTACK_INVALID]     ;IF PREVIOUS WRITE FAILED
    SUB WR[2] FROM LS[T3]    ;NEXT C-STACK LOCATION
    MOV MEM.DATA TO LS[T7]   ;READ THE NEW LENGTH
     SKIP.IF[MEM.REF.OK]     ; AND SKIP IF OK
    JSR [READ.TO.T7(ERROR)] ;FIX PROBLEM OR FAULT
    MEM.REQ[WRITE.V.WCHK] ADRS[T3] DT[LONG] ;START THE WRITE
;ROOM FOR AN INSTRUCTION HERE
    WRITE.MEM LS[T7],        ;WRITE THE LENGTH OF THE
     SKIP.IF[MEM.REF.OK]     ; NEW CALL TABLE INTO THE
```

```
                              ; SECOND LONGWORD OF THE NEW
                              ; C-STACK FRAME
    JSR [WRITE.T7(ERROR)]     ;FIX PROBLEM OR FAULT
```

In this block, the microcode begins to load the state of the called domain by reading values for the KSP, ESP, SSP, and USP from the domain-control block. The code sequences that load the ESP, SSP, and USP illustrate another optimization. The VAX-11/730 permits a value to be transferred from primary memory into a working register in parallel with the old contents of the working register being written to local store.

```
; READ THE KSP

    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG], ;START THE MEMORY READ
     SKIP                     ; AND SKIP
    JMP [CSTACK_INVALID]      ;IF PREVIOUS WRITE FAILED
    ADD WR[2] TO LS[T2]       ;GET POINTER TO NEXT ENTRY
    MOV MEM.DATA TO WR[1],    ;READ THE KSP
     SKIP.IF[MEM.REF.OK]      ; AND SKIP IF OK
    JSR [READ.TO.WR1(ERROR)];FIX PROBLEM OR FAULT

; READ THE ESP
    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG], ;START THE MEMORY READ
     SKIP                     ; AND SKIP
    JMP [CSTACK_INVALID]      ;IF PREVIOUS READ FAILED
    ADD WR[2] TO LS[T2]       ;GET POINTER TO NEXT ENTRY
    MOV MEM.DATA TO WR[1] XCHG TO LS[KSP],  ;READ ESP & SAVE KSP
     SKIP.IF[MEM.REF.OK]      ; AND SKIP IF OK
    JSR [READ.TO.WR1(ERROR)];FIX PROBLEM OR FAULT

; READ THE SSP
    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG], ;START THE MEMORY READ
     SKIP                     ; AND SKIP
    JMP [CSTACK_INVALID]      ;IF PREVIOUS READ FAILED
    ADD WR[2] TO LS[T2]       ;GET POINTER TO NEXT ENTRY
    MOV MEM.DATA TO WR[1] XCHG TO LS[ESP],  ;READ SSP & SAVE ESP
     SKIP.IF[MEM.REF.OK]      ; AND SKIP IF OK
    JSR [READ.TO.WR1(ERROR)];FIX PROBLEM OR FAULT

; READ THE USP
    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG], ;START THE MEMORY READ
     SKIP                     ; AND SKIP
    JMP [CSTACK_INVALID]      ;IF PREVIOUS READ FAILED
    ADD WR[2] TO LS[T2]       ;GET POINTER TO NEXT ENTRY
    MOV MEM.DATA TO WR[1] XCHG TO LS[SSP],  ;READ USP & SAVE SSP
     SKIP.IF[MEM.REF.OK]      ; AND SKIP IF OK
```

```
    JSR [READ.TO.WR1(ERROR)];FIX PROBLEM OR FAULT
    MOV WR[1] TO LS[USP],   ;SAVE THE USP
     SKIP                   ; AND SKIP
    JMP [CSTACK_INVALID]    ;IF PREVIOUS READ FAILED
```

Here, the microcode continues loading the state of the called domain by reading values for the memory management registers, PME, and ASTLVL from the domain-control block.

```
; Load P0BR

    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG]  ; Start fetching P0BR
    ADD WR[2] TO LS[T2]     ;GET POINTER TO NEXT ENTRY
    MOV MEM.DATA TO WR[0],  ; Read the new P0BR.
     SKIP.IF[MEM.REF.OK]    ; Continue if no error.
    JSR [READ.TO.WR0(ERROR)]; Fix the problem or fault
    NOP,                    ; DO NOTHING
     SKIP                   ; AND SKIP
    JMP [CSTACK_INVALID]    ;IF PREVIOUS READ FAILED

    JSR [MTPR.P0BR]         ; Load the new P0BR.
    JMP [RESERVED.OPERAND.FAULT]    ; Fault if value is reserved.
;
; Read P0LR and ASTLVL, and load P0LR.
;
    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG]
                            ; Start fetching <P0LR & ASTLVL>.
    ADD WR[2] TO LS[T2]     ;GET POINTER TO NEXT ENTRY
    MOV MEM.DATA TO WR[0],  ; Read the new P0LR & ASTLVL.
     SKIP.IF[MEM.REF.OK]    ; Continue if no error.
    JSR [READ.TO.WR0(ERROR)]; Fix the problem or fault

    MOV WR[0] TO LS[T3],    ; Save a copy of ASTLVL in T3<26:24>.
     SKIP                   ; AND SKIP
    JMP [CSTACK_INVALID]    ;IF PREVIOUS READ FAILED

    JSR [MTPR.P0LR]         ; Load P0LR
    JMP [RESERVED.OPERAND.FAULT]    ; Fault if bad value.
;
; Load ASTLVL.
;
    MOV LS[#7000000] TO WR[0] ; Get a copy of ASTLVL in <26:24>.
    AND WR[0] TO LS[T3]     ;Clear all but <26:24>.
    BIC LS[#2000000] TO WR[0] ;Build 5000000 constant
    CMP LS[T3] WITH WR[0],  ; ASTLVL is less than 5.
     DT(LONG)&SET.ALU.CC    ;
```

```
    JMP.IF[GEQU] TO [RESERVED.OPERAND.FAULT];Fault if too large.
    MOV LS[T3] TO WR[0]     ;Get Users ASTLVL for LS location.
;
; Load P1BR.
;
    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG]
    ADD WR[2] TO LS[T2]     ;GET POINTER TO NEXT ENTRY
    MOV MEM.DATA TO WR[0] XCHG TO LS[ASTLVL],; Read the new P1BR.
                            ;and load ASTLVL read before.
     SKIP.IF[MEM.REF.OK]    ; Continue if no error.
    JSR [READ.TO.WR0(ERROR)]; Fix the problem or machine check.
    NOP,                    ; DO NOTHING
     SKIP                   ; AND SKIP
    JMP [CSTACK_INVALID]    ;IF PREVIOUS READ FAILED

    JSR [MTPR.P1BR]         ; Load the new P1BR.
    JMP [RESERVED.OPERAND.FAULT] ; Fault if the value is reserved.
;
; Load P1LR from the DCB.  PME is ignored.
;
    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG]
    ADD WR[2] TO LS[T2]     ;GET POINTER TO NEXT ENTRY
    MOV MEM.DATA TO WR[0],  ; Read the new P1LR.
     SKIP.IF[MEM.REF.OK]    ; Continue if no error.
    JSR [READ.TO.WR0(ERROR)]; Fix the problem or machine check.

    BIT LS[BIT31] WITH WR[0], ; Check if PME should be on.
     DT(LONG)&SET.ALU.CC,
     SKIP                   ; AND SKIP
    JMP [CSTACK_INVALID]    ;IF PREVIOUS READ FAILED

    MOV LS[CONSOLE.CSR.IES] TO WR[1]; Get present PME value.
    BIT LS[BIT6] WITH WR[1],;Check if present PME on.
     DT(LONG)&SET.ALU.CC,
     SKIP.IF[BIT.SET]       ;skip if should be on.
    BIT LS[BIT30] WITH WR[0], ;Perform operation to clear the cc's
     DT(LONG)&SET.ALU.CC,   ;so the next test will pass.
     SKIP.IF[BITS.CLR]      ;skip if should be clear and is.
    JMP.IF[BITS.CLR] TO [PME.CHANGE] ; Jump to change PME.
    JSR [MTPR.P1LR]         ; Load the new P1LR.
    JMP [RESERVED.OPERAND.FAULT] ; Fault if the value is reserved.
```

In this block, the cross-domain call is completed. The microcode flushes the process part of the translation buffer, loads the new PC, SP, and PSL, checks for ASTs and software interrupts, loads the hardware copy of the PSL, and makes a final check for PDP-11 compatibility mode. It then fetches the next VAX (or

PDP-11) instruction. Note that the translation buffer can only be flushed by a microcode loop which writes one entry at a time.

```
; NOW FLUSH THE PROCESS PART OF THE TB
;
    MOV LS[#1FF] TO WR[0]    ;GET ADDRESS IN PAGE BOUNDARY AREA
    MOV WR[0] TO LS[T3]      ; OF FIRST PAGE INTO LOCAL STORE
    MOV LS[#20] TO WR[0]     ;USE LOOP COUNT OF 32
    MOV LS[#400] TO WR[1]    ;INCREMENT PAST TWO ENTRIES AT A TIME
    JSR [CROSS.SWEEP.TB.01] ;START THE LOOP

CROSS.SWEEP.TB.01:
    MEM.REQ[WRITE.TB.STEP] ADRS[T3] DT[LONG]    ;START THE WRITE
    DEC WR[0],               ;DECREMENT THE LOOP COUNT
     DT(LONG)&SET.ALU.CC     ; AND SET THE CONDITION CODES
    WRITE.MEM LS[ZERO]       ;WRITE THE ZERO INTO THE TB
    ADD WR[1] TO LS[T3],     ;INCREMENT TO TWO MORE TB ENTRIES
     LOOP.IF(NEQ)            ; AND LOOP IF MORE ENTRIES

;NOW LOAD PC, SP, AND PSL
    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG]  ;START FETCHING PC
    ADD WR[2] TO LS[T2]      ;INCREMENT TO NEXT LONGWORD OF DCB
    MOV MEM.DATA TO WR[0],   ;GET THE NEW PC VALUE INTO WR[0]
     SKIP.IF[MEM.REF.OK]     ; AND SKIP IF OK
    JSR [READ.TO.WR0(ERROR)];FIX OR FAULT
    MEM.REQ[READ.V.RCHK] ADRS[T2] DT[LONG], ;START FETCH OF PSL
     SKIP                    ; AND SKIP
    JMP [CSTACK_INVALID]     ;IF PREVIOUS READ FAILED
    MOV WR[0] TO LS[PC]      ;PUT VALUE INTO THE PC
    MOV MEM.DATA TO WR[0],   ;GET NEW PSL INTO WR[0]
     SKIP.IF[MEM.REF.OK]     ; AND SKIP IF OK
    JSR [READ.TO.WR0(ERROR)];FIX OR FAULT
    MEM.REQ[READ.V.RCHK.IFILL] ADRS[PC] DT[LONG],
                             ;FLUSH AND FILL IB,
     SKIP                    ; AND SKIP
    JMP [CSTACK_INVALID]     ;IF PREVIOUS READ FAILED
    JSR [REI.LOAD.NEW.SP]    ;LOAD THE NEW SP VALUE
    JSR [REI.LOAD.PSL]       ;LOAD THE PSL
    JSR [REI.CHECK.ASTLVL]   ;CHECK FOR ASTs
    JSR [REI.CHECK.SISR]     ;CHECK FOR SOFTWARE INTERRUPTS
    MOV WR[0] TO LS[PSL.HW] ;LOAD THE HARDWARE WITH THE NEW PSL
    NOP,                     ;WAIT A STATE FOR PSL<T> TO PROPOGATE,
     SKIP.IF[STATE.ZERO.SET]; AND SKIP IF ENTERING -11 MODE
    JMP [IRD]                ;NEXT NATIVE MODE INSTRUCTION
    JMP [CM.RESTORE.Q.AND.IRD] ;NEXT -11 MODE MODE INSTRUCTION
; END OF CROSS-DOMAIN CALL INSTRUCTION
```

Index

*–property. *See* confinement property.

A-segment, 166
A1 rating, 18, 35, 87, 218
  beyond, 218
AAS, 101
Abbott, R. P., 22
abstract data types, 45, 51, 163
abstract type managers, 196
access classes, 33–34, 69–72, 108
access control lists, 26, 31, 51, 67, 74, 80–82, 83, 218
  inspection, 104
  separation-of-duty, 97–100
Access Isolation Mechanism, 35, 119
access matrix, 31–32
access modes, 233
access sets, 73
access-control-list entries, 99
access-control-list system, 17, 68
access-violation fault, 110, 135
Accetta, Mike, 54, 63
accounting records, 71
ACEs. *See* access-control-list entries.
Ackerman, W. B., 27
ACL. *See* access control lists.
Acorn RISC machine, 126
active segment table, 109, 113
active-object manager, 252
active-object table, 252
Ada compiler, 131
Addison, Katherine, 35
address space entities, 54–55
address space numbers, 50, 139–141, 144, 149, 153–159, 195
ADEPT–50, 34
AIM, 119
Air Force
  computer security panel report, 22, 27
$\aleph_0$, 162
Am29000, 126, 162
Am2901A, 241
Ames, S. R., 34, 36, 91
Amoeba, 43, 117
  bank service, 121
Anderson, D. H., 22
Anderson, James P., 22–24, 27
Anderson, M., 73, 120
Anderson, Poul, 21–22
AOBLEQ, 259–260
AP, 231
Archibald, James, 144
argument copying, 134
argument passing, 57, 133, 163–164
  capabilities, 166–169
argument pointer, 231
argument validation, 129–131, 142, 234
ARM, 126
ASNs, 50, 153–159, 186–188. *See also* address space numbers.
ASSIGNED GOTO, 23
assignment statements, 23
assured pipelines, 103
AST level, 232
ASTLVL, 186, 191, 232, 260, 267, 273
ASTs, 191, 235, 274. *See also* asynchronous system traps.
asynchronous system traps, 191, 235
Atlas, 135, 142, 190, 219
atomic transaction protocol, 252
Attanasio, C. Richard, 22

AT&T, 255
audit trail, 39, 96–98, 100, 105, 217
authentication, 28
authentication forwarding, 101
authentication server, 101
authorized individuals, 22
authorized pointers, 72

B–lines, 190
B1 rating, 35, 218
B2 rating, 35, 218
B3 rating, 35, 218
Babaoğlu, Özap, 154
Baer, Jean-Loup, 144
Baldridge, T., 22
bank service, 121
Baron, Larry, 35
Baron, Robert, 54, 63
barrel shifter, 239
Barron, D. W., 26
batch jobs
    name translation in. *See* name translation in batch jobs.
    pre-compiled. *See* pre-compiled batch jobs.
batch queues, 61
Bate, Simon F., 115
BBN Tenex penetration, 22
Beckhardt, S. R., 255
Beckman, Joseph M., 74, 103
Bell and LaPadula security model, 34, 37, 51, 58, 70, 72, 165
Bell, David E., 34–37, 58, 86, 165
benchmarks
    effects of compiler optimization on, 126
Beneich, Denis, 25
Bensoussan, A., 35, 120
Berkeley RISC, 162
Biba integrity model, 37, 95, 100–103, 106, 165
Biba, Kenneth J., 95, 100, 165
Birnbaum, Joel S., 126
Birrell, Andrew D., 57, 62, 64
Bishop, Peter B., 133
Blotcky, Steven, 35
Boebert, W. E., 28, 74, 91, 103
Bolosky, William, 54, 63
Bomberger, A. C., 75
Bonnes, Guus, 22
bootstrapping Trojan horse, 25
bounds checking, 138–139, 195
Boyer, Robert S., 28, 69
Bradshaw, F. T., 34, 36, 91
Bratt, Richard G., 44, 141
Breton, Thierry, 25
browsing, 21–23
Buckingham, B. R. S., 73
buffered-I/O-byte-count quota, 111
Bunch, Steve, 256
Burroughs B6700, 42
    penetration, 22
    Work Flow Language, 89

C-stack, 53–54, 57, 61, 161–181, 185–186, 191–193, 264–271
C-stack-not-valid abort, 169
C.mmp, 27
C1 rating, 217
C2 rating, 217
CAD-based viruses, 24
CAL, 27, 85, 195
CALLS, 185, 187
CALLx, 181, 231
Cambridge
    Capability System. *See* CAP.
    File Server, 42
    ring, 245
    University of, 17
CAP, 14, 17–18, 27, 42, 82, 245
    arguments, 134
    capability unit, 69, 81
    refinements, 133
    sessions, 53
CAP-I, 54, 77, 140, 166, 169, 257
    capabilities
      inform and outform, 62
    cross-domain calls, 62, 161
    ENTER, 63, 163
    LINKER, 62
    operating system, 85
    PARMS, 86
    performance, 187
    PRL, 252

procedure control block, 55
CAP-III, 45, 51, 63, 111, 140, 164, 257
CAP2, 63
capabilities
enter, 189, 246
sealed, 45
software, 51
capability arguments, 130
capability cache, 69, 105
capability list, 83
capability principles, 41–45
capability refinements. *See* refinements.
capability registers, 42
capability segments, 42
capability-based systems, 17, 23, 26–27, 31, 68
lack of popularity, 255
Trojan horses, 25
capability-confinement problem, 79
capability-system performance, 125–127
Carnegie-Mellon University, 68
cartouche, 92
cascaded network connections, 102
Case Western Reserve University, 34, 36, 91
Case, Brian, 162
categories, 33
category management, 104, 106
CDC 6400, 27
CDIs, 38–39, 95–106
certification, 96
Chaitin, G. J., 161
Chan, C., 22
Chandersekaran, C. S., 35, 88
Chang, Albert, 126, 147, 155
Chang, D. P., 22
change mode instructions, 233
Chapman, Robert S., 35, 88
Chehyl, Maureen Harris, 72
Chin, J. S., 22
CHMx, 233
Chow, Frederick, 161
CICS, 96
CISC, 127
Clancy, Gerald F., 44, 141
clandestine software modifications, 23
Clark and Wilson commercial-integrity model, 29, 37–39, 51, 95–106
Clark, David D., 22, 37–38, 95, 221
Clark, Douglas W., 125, 139–140
Classroom Information and Computing Service, 221
clearances, 33–34
CLI-callback mechanism, 86
CLICS. *See* Classroom Information and Computing Service.
CLIMB, 180
CLIPPER, 144
co-routines, 62
Cohen, Fred, 24
collision probability, 149
Colwell, Robert P., 126, 131, 138, 161, 168
Command Definition Utility, 86
command-interpreter domains, 63
command-language interpreter, 14, 52, 61
commercial data integrity, 19, 22, 95–106
communications security, 29
compatibility property, 119
compilers, 117, 196
trap doors in, 24
complete mediation, 27
complete virtualization, 27
Complex Instruction Set Computer, 126
complexity
minimization of, 125
of user interface, 106
Computer Laboratory, 19
computer network, 21
conditional clear on return, 177–181, 188
confined domain, 68
confinement problem, 17–19, 32–34, 67–77, 196

undecidability of, 93
confinement property, 14, 34, 81, 96
conflict matrix, 101
connected-segment manager, 252
console processor, 183, 237, 239
console receiver data buffer, 232
console receiver status, 232
console transmit data buffer, 232
console transmit status, 232
constrained data items, 38–39, 95
context switching, 18, 139–142, 195
Cook, Douglas John, 75, 164, 187
Coordinator Entry, 245
Copple, Mark, 35
copying arguments, 130
Cornwall, Hugo, 21–23
Cosserat, D. C., 27, 42
covert channels, 32, 218
Cragun, Don, 35
CRAY-1, 162
create-authorized-pointer, 72
cross-domain call, 19, 43, 50, 52, 54, 57–58, 62, 127, 131, 189–194, 246, 257
at elevated IPL, 190–191
frequency, 163–164
non-locking, 64
optimization, 161–181, 183–188
performance, 253, 259–275
stack. *See* C-stack.
cross-domain call table, 265
cross-domain linkage table, 170–172, 175, 263
cross-domain return, 50, 191–193
cross-domain-call, 252
cross-ring calls, 130
Crudele, L., 126
cryptographic secret-sharing, 103
CSP, 270
CTL Modular One, 245
Currie, I. F., 27
CUR_MOD, 233–234

D rating, 217
Daley, R. C., 26
Data General
unique-ID machine, 44, 141–142
data path module, 241
data rotator, 239, 242
data-path module, 237
data-type safety, 51
database management, 43
database-management systems, 51
Date, C. J., 105
De Millo, Richard A., 96
DEC/MMS, 88
declining memory costs, 114
DECtape II, 237
DeLashmutt, L., 44, 141
Dellar, Carl Nigel Robert, 245
demand paging, 127, 132
DeMoney, M., 142, 225
denial of service, 31, 39–40
Denning, Dorothy E., 33, 73
Denning, Peter J., 25
Dennis, Jack B., 27, 129
descriptive top level specification, 218
Desmedt, Yvo, 24
Detlefsen, G. D., 140
dial-up line, 21
Digital Equipment Corporation, 17, 163
PDP-1, 180
PDP-10, 180
PDP-10 penetration, 22
PDP-11 compatibility mode, 186. *See also* PDP-11 compatibility mode.
PDP-11/45, 255
Secure Systems Development Group, 19
VAX. *See* VAX.
Dion, Luke, 28
direct penetration, 22–23, 27
directory management, 83–85
directory manager, 252
discretionary access controls, 32, 80–82, 87, 217
discretionary access rights, 54

discretionary Trojan horses, 19, 25, 52, 63, 76, 83–93
disk driver, 250
disk quota, 107–109, 117–121, 250
Dobberpuhl, Daniel W., 223
domain control block, 170–172
Domain Definition Tables, 103
domain enter capabilities, 62
Domain Transition Tables, 103
domain-control block, 264–273
domains, 19, 51, 53–59, 156, 257
  creation of, 62–63, 82
domains of protection, 83, 125, 196
Donnelley, J. E., 22
Doran, R. W., 42
Dotterer, Leslie J., 35, 88
downgrading, 72
Downs, Deborah D., 83
Drongowski, P. J., 35, 72
DTLS, 218
duality of message passing and procedure calls, 53, 63–64
dynamic linker, 62, 170

Eckhouse, Jr., Richard H., 229
Edwards, D. B. G., 142, 219
Edwards, Daniel J., 24
Edwards, P. W., 27
Eggers, S. J., 144
electronic mail, 43, 51, 55, 59
eligibility, 252
Emer, Joel S., 139–140
encapsulation, 54
encryption, 21, 28
English, William, 224
enhanced dynamic linker, 91
enter capabilities, 43–44, 57, 265. *See also* capabilities, enter.
ESP, 179, 186, 232, 270, 272
Estrin, Deborah Lynn, 102
event monitors, 246
eventcounts, 100, 113–116, 250
executive stack pointer, 232
external fragmentation, 133, 138
extracode, 135
EXTRACT, 134–135

Fabry, R. S., 41, 141
factories, 75
Fairchild CLIPPER, 144
false alarm rate, 91
fault tolerance, 165
Faust Mathieu, Hilda, 28
Feiertag, Richard J., 28, 69, 164
Ferguson, C. T., 91
Fernández, Eduardo B., 97, 105
filter, 75
Fishman, D., 22
Fiske, R. S., 86
flat file system, 252
Flex, 75–76, 91–92
Floating-Point Accelerator, 237, 243
formal methods, 218
formal models, 31–40
formal specification, 28
formal top level specification, 218
formal verification, 51, 87
Forsdick, Harry C., 130
FORTRAN, 111
FORTRAN compiler, 86, 91
  Trojan horse in, 84, 86
FORTRAN IV, 23
Fossum, Tryggve, 224
Foster, J. M., 27
FP, 231
FP730, 237
FPA, 237, 243
FPHOT, 242
FPWARM, 242
Fraim, Les, 28
frame pointer, 231
Frantz, W. S., 75
Fraser, A. G., 26
Freeman, C. P., 142
Freitas, D., 126
FTLS, 218
Fu, John, 173

garbage collection, 14, 19, 52, 117–121
Gasser, Morrie, 72, 86
GCOS, 34
  penetration, 22, 23

GCOS 8, 180
GEC 41XX machines, 64
Gehringer, Edward F., 41–42, 149, 161
General Electric. *See* Honeywell.
  GE-600, 225
  GE-645, 130, 219
general-purpose registers, 229, 241
Girling, C. G., 101
Gligor, Virgil D., 35, 88
global page table, 115
GNOSIS. *See* KeyKOS.
Gold, B. D., 72
Goldberg, Robert P., 258
Goldenberg, Ruth E., 115
Golub, David, 54, 63
Goshgarian, P., 22
Gould
  UTX/32S, 256
GPRs, 229, 241. *See also* general-purpose registers.
Graham, G. Scott, 25
Graham, Robert M., 221
Grampp, Frederick T., 22
Gray, J. N., 100
Green, P., 35, 120
Grochow, Jerrold M., 162
Grosso, P., 22
group access control lists, 81
groups, 99–100
Groves, R. D., 142

hackers, 21
Haduch, Kenneth J., 173
Halton, D., 247
Hansen, C., 126
Hansen, Paul M., 125
Hansohn, S. A., 28, 74
Harbison, Samuel P., 27, 68, 83, 195
hardware argument validation, 130
hardware interrupts, 52, 245–247
hardware reliability, 28
hardware verification, 218
hardware viruses, 24
Hardy, N., 75
Harrison, Michael A., 32, 93
Hartley, D. F., 26
hash collision probability, 149
hash deletion
  linear probing, 157
hash function costs, 149
hash-collision resolution
  chaining, 147, 150
  open addressing, 150–158
hashed page tables, 127, 153–159. *See also* inverted page tables.
Hebbard, B., 22
Hecht, Matthew S., 35, 88
Hennessy, John, 161
Herbert, Andrew J., 17, 28, 42, 45, 62–63, 67, 77, 101, 103, 105, 140, 161–164, 245, 257
Hester, Phillip D., 126, 147
Hewlett-Packard
  Spectrum, 126
higher level language, 23
higher level scheduler. *See* scheduler, higher level.
higher-level scheduler, 252
Hilton, T., 22
Hin, Lee Hock, 22
Hinke, T. H., 100
history-dependent control, 97
HMOS, 126
Honeywell
  800, 162
  DPS 8, 180
  DPS 88, 246
  H6000, 225
    penetration, 22
  H6180, 130, 233
    penetration, 22
  H635
    penetration, 23
  H645
    penetration, 22
  LOCK. *See* Secure Ada Target.
  Secure Ada Target. *See* Secure Ada Target.
Hoshen, J., 22
Hospers, Keith, 35
Houdek, M. E., 141, 147, 153–154

Hoult, K., 22
Howarth, D. J., 135, 190
Hudson, E., 126
Huff, George A., 72
human error, 89
human review, 92
Hunt, D., 35, 120
Huntley, G., 22
HYDRA, 27, 68
Hydra, 195

I/O interrupts, 250
I/O manager, 250
I/O statements, 23
Ibbett, Roland N., 140, 143
IBM
    Advanced Administrative System. *See* AAS.
    CICS, 96
    Hursley Laboratory, 96
    RT PC, 126, 142, 147, 150–151, 155
    Secure XENIX, 35
    SWARD, 73
    System/3, 255
    System/32, 255
    System/34, 255
    System/360
      penetration, 22
    System/370, 75
      penetration, 22
    System/38, 27, 42, 72, 125, 141, 147, 149–151, 246, 255
      page tables, 115
      penetration, 22
IC-stack, 191–193
ICCS, 232
ICL
    Perq workstation, 92
ICLIMB, 246
ICR, 232
IDC, 237, 239
in-line expansion, 253
index registers, 190
infinite indirection, 130
inform-capability manager, 252
information flow, 107–109, 118–121
Input-Output Multiplexor, 246
INSERT, 134–135
insiders, 22
instruction alignment, 187, 259
instruction decoding, 185
instruction emulation, 135
Integrated-Disk Controller, 237, 239
integrity, 95–106
integrity access classes, 37
integrity categories, 103, 106
integrity models, 22, 37–39
integrity verification procedures, 38–39, 95
Intel
    432, 27, 125, 161, 163, 168, 246
      Ada compiler, 131
      procedure calling, 131
    8085A, 183, 239
    8086, 126
    HMOS, 126
internal fragmentation, 134
internal processor registers, 229, 232, 241
interprocessor interrupt, 143–144
interrupt C-stack, 191, 193
interrupt handling, 189, 192, 245–247
interrupt priority level, 52, 189–190, 232, 234
interrupt stack, 190–192
interrupt stack pointer, 192, 232
interrupt-latency times, 194
interrupts
    hardware, 52
interval clock control, 232
interval count register, 232
interval timer, 183, 239
inverted page tables, 147–152
IOM, 246
IPL, 52, 232, 234–235, 266
IPRs, 229, 232, 241. *See also* internal processor registers.
ISP, 192, 232
IVPs, 38, 95, 104

Janson, Philippe A., 28, 164, 170, 249, 252–253
Jiang, Wen–Der, 35, 88
job creation, 59
jobs, 53, 58–59
Johnson, Martyn A., 161
Johnson, Mike, 126, 162
Johri, Abhai, 35, 88
Jonekait, Jay, 75
Jordan, Carole S., 83
Jordan, Patricia, 35
Joy, William, 154
JSB, 185, 187, 259–260
Jump-to-Subroutine, 185

KA730, 19, 237, 240
Kahn, Clifford E., 89
Kahn, W., 162
Kain, Richard Y., 28, 74, 76–77, 103
Kampe, Mark, 35
Kanodia, Rajendra K., 100, 113, 250
Karger, Paul A., 17, 22–25, 28, 67, 77, 83, 95, 101, 103–105, 107, 130, 249, 253
Katz, R. H., 126, 144
Keller, James B., 173
Kenah, Lawrence J., 115
kernel mode, 260–263
kernel stack pointer, 232
kernel-segment manager, 250
kernel-stack-not-valid abort, 169, 264
kernel/emulator approach, 35–36
Key Logic, Inc., 75
KeyKOS, 75
keys, 27, 75
Kidder, Tracy., 141
Kilburn, T., 135, 142, 190, 219
Kline, Charles S., 35
Knowles, Alan E., 147
known-object manager, 252
known-object table, 252
Knuth, Donald E., 149–151, 157
Kobziar, A., 35, 120
Konigsford, W. L., 22
Kramer, Steven, 35
KSOS, 35
KSP, 179, 232, 270, 272
Kung, Kenneth C., 83

Lampson, Butler W., 27, 32, 141, 195
  access matrix, 25, 79. *See also* access matrix.
Landau, C. R., 75
Landreth, Bill, 21
Landwehr, Carl E., 34, 76–77
Landy, B., 26
Lanigan, M. J., 142, 219
LaPadula, Leonard J., 34–37, 58, 165
LaTeX, 12
lattice security model, 33–34, 36, 68–77, 82, 96, 217. *See also* Bell and LaPadula security model.
Lattin, William W., 126
Lauer, Hugh C., 53, 63
Lawrence Livermore Laboratory, 162
layered security kernel, 51
LDPCTX, 185–186, 260
Leaman, Jeffrey R., 74, 103
Lechner, Mikel, 35
Lee, T. M. P., 102–104
Leonard, Timothy E., 44, 115, 169–170, 176, 183, 191, 220, 229, 231
levels of abstraction, 28
Levin, Roy, 27, 68, 83, 195
Levitt, Karl N., 28, 69
Levy, Henry M., 41, 72, 125, 229
light-weight processes, 51, 54, 192
Linde, R. R., 72
linear demand-paged virtual memory, 133
linkers, 196
Linton, Mark A., 125
Linus IV, 35
Lipner commercial integrity model, 37, 95, 101, 106

Lipner, Steven B., 28, 33, 35, 64, 83, 100, 103–104, 141, 158, 163, 249, 253, 255
Lipton, Richard J., 96
LISP, 118, 180
Load Process Context, 185
load/store architecture, 49, 126
Lobel, Jerome, 34
local area network, 21
local store, 155, 241, 272
locate mode I/O, 134
LOCK. *See* Secure Ada Target.
Logical Coprocessor Kernel. *See* Secure Ada Target.
login control, 217
LOGIN.COM, 84–86
Lomas, T. Mark A., 90
long returns, 180–181
Lourie, N., 162
low-cost processes, 58
lower level scheduler. *See* scheduler, lower level.
lower-level scheduler, 250, 252
Luckenbaugh, Gary L., 35, 88
Lynch, Kevin, 35

Mach, 54, 63
machine registers, 53, 57
MACRO, 183–184, 186
Manley, Michael, 35
Mann, G. A., 180
MAPARG, 166, 177–180
MAPEN, 232
March, R., 126
Markstein, Peter W., 22
Mashey, J., 142, 225
Mason, Andrew H., 120
master coordinator, 245
Maybury, William, 63
Mayo, A. J., 22
Mayo, Robert N., 125
McCain, Mark, 25
McCauley, E. J., 35, 72
McElroy, James B., 224
MCP
    penetration, 22
MCT, 242
McWilliams, Thomas M., 162
memory clearing, 168
memory controller module, 239
memory management, 126
memory management enable, 232
memory segments, 163
memory tagging, 195
memory-control bus, 237
memory-controller module, 237
Mergen, Mark F., 155
Mesa, 63
message passing, 63–64, 246
    microprogrammed, 64
METAFONT, 12
MFPR, 232, 260
Michigan Terminal System. *See* MTS.
MICRO2, 243
microcode, 135, 218, 237–243
microprogramming, 126
MicroVAX, 135
MicroVAX 78032 chip, 223
MicroVAX-II, 243
military lattice model, 33
Millen, Jonathan K., 72
MIPS Computer Systems, 126, 142, 225
missing-segment faults, 110
MIT PDP-1, 27
Mitchell, G. R., 141, 147, 153–154
Mitchell, James G., 63
MITRE Corporation, 34, 36, 255
    Secure UNIX, 35
modular design, 28
modular programming, 129
Module Management System, 88
Monash University
    password-capability system, 43, 73–74, 117, 120
    rent collection, 120–121
monitors, 63
Montgomery, Warren A., 110, 114, 143
MOO, 162
Moore, J., 142, 225
Morris, Derrick, 140, 143

Morris, Robert H., 22, 149
Morse, Stephen P., 126
mouse, 92
Moussouris, J., 126
MOVC, 134
Move Character, 134
Move Processor Status Longword, 258
Move to Processor Register, 191
MOVPSL, 258
MTPR, 191, 232
MTS
  penetration, 22
MU5, 140, 143
MU6-G, 147
Mullender, Sape J., 43, 121
Multics, 26, 68, 98, 107, 115, 140, 143, 219–220, 233, 249
  Access Isolation Mechanism, 119
  AIM, 35
  argument validation, 129
  descriptor segments, 252
  disk quota, 119
  dynamic linker, 170
  eligibility, 252
  file DIM, 164
  file system, 119
  interprocess signals, 192
  missing-segment-faults, 81
  penetration, 22
  PL/I, 131
  procedure call, 131
  Project Guardian, 35
  protection rings, 44
  revocation, 109–113, 115
  ring-alarm register, 191
  salvager, 252
  security kernel, 249, 253
  shared page tables, 115
  trap door, 24
multiple register sets, 161–163
Multiple-Access, 45
multiprocessor kernel, 57
multiprocessors
  shared memory, 143–145
Mundie, Craig J., 44, 141
Murphy, Marguerite, 125
Murray, W. R., 101
mutual suspicion, 43
mutually-suspicious subsystems, 133, 165, 256
MVS, 255
  penetration, 22
Myers, G., 73

N-segment, 166
name checking with access control lists, 90
name translation in batch jobs, 88–90
name-checking protected subsystem, 52, 85–93
National Computer Security Center, 18, 35–36, 87, 217
Naval Postgraduate School, 249
need-to-know, 217
Needham, Roger M., 26–27, 42, 53, 63, 69, 86, 101, 169, 245, 252, 257
Neely, Rich, 28
Nelson, Bruce Jay, 57, 64
network driver
  Trojan horse, 24
Neumann, Peter G., 26, 28, 69
next interval count register, 232
Nibaldi, G. H., 35
NICR, 232
non-discretionary, 14, 17, 31–37, 51, 54, 58–63, 67, 70, 72, 74–77, 81–82, 83, 87, 91–92, 107, 109, 117–119, 165, 217
non-discretionary access controls, 32–36, 58–59, 107
nuclear weapons, 37

object deactivation, 114
object manager, 252
object-oriented programming, 45, 129, 138
objects, 25
Ogden, W. F., 34, 36, 91
on-line programming, 23

opcodes, 233
operand specifiers, 233
operating-system emulator, 35
operators, 29
orange book, 36, 87, 217
Organick, Elliott I., 27, 42, 68, 81, 131, 164, 170, 219, 246, 252
OS/360
  penetration, 22
overlapping register sets, 162
Oxford University Programming Research Group, 96

P0 base register, 232
P0 length register, 232
P0 space, 132, 153–154, 158, 221, 233
P0BR, 179, 232
P0LR, 179, 232
P1 base register, 232
P1 length register, 232
P1 space, 132, 153–154, 158, 221, 233
P1BR, 179, 232
P1LR, 179, 232
Paans, Ronald, 22
Page Address Registers, 219
page alignment, 133
page faults, 250
page size, 153–154
page tables, 147–152, 219–222
page-fault handlers, 49
page-table entries, 115, 233
page-table manager, 250
panel report, 22, 27
Pardoe, J. B. D., 164
PARMS, 89. *See also* CAP-I, PARMS.
PARs, 219
partial ordering, 33
partially-trusted subjects, 103
PASSARG, 177–180
passive-object manager, 252
password, 28
  guessing, 21
password capabilities. *See* Monash University, password-capability system.
password-capability systems, 41–43
Patterson, David A., 125–126, 162
Payne, R. B., 135, 190
PC, 185, 192, 229, 260, 269, 274
PCB, 170
PCBB, 185, 232
PCLIMB, 180
PDP-11 compatibility mode, 186, 266, 274
Peeler, R. J., 72
performance, 14, 17–18
performance monitor enable, 232
Perkins, C. L., 144
Perlis, Alan J., 96
Phillips, Ray J., 22
physical security, 28
PL/I, 131, 183
Plessey System 250, 27, 42, 77, 247
Plummer, W. W., 27
PME, 186, 232, 260, 267, 273
Pohlman, William B., 126
polling system, 247
Popek, Gerald J., 35, 258
popular press, 21
Pose, R. D., 73, 120
post-authorization, 90
pre-compiled batch jobs, 88
primary-memory page manager, 250
principal, 53
PRL, 170, 252
PROBE instructions, 234
probing
  avoiding, 169
procedural security, 23, 29
procedure calls, 63–64
procedure-entry masks, 170
process control block, 170
process control block base, 232
process creation, 61
process resource list, 170, 252
process space, 153, 221, 224, 233, 239

process switching, 126
processes, 19, 51, 53–54, 58–59, 250
processor status longword, 176, 192, 229
processor-control-block-base, 185
program counter, 192, 229
program-integrity, 102
programming generality, 14, 19, 51, 129–135, 195
proof of program correctness, 104
propagation of capabilities, 68
protected subsystem, 19, 43–45, 53–58, 257
  creation, 61
  enter capability, 62
  Trojan horse in, 67
protection domain, 14, 43, 130
protection rings, 180
prototype implementation, 14
Provably Secure Operating System, 69
proxy-login, 101
PRV_MOD, 234
Przybylski, S., 126
PSL, 176, 178, 186, 192, 229, 231, 233–234, 260, 264, 266, 269, 274
PSOS, 68–69, 73–77
PTE, 115, 233
public-domain software
  Trojan horses, 25

query optimization, 105
query-based transaction system, 23
quota cells, 117, 120
quotas, 39

Rabin, Michael O., 103
Rajunas, S. A., 75
Rashid, Richard, 54, 63
Rattner, Justin, 126
RB730, 237
Reach, R., 162
read down, 72
real-time, 189–194
recompaction, 138
Record Management System, 134
Redell, David D., 45, 81, 109–113
reduced instruction set computer, 14, 18. *See also* RISC.
reduced instruction set computers, 126
Reed, David P., 57, 100, 113, 130, 250, 252
  two-level scheduler, 57–58
REFINE, 134–135
refinements, 41, 132–135
register allocation
  by trust, 164–166, 186, 188
register assignment
  graph coloring, 161
  link time, 161–181
register file, 49
register masks, 170, 186
register saving, 189, 246
  speed, 172–173
register windows, 162
registration-and-sequencing server, 101
REI, 172, 191–193, 260
relational database management system, 125
remote procedure calls, 57, 64
removable disk packs, 250
rent collection, 52, 117, 120–121
RET, 185, 231
Return from Exception or Interrupt, 191
Return-from-Procedure, 185
Return-from-Subroutine, 185
Revenel, Bruce W., 126
revocation, 14, 19, 51, 68, 81, 107–116, 118, 196, 250
  by chaining, 52, 115–116, 152
  chaining inverted-page-table entries, 152
  with eventcounts, 52, 109, 113–116, 143, 152
revoker capabilities, 81, 111–113
ring-alarm register, 191
rings of protection, 43, 233
Riodan, T., 126

RISC, 14, 18, 49, 74, 82, 126, 142, 186, 193, 196, 225, 227
  cross-domain call, 175–176
Ritchie, Dennis M., 61
RMS, 134
  locate mode I/O, 134
Robinson, Lawrence, 28, 69
Rounds, W. C., 34, 36, 91
Rowen, C., 126
Royal Signals and Radar Establishment, 75, 91
RPG III, 125
RSB, 185, 259–260
RSRE, 75, 91
RSS, 101
Rub, Jerzy R., 83
Ruzzo, Walter L., 32, 93
RXCS, 232
RXDB, 232

S–1, 162
S0 space, 132, 153, 155, 158, 233
S1 space, 153, 155, 233
sabotage, 31, 36–39, 92
Saltzer, Jerome H., 41, 43–44, 85, 129–130, 191, 221, 233
sanitization, 72
SAT, 74, 77, 103
Save Process Context, 185
Saxena, A. R., 250
Saydjari, O. Sami, 74, 103
SBR, 232
SCAP, 14, 18, 39, 67–77, 97, 103, 105, 118, 189–194, 243, 256. *See also* secure capability architecture.
  architecture, 49–52
  domain, 58
  domain model, 53–64
  process, 54–55, 58
  security kernel, 61
  storage management, 133
SCB, 192, 234, 260
SCBB, 232
Schaefer, Marvin, 28, 72, 100
Schaufler, Case, 35
scheduler, 192–193
  higher level, 58
  lower level, 58
scheduling entities, 53–54
Scheid, J. F., 72
Schell, Roger R., 22–25, 28, 57, 102, 130, 249
  multiprocessor kernel, 57
Schiller, W. L., 255
Schleimer, Stephen I., 44, 141
Schmookler, M. S., 142
Schrimpf, H., 162
Schroeder, Michael D., 35, 41, 43–45, 85, 129–130, 140, 165, 181, 191, 221, 233, 249, 256
scratch files, 88
SDW, 26
seal, 51
sealed capabilities. *See* capabilities, sealed.
sealing, 163
search lists, 85
Secure Ada Target, 69, 74, 103
secure attention key, 88
secure capabilities, 51, 69–82
secure capability architecture, 14, 17–18, 67–77, 256
secure committees, 103
secure document manager, 76
secure documents, 69
secure object manager, 69
secure readers–writers problem, 100
secure server, 87–88, 120
  domain, 61–62
  process, 59
Secure Systems Development Group, 19
secure window manager, 92
security kernel, 17, 27–28, 35, 54, 57, 61, 87, 111, 157–158, 190, 196, 218, 255
  design, 249–253
  layered, 51, 163
  SCAP, 61
  size, 28
  technology, 23

security manager, 81
security officer, 32, 38
security verification, 72
security-kernel domain, 72
segment descriptor word, 26
segmentation, 133, 137–139, 143
sensitive instructions, 258
sensitivity levels, 33
separation of duties, 37–39, 95–106
setgid, 256
setuid, 256
Shamir, Adi, 103
shared memory multiprocessors, 53. *See also* multiprocessors, shared memory.
shared page tables, 115, 143
Sheldon, R. G., 144
Shirley, Lawrence J., 102
Shockley, William R., 102–104
Shumway, D. G., 34, 36, 91
SID, 232
SIDEARM. *See* Tagged Object Processor.
simple security property, 34
simplicity, 27
Simpson, Richard O., 126, 147
single-level store, 125
SIRR, 232, 235
SISR, 232, 235
Sites, R. L., 162
Slinn, Christopher John, 133, 164
SLR, 232
Smith, Alan Jay, 137
Smith, Leroy, 27
snoopy cache, 144
snoopy translation buffers, 143–144. *See also* translation buffers, snoopy.
software, 274
software capability systems, 27
software compatibility, 14, 196, 255–258
software developers, 82
software interrupt request register, 232
software interrupt summary register, 232
software interrupts, 191, 235
Soleglad, Mike, 28
SP, 229, 274
special directory trees, 88
Sperry 1100
    penetration, 22
spooled stream protected procedure, 164
SQL, 89
SSP, 179, 186, 232, 270, 272
stack pointer, 229, 264
Stanley, Margaret, 91
Steele Jr., Guy Lewis, 180
Stern, J., 35, 120
Stewart, Lawrence C., 144
Stolarchuk, M., 22
storage channels, 32, 35, 93, 107–109, 111, 117, 165
storage management, 133
storage quotas, 52
Stork, D. F., 255
Stoughton, Allen, 35
strict need-to-know, 91
Stroustrup, Bjarne, 180
Sturgis, H. E., 27, 85, 195
subjects, 25
subroutine-call stack, 53
subroutines, 23
Summers, Rita C., 97, 105
Sumner, F. H., 142, 219
Sun
    Secure SunOS, 35
supervisor stack pointer, 232
Supnik, Robert M., 223
SVPCTX, 185–186, 260
SWARD. *See* IBM, SWARD.
Sweet, Richard, 63
system base register, 232
System Control Block, 189–190, 192, 234
system control block base, 232
system high, 33, 71
system identification, 232
system length register, 232

system low, 33
system managers, 29, 32
system penetration, 21
system space, 132, 153, 192, 221, 224, 233, 239

Tabak, Daniel, 126
tagged capability architecture, 91
tagged memory, 42
Tagged Object Processor, 74
tail recursion, 180
tamper resistance, 27
tampering, 31, 36–39, 92
Tangney, John D., 27
Tasker, P. S., 86
TBCHK, 232
TBIA, 232
TBIS, 144, 232
Tenex penetration, 22
terminal input routine
  Trojan horse, 24
Tevanian, Avadis, 54, 63
TEXdraw, 12
TEXindex, 12
text editor, 117
  human interface, 87
  Trojan horse in, 85
Thacker, Charles P., 144
Thakkar, Shreekant S., 147
Thompson, Ken, 24, 61
threads, 54, 58
  creation, 61
tickets, 27
tightly-coupled multiprocessor, 41, 51, 61–62, 68
time-of-year clock, 239
time-of-year register, 232
timing channels, 32
Titan, 45
  multi-access system, 26
TODR, 232
token capabilities, 97–106
Torres, A., 142
TPs, 38–39, 95–106
traceability of access, 14, 19, 51, 68, 79, 196
trailer records, 111
trailer-record quota, 111
transaction processing, 43
transformation procedures, 38–39, 95
translation buffer, 19, 50, 114, 127, 137–145, 246
  direct mapped, 188, 223–226, 239
  filling, 50, 114, 142–143, 219
  flushing, 139–140, 143, 157, 186, 274
  fully associative, 219, 223
  level of associativity, 137, 223–227
  manager, 250
  refills, 183
  set associative, 223, 226–227
  snoopy, 144–145
  swapping, 139–140
translation buffer check, 232
translation buffer invalidate all, 232
translation buffer invalidate single, 144, 232
transparent security, 82
trap door, 23–25, 40
  feasibility, 24
Trojan horse pointer, 234
Trojan horses, 23–25, 27, 32–34, 36, 39, 72, 104
  discretionary, 14, 52. *See also* discretionary Trojan horses.
  non-discretionary, 165
trusted linker, 61, 169–170, 186, 195
trusted path, 86, 218
trusted processes, 72
TU58, 237, 239, 243
Tukubo, S., 22
Turing Award Lecture, 24
Turing-machine halting problem, 32, 39
two-person control, 37
two-phase commit, 100
two-phase commit protocol, 100, 252
two-way authentication, 88

TXCS, 232
TXDB, 232
Tygar, J. D., 103
Tymshare Corporation, 75
type managers, 28, 249
type-id, 45
typed memory, 42

UCLA Secure UNIX, 35
UDIs, 102
UIC, 99
ULD file, 243
ULDTOBIN, 243
Ullman, Jeffrey D., 32, 93
ULTRIX-32, 52, 256–257
unaligned refinements, 134
unauthorized actions, 22
unauthorized disclosure, 31
unauthorized pointers, 72
UncfRts, 68
unconfined rights, 68
unconstrained data items, 38–39, 95
UNIBUS, 237, 239
unique-ID addressing, 131, 139, 141–142
University of Cambridge
    Computer Laboratory, 19
University of Manchester, 140, 142, 147
UNIX, 26, 61, 142, 149, 255
    /tmp directory, 88
    4.2bsd, 153
    C compiler
      trap door, 24
    execve(2), 256–257
    fork(2), 61, 256–257
    kernelized, 255
    make(1), 88
    penetration, 22
    processes, 54, 257
    protected subsystems, 256–257
    setuid, 45
    shell, 256
unseal, 51
upgraded directory, 119
Urban, Michael, 35
user identification code, 99
user interface complexity, 106
user mode, 260–263
user profile, 72
user stack pointer, 232
USP, 179, 232, 270, 272

Van Horn, Earl C., 27
Van't Hof, D., 126
Vasudevan, N., 35, 88
VAX, 18, 74, 140, 144, 149, 189–193, 220–221, 229–235
    access modes, 257
    ASTLVL, 191
    calling standard, 165, 185
    instruction set, 50
    interrupt priority levels, 190
    page tables, 115, 158
    penetration, 22
    protection rings, 44
    security kernel, 163, 181
    subsetting rules, 183
    virtual address, 132
VAX 8550, 19, 183–185
VAX 8600, 224
VAX 8800, 173
VAX DATATRIEVE, 89
VAX DBMS, 44
VAX-11/730, 14, 18–19, 49–50, 82, 147, 150, 153, 173, 181, 183–188, 196, 224, 237–243, 263–275
VAX-11/780, 125, 139, 173, 227, 237
VAX-11/785, 227
VAX/VMS, 44, 52, 98–99, 115, 149, 256–257
    disk quota, 120
    jobs, 53
    penetration, 22
    process, 55
    processes, 54
    quotas, 111
    security enhancements, 35
verification, 28
Viney, I. T., 22
virtual address space, 195

virtual machine monitor, 257–258
virtual memory, 27, 219
virtual money, 120
virtual processors
    level one, 58, 250
    level two, 58, 252
virtual-machine environment, 192
virtualization
    complete, 27
virus, 24
VM/370, 255
    penetration, 22
vp1s, 250
vp2s, 252
vulnerabilities, 21, 129

Waldecker, D. E., 142
Walker, R. D. H., 245
Wall, David W., 161, 169
Wallace, C. S., 73, 120
Wallach, Steven J., 44, 141
Walter, K. G., 34, 36, 91
Walton, Evelyn J., 35
Ward, P. D., 72
Warner, L., 22
Watson, Desmond John, 63
WCS, 19, 237, 239, 243
Webb, D. A., 22
Weissman, Clark, 34
well-formed transactions, 37–39
WFL, 89
Wheeler, David J., 141
Whiteside, T. G., 142
Whitmore, J., 35, 120
Widdoes, Jr., L. Curtis., 162
wide area network, 21
wildcard
    authorization, 89
    separation-of-duty, 98
Wilkes, Maurice V., 17–18, 27, 42, 63, 69, 86, 126, 169, 173, 252, 257
Wilkinson, A. L., 22
Williams, R., 22
Wilson, David R., 37–38, 95
Wimbrow, J. H., 101
window-based editor, 92
wiretapping, 21
Wiseman, Simon, 75, 91
Witek, Richard T., 223
Witten, Ian H., 25
Wood, Christopher, 97, 105
Wood, D. A., 144
Woodward, J. P. L., 35
Wordsworth, J. B., 96
Work Flow Language
    B6700, 89
working register, 272
workstation, 53
    physical security, 28
Worley, Jr., William S., 126
Wright, C. G., 142
Wright, W., 22
writable-control store, 19, 237, 239
write up, 72
Wulf, William A., 27, 68, 83, 195
WWMCCS, 34

XENIX, 35

Young, Michael, 54, 63
Young, W. D., 28, 74

Zilog Z8000, 249