

NAVAL POSTGRADUATE SCHOOL Monterey, California



THESIS

**LINUX, OPENBSD, AND TALISKER:
A COMPARATIVE COMPLEXITY ANALYSIS**

by

Kevin R. Smith

March 2002

Thesis Advisor:

Cynthia E. Irvine

Second Reader:

Timothy E. Levin

Approved for public release; distribution is unlimited.

This thesis was completed in cooperation with the
Institute for Information Superiority and Innovation.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE March 2002	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: Linux, OpenBSD, and Talisker: A Comparative Complexity Analysis			5. FUNDING NUMBERS
6. AUTHOR(S) Kevin R. Smith			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE
13. ABSTRACT (maximum 200 words) Security engineering requires a combination of features and assurance to provide confidence that security policy is correctly enforced. Rigorous engineering principles are applicable across a broad range of systems. The purpose of this study is to analyze and compare three operating systems, including two general-purpose operating systems (Linux and OpenBSD) and a commercially available, embedded operating system (Talisker). The basis for the comparison considers secure software design principles, such as information hiding, hierarchical structuring, and modularity, as well as software complexity metrics, such as the McCabe Cyclomatic Complexity and the number-of-lines-of-code. In this analysis, we use a reverse engineering tool to show how the three operating systems compare to each other with respect to the qualities of a secure operating system design. The operating systems, their kernels, and their scheduling subsystems are analyzed and compared. From the results, it is shown that the OpenBSD operating system, kernel, and scheduler are the best when considering hierarchical structuring, modularity, and information hiding. The Linux kernel and scheduler and the Talisker operating system are least complex when considering the McCabe complexity and the number-of-lines-of-code.			
14. SUBJECT TERMS Protection, Secure Operating Systems, Security Kernel, Cyclomatic Complexity, Software Development, Reference Monitor Concept, Trusted Computing Base, Multics, Verification, Assurance.			15. NUMBER OF PAGES 153
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

This thesis was completed in cooperation with the
Institute for Information Superiority and Innovation.

LINUX, OPENBSD, TALISKER: A COMPARATIVE COMPLEXITY ANALYSIS

Kevin R. Smith
Lieutenant, United States Navy
B.S., Boston University, 1994

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
March 2002**

Author:

Kevin R. Smith

Approved by:

Cynthia Irvine, Thesis Advisor

Timothy Levin, Second Reader

Christopher Eagle, Chair
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Security engineering requires a combination of features and assurance to provide confidence that security policy is correctly enforced. Rigorous engineering principles are applicable across a broad range of systems. The purpose of this study is to analyze and compare three operating systems, including two general-purpose operating systems (Linux and OpenBSD) and a commercially available, embedded operating system (Talisker). The basis for the comparison considers secure software design principles, such as information hiding, hierarchical structuring, and modularity, as well as software complexity metrics, such as the McCabe Cyclomatic Complexity and the number-of-lines-of-code. In this analysis, we use a reverse engineering tool to show how the three operating systems compare to each other with respect to the qualities of a secure operating system design. The operating systems, their kernels, and their scheduling subsystems are analyzed and compared. From the results, it is shown that the OpenBSD operating system, kernel, and scheduler are the best when considering hierarchical structuring, modularity, and information hiding. The Linux kernel and scheduler and the Talisker operating system are least complex when considering the McCabe complexity and the number-of-lines-of-code.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	PURPOSE	1
B.	MOTIVATION.....	1
C.	EXPECTED BENEFITS OF RESEARCH	2
D.	RESEARCH OBJECTIVES	2
E.	THESIS ORGANIZATION.....	3
II.	OVERVIEW OF SECURE OPERATING SYSTEMS.....	5
A.	INTRODUCTION.....	5
B.	ASSESSMENT OF THREATS TO SECURITY	5
C.	PROTECTION IN OPERATING SYSTEMS.....	10
D.	THE REFERENCE MONITOR CONCEPT	11
E.	THE TRUSTED COMPUTING BASE.....	16
F.	CONCLUSION.....	17
III.	BUILDING A SECURE OPERATING SYSTEM.....	19
A.	INTRODUCTION.....	19
B.	THE SECURITY POLICY AND MODEL	20
1.	The Policy.....	20
2.	The Model	20
C.	TRUSTED SYSTEM DESIGN ELEMENTS.....	21
D.	SECURE SOFTWARE DESIGN PRINCIPLES	23
1.	Modular Programming.....	23
2.	Hierarchical Structuring	24
3.	Information Hiding	25
E.	THE SECURITY KERNEL.....	26
F.	EVALUATION.....	27
G.	ASSURANCE	30
H.	THE MULTICS EXPERIENCE	31
1.	Constructing a Secure System.....	32
2.	Common Mechanism Risks.....	33
3.	Complexity and Dependencies	35
4.	Security Kernel Design	36
a.	<i>Non-kernel Software</i>	<i>37</i>
b.	<i>Multics Security Kernel Goals</i>	<i>37</i>
c.	<i>Categories of Security Kernel Activities</i>	<i>37</i>
d.	<i>Redefining Key Aspects of the Project.....</i>	<i>38</i>
e.	<i>Conclusions of the Kernel Design and Redesign.....</i>	<i>39</i>
IV.	MANAGING COMPLEXITY IN OPERATING SYSTEM DESIGN.....	41
A.	INTRODUCTION.....	41
B.	BUILDING SYSTEMS THAT FAIL	42
C.	SOFTWARE COMPLEXITY	43

D.	COMPLEXITY AND TESTING	45
E.	MCCABE CYCLOMATIC COMPLEXITY METRIC	47
1.	Definition and Applications	47
2.	Limiting the Cyclomatic Complexity	49
3.	Simplified Complexity Calculations	51
4.	Automated Tool	53
V.	IMAGIX 4D REVERSE ENGINEERING TOOL	55
A.	INTRODUCTION	55
B.	DATA MODEL	55
C.	VIEWS AND DISPLAY WINDOWS	57
1.	Graph Window	57
2.	Graph Symbols Key	60
3.	List Window	60
4.	File and Class Browsers	60
5.	Flow Chart	61
6.	File Editor	62
7.	Reports	64
D.	MODES OF OPERATION	64
1.	Browsing the Source Code	65
2.	Exploration and Analysis of the Source Code	66
3.	Control Flow Analysis of the Source Code	69
E.	CONCLUSION	71
VI.	COMPLEXITY ANALYSIS OF LINUX	73
A.	INTRODUCTION	73
B.	BACKGROUND	73
1.	Linux	73
2.	OpenBSD	74
3.	Talisker	76
C.	BASIS FOR ANALYSIS	77
1.	Hierarchical Structuring	77
2.	Modular Programming	78
3.	Information Hiding	79
4.	McCabe Cyclomatic Complexity	80
5.	Number-of-Lines-of-Code	81
D.	SOURCE CODE FILES INCLUDED IN THE ANALYSIS	81
1.	Operating System Source Code	81
2.	Kernel Source Code	82
E.	INFORMATION GATHERED FROM IMAGIX 4D	84
1.	Function Call Graphs	84
a.	Kernel Function Hierarchies	85
b.	Schedule Module Function Hierarchies	88
c.	Operating System Layering	91
d.	Kernel Layering	94
2.	File Summary Reports	96
a.	Operating System Files	97

	<i>b.</i>	<i>Kernel Files</i>	98
3.		Function Information Reports	99
	<i>a.</i>	<i>Operating System Functions</i>	100
	<i>b.</i>	<i>Kernel Functions</i>	101
	<i>c.</i>	<i>Schedule Module Functions</i>	103
4.		Global Variable Graphs	105
	<i>a.</i>	<i>Kernel Global Variables</i>	106
	<i>b.</i>	<i>Schedule Module Global Variables</i>	109
F.		ANALYSIS OF DATA GATHERED	111
	1.	Hierarchical Structuring	111
		<i>a.</i> <i>Operating System Layering</i>	112
		<i>b.</i> <i>Kernel Layering</i>	112
	2.	Modularity	113
		<i>a.</i> <i>Operating System Code</i>	113
		<i>b.</i> <i>Kernel Code</i>	114
		<i>c.</i> <i>Schedule Module Code</i>	115
	3.	Information Hiding	116
		<i>a.</i> <i>Operating System Global Variables</i>	116
		<i>b.</i> <i>Kernel Global Variables</i>	117
		<i>c.</i> <i>Schedule Module Global Variables</i>	118
	4.	McCabe Cyclomatic Complexity Distribution	118
		<i>a.</i> <i>Operating System Complexity</i>	119
		<i>b.</i> <i>Kernel Complexity</i>	119
		<i>c.</i> <i>Schedule Module Complexity</i>	120
	5.	Number-of-Lines-of-Code	121
		<i>a.</i> <i>Operating System Code</i>	121
		<i>b.</i> <i>Kernel Code</i>	122
		<i>c.</i> <i>Schedule Module Code</i>	122
G.		COMPLEXITY IN OPERATING SYSTEMS	123
H.		CONCLUSION	123
VII.		CONCLUSION	125
	A.	OPERATING SYSTEM SECURITY REQUIREMENTS	125
	B.	SECURE OPERATING SYSTEM DEVELOPMENT	125
	C.	COMPARATIVE ANALYSIS CONCLUSIONS AND COMMENTS ..	127
	D.	FUTURE WORK	129
		LIST OF REFERENCES	131
		INITIAL DISTRIBUTION LIST	135

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1.	User and Operating System Protection Domains.....	12
Figure 2.	Gate Calls in Protection Domains.....	13
Figure 3.	The Reference Monitor Concept.....	14
Figure 4.	Sample Monolithic TCB Architectures.....	16
Figure 5.	TCSEC Requirements for Assurance.....	29
Figure 6.	CC Evaluated Assurance Levels (EAL).....	30
Figure 7.	Euclid Module Code With Cyclomatic Complexity [MCC96].....	48
Figure 8.	Threshold Values.....	50
Figure 9.	Binary and Multi-way Decisions [MCC96].....	51
Figure 10.	Planar Control Flow Graph With Regions [MCC96].....	52
Figure 11.	Complexity Plug-in Design and Implementation Scores.....	54
Figure 12.	Graph Window - Set View Options.....	56
Figure 13.	List Options Menu for Imagix 4D Database.....	57
Figure 14.	Graph Window and List Window.....	58
Figure 15.	Graph Symbols Key.....	59
Figure 16.	File Browser.....	61
Figure 17.	Flow Chart and Symbols.....	62
Figure 18.	File Editor.....	63
Figure 19.	Typical Report – Function Information.....	64
Figure 20.	Find Query in Browse Mode.....	65
Figure 21.	Grep Query in All Modes.....	66
Figure 22.	Find Query Options in Explore and Analyze Modes.....	67
Figure 23.	Add Function in Explore and Analyze Modes.....	68
Figure 24.	Analyze Mode - Software Complexity Analysis.....	69
Figure 25.	Threshold Complexity Level Setting.....	69
Figure 26.	Control Flow Analysis.....	70
Figure 27.	Linux Kernel – Function Hierarchy (Compact View).....	86
Figure 28.	OpenBSD Kernel – Function Hierarchy (Compact View).....	87
Figure 29.	Talisker Kernel – Function Hierarchy (Compact View).....	88
Figure 30.	Linux Scheduling – Function Hierarchy (Compact View).....	90
Figure 31.	OpenBSD Scheduling – Function Hierarchy (Compact View).....	90
Figure 32.	Talisker Scheduling – Function Hierarchy (Compact View).....	91
Figure 33.	Linux Operating System Layering.....	92
Figure 34.	OpenBSD Operating System Layering.....	93
Figure 35.	Talisker Operating System Layering.....	94
Figure 36.	Linux Kernel Layering.....	95
Figure 37.	OpenBSD Kernel Layering.....	95
Figure 38.	Talisker Kernel Layering.....	96
Figure 39.	Linux Operating System – File Summary Report.....	97
Figure 40.	OpenBSD Operating System – File Summary Report.....	97
Figure 41.	Talisker Operating System – File Summary Report.....	97

Figure 42.	Linux Kernel – File Summary Report.....	98
Figure 43.	OpenBSD Kernel – File Summary Report.....	98
Figure 44.	Talisker Kernel – File Summary Report.	98
Figure 45.	Linux OS – Function Information Sorted By Complexity.....	100
Figure 46.	OpenBSD OS – Function Information Sorted By Complexity.....	100
Figure 47.	Talisker OS – Function Information Sorted By Complexity.	101
Figure 48.	Linux Kernel – Function Information Sorted By Complexity.....	102
Figure 49.	OpenBSD Kernel – Function Information Sorted By Complexity.	102
Figure 50.	Talisker Kernel – Function Information Sorted By Complexity.....	103
Figure 51.	Linux Kernel Schedule Module – Function Information Report.....	103
Figure 52.	OpenBSD Schedule Module – Function Information Report.....	104
Figure 53.	Talisker Kernel Schedule Module – Function Information Report.....	104
Figure 54.	Linux Kernel – Global Variables.	106
Figure 55.	OpenBSD Kernel – Global Variables.	107
Figure 56.	Talisker Kernel – Global Variables.....	108
Figure 57.	Linux Kernel Schedule Module – Global Variables.....	109
Figure 58.	OpenBSD Kernel Schedule Module – Global Variables.	110
Figure 59.	Talisker Kernel Schedule Module – Global Variables.....	111

LIST OF TABLES

Table 1.	Linux 2.4.17 Source Directories – X86 Architecture.....	82
Table 2.	OpenBSD 2.9 Source Directories – X86 Architecture.....	82
Table 3.	Talisker Source Directories – X86 Architecture.	82
Table 4.	Linux Kernel Source Files – All Platforms.....	83
Table 5.	Linux Kernel Source Files – X86 Architecture.....	83
Table 6.	OpenBSD Kernel Source Files.....	83
Table 7.	Talisker Kernel Source Files.	84
Table 8.	Operating System Layers and Dependencies.	112
Table 9.	Kernel Layers and Dependencies.	113
Table 10.	Summary of Operating System Modules and Functions.....	114
Table 11.	Summary of Kernel Modules and Functions.....	114
Table 12.	Summary of Schedule Module Functions and Callers.	115
Table 13.	Operating System Global Variable Summary.	116
Table 14.	Kernel Global Variable Summary.....	117
Table 15.	Schedule Module Global Variable Summary.....	118
Table 16.	Operating System Complexity Distribution.	119
Table 17.	Linux Kernel Complexity Distribution.	120
Table 18.	Schedule Module Dependencies and Complexity Summary.	120
Table 19.	Operating System - Lines of Code Complexity Summary.....	121
Table 20.	Kernel - Lines of Code Complexity Summary.....	122
Table 21.	Schedule Module - Lines of Code Complexity Summary.	122
Table 22.	Summary of Comparative Analysis.	124

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PURPOSE

Security engineering requires a combination of features and assurance to provide confidence that security policy is correctly enforced. Rigorous engineering principles are applicable across a broad range of systems.

The purpose of this study is to analyze and compare three operating systems, including two general-purpose operating systems (Linux and OpenBSD) and a commercially available, embedded operating system (Talisker). The basis for the comparison considers secure software design principles, such as information hiding, hierarchical structuring, and modularity, as well as software complexity metrics, such as the McCabe Cyclomatic Complexity and the number-of-lines-of-code. In this analysis, we use a reverse engineering tool to show how the three operating systems compare to each other with respect to the qualities of a secure operating system design.

B. MOTIVATION

The questions that continuously challenge and trouble the typical desktop owners, using their personal computers for records and data storage, as well as government agencies with a vested interest in information security, is what flaws that have yet to be discovered are resident in their system, and how will those flaws affect the end user? Is it possible that undocumented functionality, whether harmless or malicious, exists inside the source code? It has been noted that contemporary systems, in general, provide little protection against accidental violation of operating system safeguards, let alone any protection against deliberate attempts to gain control of the operating system. This is unacceptable. There needs to be some type of defined controls or limitations enforced by the operating system to protect each user from all other users. Additionally, as the Trusted Computer Security Evaluation Criteria [DOD85] states in its accountability control objective, systems used to process or handle classified or other sensitive information must assure individual accountability whenever either a mandatory or discretionary security policy is invoked.

The steps to building a secure system involve creating a model or ideal description of a secure system from its uses, environment, and threats; developing the

security portion of the kernel to handle access control and authorization mechanisms; and then integrating the kernel with the rest of the system. It is the omission of this design for secure operation and the lack of strict control of programs in execution that characterizes most contemporary systems, and which in combination with the size and complexity of the systems makes it impossible to conduct meaningful testing or certification to determine that the systems are secure. Any type of methodical design is apparently undesirable in the commercial arena because products so developed take too much time to get to market. In addition, these methods require a higher intellectual caliber than that required by the customary commercial approaches.

Nevertheless, some basic guidelines in secure operating system design need to be established and shared with the commercial and open operating system worlds. Thus, the use of good hierarchical structuring, accurate problem decomposition and modular programming, and the proper design of interfaces should be applied to all system designs.

C. EXPECTED BENEFITS OF RESEARCH

By providing a comparative analysis between several systems, political and military decision makers will be able to see what characteristics of a secure operating system are desired, and how the systems available today fare against these qualities as well as each other. With the threats in the computer security problem evolving in an ever growing domain of users world-wide, the government and the military need to consider the benefits of building a secure operating system and demand that commercial-off-the-shelf vendors meet a certain standard of assurance using independent code audits. This research will provide a foundation for understanding the complexity that is found in commercial operating systems as well as their lack of secure software design principles.

D. RESEARCH OBJECTIVES

The principal, analytical approach used in this thesis is that of reverse engineering. Reverse engineering starts with an existing system, analyzing it to identify its components, their behavior, and the interrelationships among components. A principal benefit of reverse engineering is the discovery of useful information and structures, such as reusable data models, control structures, interface descriptors, design, behavior properties, function and performance requirements, data structures, algorithms, and architecture. In this research, we plan to utilize the Imagix 4D reverse-engineering

and documentation tool to provide a fair comparative analysis of the Linux, OpenBSD, and Talisker operating systems.

E. THESIS ORGANIZATION

This thesis consists of seven chapters. Chapter II provides an overview of secure operating system characteristics. Chapter III discusses the methodology for building secure operating systems using trusted system design elements and software design principles. A case study of the Multics Project is then presented to review the lessons learned from retrofitting security in an existing system. Chapter IV describes the management of complexity in operating system development and introduces the McCabe Cyclomatic Complexity metric. Chapter V reviews the Imagix 4D reverse-engineering and documentation tool that was used in the comparative analysis. Chapter VI examines the Linux, OpenBSD, and Talisker operating systems with the Imagix 4D tool, shows the data that was gathered with illustrations, and provides the results of the comparative analysis. Chapter VII concludes the thesis with a review of the findings and suggestions for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. OVERVIEW OF SECURE OPERATING SYSTEMS

A. INTRODUCTION

When computers were first used for automated data processing, few considered the importance of information security outside the realm of physical security. If the system designer never addressed parameter checking to protect the user from crashing his own system, then why would he consider classes of attacks where the user may be playing a malicious role? Security specialists, such as James Anderson [AND72], have found it useful to place potential security violations into three different categories: unauthorized information release, unauthorized information modification, and unauthorized denial of service. Today, security professionals refer to these three categories simply as confidentiality, integrity, and availability respectively.

To develop a secure operating system that addresses the protection of information from potential security breaches, the computer security problem must be well understood. In this chapter, an assessment of the threats to security will be given, including a classification of attackers followed by a set of techniques whereby computer misuse results in the unauthorized disclosure or modification of information. After providing the motivation for protection in operating systems, the reference monitor concept and the notion of a trusted computing base will be introduced. These concepts coupled together will provide a set of principles that can be applied both to the selection and design of security features, the developmental assurance required to implement a secure system, and how the mechanization of the security policies can be implemented as a subset of the operating system.

B. ASSESSMENT OF THREATS TO SECURITY

Undertaken by the Air Force, the Advanced Research Projects Agency (ARPA), and other defense agencies, a study was conducted to provide a solution for the technical problems associated with controlling the flow of information in resource and information sharing computer systems [AND72]. The resulting technical report, commonly called the “Anderson Report,” described an architectural framework for handling mediation of access in the face of potentially hostile users. It additionally noted that contemporary systems, in general, provided little protection against accidental violation of the operating

system, let alone any protection against deliberate attempts to gain control of the operating system. This was unacceptable. There needed to be some type of defined controls or limitations enforced by the operating system to protect each user from all other users. Due to the problems in hardware and operating system design at the time, the ad hoc programs developed to mediate between applications and operating systems were significant in size and were only capable of providing a degree of security in a non-malicious environment, but they did not provide adequate protection to defend against a hostile programmer.

One attempt to address security flaws in operating system and application design is appropriately named the “penetrate and patch” approach. Using this methodology, a “tiger team” of testers, code auditors, and hostile programmers are tasked to test the protection offered by an operating system, exploiting implementation errors and design oversights. Another team of designers then patches these flaws and vulnerabilities, after which the tiger team will again attempt to penetrate the operating system. Anderson [AND72] commented in his report that any attempts to “patch” an off-the-shelf system for security tended to obscure penetration routes, but had little impact on the underlying security problems. His conclusion was that if a system were designed to include many central privileged functions, it would inevitably become quite large and capable of concealing numerous flaws. These are the system flaws in particular that the malicious user tries to find and exploit in the pursuit of gaining supervisory control, commonly called “administrator level privileges” of the system.

Once supervisory control is obtained, the malicious user now “owns the box” and can reference any data or programs in the system. As illustrated by Brinkley and Schell [BRI93], the threat of malicious users does not end with just the exploitation of errors in the operating system code. Subversion, which can occur at any time in the life cycle of a system, can result in the exploitation of clandestine mechanisms called *artifices* that are constructed and inserted into a system to circumvent normal control or protection features. The most common types of artifices are known as *trap doors*, which permits the attacker to bypass internal controls at a later time. To support the argument that the DoD needs to employ secure operating systems for sufficient protection against these subversive threats, some recent examples of possible threats will be reviewed.

The questions that continuously challenge and trouble the typical desktop owners, using their personal computers for records and data storage, as well as several government agencies with a vested interest in information security, is what flaws that have yet to be discovered are resident in their system, and how will those flaws affect the end user? Is it possible that undocumented functionality, whether harmless or malicious, exists inside the source code? Involving the popular software-distributor Microsoft, recent examples of possible subversive threats have been documented in the media. From January 30-31, 2001, someone posing as a Microsoft employee tricked VeriSign, which hands out digital certificates, into issuing two certificates in Microsoft's name [LEM01]. Empowered with these Verisign-issued certificates, a malicious user could post a virus on the Internet that would appear to be from Microsoft, but might actually wipe an unsuspecting user's hard drive, for example. On Saturday, October 14, 2001, hackers gained access to Microsoft's internal network system and roamed source code files and other high-level secrets for approximately 12 days to five weeks [BBC01]. If the hackers had managed to access the source code of a Microsoft program already on the market, they would have been able to distribute versions of the product that looked legitimate but contained security holes or viruses. After the unfortunate events of September 11, 2001, when commercial airliners piloted by hijacking terrorists struck the World Trade Center and the Pentagon, a suspected member of the Al Qaeda terrorist network claimed that Islamic militants infiltrated Microsoft and sabotaged the company's Windows XP operating system [MCW01]. These and other examples have motivated computer scientists, information technology managers, and the general public to rethink what should be more important in an operating system design – security or functionality.

Profiles of the different adversaries will be reviewed to build an understanding of the threat environment in which a computer system resides. Ross Anderson [AND96] categorized hostile attackers in three brief bullets:

- Class I (Clever Outsiders): These individuals are often very intelligent but may have little knowledge of the system internals. They only have access to moderately sophisticated equipment and often take advantage of known flaws.

- Class II (Knowledgeable Insiders): These may be employees that have substantial specialized technical education and experience, have varying degrees of understanding certain parts of the system, but potentially have unlimited access. They often have sophisticated tools and instruments for analysis.
- Class III (Funded Organizations): These groups are able to assemble teams of specialists with related and complementary skills backed by great funding and resources. They are capable of in-depth analysis, designing sophisticated attacks, and using advanced analysis tools.

This taxonomy gives a backdrop to the flavor of malevolent individuals and organizations that exist in the world today. These adversaries may utilize various misuse techniques as presented by Brinkley and Schell [BRI93], including (1) human error, (2) user abuse of authority, (3) direct probing, (4) probing with malicious software, (5) direct penetration, and (6) subversion of security mechanism. Common to these six techniques are two elements. First, a *flaw* or vulnerability in a computer system exists in either the design or implementation. And second, the possible *threat* of attack exists by an authorized or unauthorized user (or agent) of the system attempting to exploit the flaw. The product of these two factors is equivalent to the amount of risk a user needs to consider when operating a system.

We will examine these misuse techniques and their related countermeasures. The first of these is *human error*, which happens to be probabilistic in nature. This technique may involve a combination of human, hardware, and timing factors that could allow unauthorized access to information. Better human-computer interactive design should provide for a simple user interface to reduce the probability of human error.

User abuse of authority, also known as the insider threat, is the second misuse technique. An example of this is a bank teller performing an illegal transaction. This authorized individual abuses the trust and authority granted in order to perform an irresponsible act for personal gain. To prevent this behavior, the system should confine the authorized users to their own domains of authority using techniques such as identification and authentication, and should audit their access requests.

Guessing passwords or searching for readable files are examples of *direct probing*, the next class of techniques to protect against. In this category, individuals try to use computer systems in ways not intended by the system designers or owners. Even though it is unethical to probe through another machine's file directory, nothing prevents or disallows a user from doing so. Correct system configuration and fail-safe defaults, which bases access decisions on permission rather than exclusion, are some examples of preventive measures to help defend against this class of misuse.

Probing with malicious software is similar to direct probing in that it is allowed but unethical; however, in this technique attackers employ specially developed software for the express purpose of carrying out the probing for them. A *Trojan Horse*, which contains overt functionality attractive to an unsuspecting user while performing covert functionality that has illegitimate side effects unknown to the user, is an example of such malicious software. There are several types of Trojan Horses in existence, including (1) viruses, which are pieces of self-replicating code that attach themselves to other programs in order to be executed, (2) time bombs, which are Trojan Horses set to trigger at a particular time, (3) logic bombs, which are Trojan Horses set to trigger upon the occurrence of a particular logical event, and (4) worms, which, like viruses self-replicate, but spread from system to system on their own exploiting system vulnerabilities. To prevent these occurrences, enforcement of non-discretionary policies and strong life cycle assurance for systems is required.

In the case of bypassing intended security controls, *direct penetration* is one of the more difficult techniques to protect against. Penetration is defined as the exploitation of a flaw using an attack to achieve an objective (compromise confidentiality, integrity, or denial of service). In many cases, one simply needs to find a single flaw in the implementation of the operating system or hardware and write a program to take control of the entire computer system. For protection, formal methods and good software engineering need to be applied to rid the system of flaws. This is where time and cost begin to grow significantly, adding difficulty to the design and construction of a secure operating system that is safe from direct penetration. Providing assurance that the operating system is implemented correctly and operates in accordance with policy is exercised throughout the management of the system lifecycle.

Subversion of security mechanism, the last of the six techniques, involves the covert and methodical undermining of internal system controls to allow unauthorized and undetected access to information within the computer system. An example of this is the insertion of an undocumented feature, like an Easter egg (www.eeggs.com), on a machine during the manufacturing process. The major concern of this technique is that subversion is possible in any activity of the system life cycle, including design, implementation, distribution, installation, and use. For protection, controls must be built to specifications in a non-malicious environment and be simple and small enough to verify correct behavior when subjected to testing and evaluation.

From the TCSEC definition [DOD85], protection must be defined from the start in terms of the perceived threats, risks, and goals of an organization. With a review of the threats to security completed, it is necessary to elaborate on protection to give a flavor of what is possible in the design of a secure operating system.

C. PROTECTION IN OPERATING SYSTEMS

In 1971, Butler Lampson [LAM71] defined “protection” as a general term for all of the mechanisms that control the access of a program to system resources. The original motivation for putting such mechanisms into computer systems was to keep one user’s malice or error from harming others by destroying or modifying data, reading or copying data without permission, or degrading the service of another user.

Saltzer and Schroeder [SAL75] suggested a division of protection schemes according to functional properties into five categories: unprotected, all-or-nothing, controlled sharing, user-programmed controlled sharing, and putting attributes on information. The *unprotected* systems have no mechanisms for preventing a determined user from having access to every piece of data stored inside the machine. *All-or-nothing* systems provide for the isolation of users. Total sharing of some information, which is available to all, sometimes moderates this type of system. *Controlled sharing* systems, which are significantly more complex, control explicitly who may access each data item resident in the system. An Access Control List (ACL) is an example of this type of system. *User-programmed control sharing* systems are used in the case that access may be restricted to a file in such a way not provided by the controlled sharing operations.

Gates, time of day, and “two person control” are all examples of this type. The systems that put *attributes on information* are able to maintain some control over the use of label-based information. In all these protection schemes, maintaining confidentiality of information with access control mechanisms is the objective.

The access control model for confidentiality was first introduced by Lampson [LAM71] and was refined by Graham and Denning [GRA72]. Harrison, Ruzzo, and Ullman [HAR76] later used Lampson’s concept of an access control model to analyze the complexity of determining the effects of a particular access control policy. Their conclusions showed that the problem of building a general-purpose algorithm to determine whether an arbitrary program, comprised of simple primitives for updating an access matrix, leaks an access right is undecidable. Therefore, the fundamental result of the *HRU Model* is that one cannot construct an algorithm to examine all arbitrary systems for leakage of access rights.

Bell and LaPadula’s [BEL73] formal policy model included mandatory access control. The model introduced the additions of labels for subjects and objects and the *simple security policy* and the **-property*. These two properties for confidentiality provided two rules for information flow between subjects and objects employing labels: (1) no read up (simple security policy) and (2) no write down (**-property* or confinement property). The first rule mimics the paper world where users can only view data they are authorized to see, and the second rule prevents Trojan Horse programs from copying sensitive data to files that are accessible by unauthorized users.

To assist with the design of a system, a model can provide a coherent and consistent expression of the policy. But before the implementation of an operating system can be described, a review of an abstract model, the reference monitor concept, is required to understand the necessary and sufficient criteria for a system to provide confidence of policy enforcement in the face of malicious threat.

D. THE REFERENCE MONITOR CONCEPT

The *Reference Monitor Concept*, which emerged as part of the Anderson Report in 1972, provides a description of an ideal mechanism representing the necessary and sufficient functions to enforce a policy to control access by subjects to objects. A subject

is defined as an active entity that causes information to flow among objects or changes system state. In comparison, an object is a passive entity that contains or receives information. Access to an object implies access to the information it contains. Hence, the reference monitor is an abstraction that controls all access or the collection of access controls applied to objects like devices, files, memory, directories, and interprocess communications [PFL97]. The process is simply a program in execution characterized by a single current execution point, represented by the machine state, and an address space. A “domain” is the set of objects that a subject has the ability to access [DOD85].

To protect the memory of each domain, an implementation technique called protection domains partitions the memory into more privileged and less privileged regions. Programs in lesser-privileged domains cannot arbitrarily read and write files or execute programs in more privileged domains. Figure 1, which illustrates two separate protection domains, demonstrates this concept. When the process calls an operating system function, it changes from the user subject to the operating system subject of the process. Since the operating system is a more privileged domain, the user programs cannot arbitrarily read or write databases or execute code in the operating system domain.

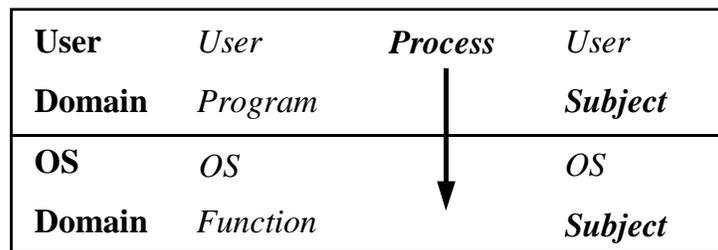


Figure 1. User and Operating System Protection Domains.

Within this system model, everything belongs to some process and cannot be accessed by any other process other than its owner. The user program can, however, call into the higher domain in a controlled way, for example, using a gate or trap mechanism that allows lesser-privileged subjects to invoke services of greater privileged subjects as shown in Figure 2. Saltzer and Schroeder [SAL75] proposed this concept of protection in computer systems with their three requirements for kernel isolation, including (1) a privileged state bit, (2) the partition of protected memory, and (3) a mechanism that authorizes transfer of control from an unprotected level to one that is protected. In an

Intel X86 processor, the features that are available to support these respective requirements include a protection bit, descriptor table segmentation, and four rings of hardware privilege levels.

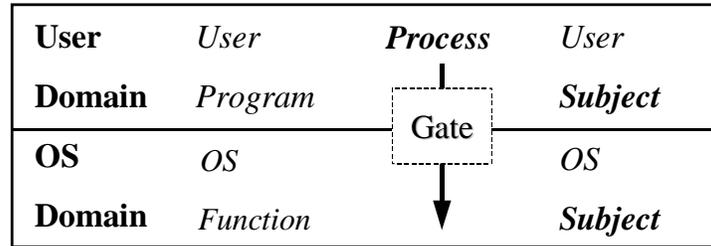


Figure 2. Gate Calls in Protection Domains.

Ring brackets are a software implementation of protection domains, where each object has a “bracket” of values that specifies what domains can access the object. An example ring brackets policy, utilizing the Intel privilege levels (PL₀-PL₃) built into the hardware, could be [R₁, R₂, R₃], where each R_n is assigned a value from {PL₀...PL₃}. In this instance, PL₀-R₁ is the write bracket (viz, subjects PL₀ through the value in R₁ may write to the associated object), PL₀-R₂ is the read bracket, and PL₀-R₃ is the execute bracket. An example of such an implementation can be found in GEMSOS [FER95]. The abstract reference monitor concept can mediate the access requests between subjects and objects according to such a ring construct.

The implementation of the reference monitor has been called the *reference validation mechanism*, a combination of hardware and software [AND72]. The *current access authorizations*, graphically modeled as the current access matrix, represent who has access to what and with what modes and rights. This can be abstractly described using a set of tuples indicating that a subject has current access to an object with a particular mode, $\langle \text{subject}, \text{object}, \text{access mode} \rangle$. To enforce policy, the mechanism requires that all subjects and objects have policy-relevant attributes associated with them and maintained in the *authorization database*, graphically modeled as the authorization matrix. Figure 3 depicts an abstract model of the reference monitor.

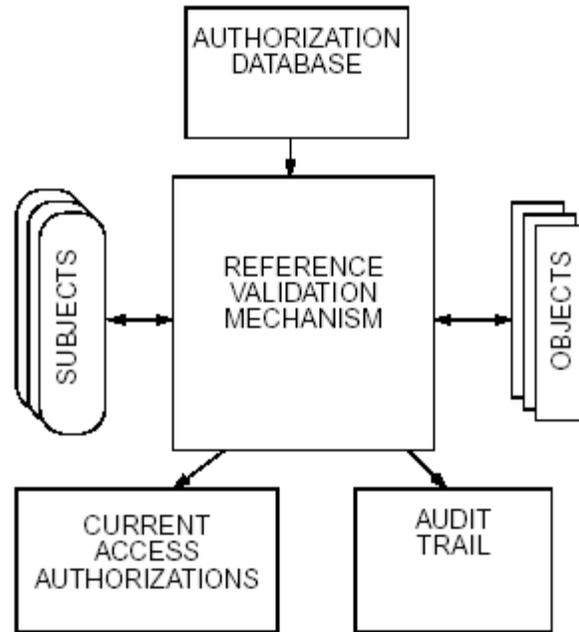


Figure 3. The Reference Monitor Concept.

From this abstract model, one can observe that the reference monitor enforces the authorized access relationships between the subjects and objects of a system. User identification, clearance levels, and current session levels could all be attached to a certain subject acting on a users behalf. An object, in comparison, could have access control lists, ring brackets, and sensitivity labels that contain information about who can access the object and what levels of clearance are required. Variations of an access control policy that is either identity-based or label-based can define the specific access control requirements.

To establish that the model's behavior complies with the requirements of an access control policy, a "basic security theorem" may be included in the model. For example, if it is shown that the current access authorizations (CAA) is a subset of the authorization database (AD) in the initial state and in every operation, and the AD is consistent with the access control policy, then we can conclude that the module enforces the policy. The practical utility of this model is that the CAA corresponds well with the descriptor-based hardware mechanisms, and it permits the basic security theorem to be expressed as an invariant. Access relationships of getting access and releasing access

between subjects and objects, as well as the reference relationships like reading and writing, define the abstract reference monitor functions.

These abstract access and reference functions are implemented by the reference validation mechanism, which constitutes the necessary and sufficient functions for enforcing a policy of controlled access to objects by subjects. By definition, it is required to be tamperproof, always invoked, and small enough to be subject to analysis and tests to assure that it is correct [AND72]. To be *tamperproof*, the mechanism needs to be self-protecting and must be constructed so that the mechanisms it relies on are at least as assured as it is. (This will be discussed later when the notion of hierarchical layering of system dependencies will be introduced.) To be *always invoked*, mediation of access to all resources needs to always be “on” and cannot be circumvented. And as defined earlier by necessary and sufficient, it needs to be *small enough* to ensure that accesses to objects are being mediated (viz, it is sufficient to the task), and nothing extra considered irrelevant to the requirement for access validation is included (viz, only the necessary functions are included).

Relative to the characteristics discussed above, the crucial role of the reference monitor in enforcing security means that it must function correctly. But because the likelihood of correct behavior decreases as the complexity and size of a program increase, the best assurance of correct policy enforcement is a small, simple, understandable implementation [MYE80]. Therefore, to achieve assurance that the reference validation mechanism is correctly implemented, minimization and rigorous software engineering is required. This topic will be discussed later in the following chapter.

It is important to remember that the mechanism described above is not a model of secure computing. Rather it is a device to provide containment of programs in execution [AND72]. Therefore, since the reference monitor only represents the access control mechanism, elements of the system for supporting the security policy also need to be included in the system implementation, including (1) identification and authentication, (2) security administrator interfaces, and (3) audit retrieval and analysis functions. With all of these supporting functions along with the reference validation mechanism, one

could integrate all of the system security controls into a portion of the operating system code. This leads us to the notion of a trusted computing base.

E. THE TRUSTED COMPUTING BASE

The National Institute of Standards and Technology (NIST), defines the *trusted computing base* (TCB) as the set of all protection mechanisms in a computing system, including the hardware, firmware, and software, which together enforce a unified security policy over a product or system [NIS91]. It contains all the elements of a system for supporting the security policy and for the support of the isolation of objects upon which protection is based.

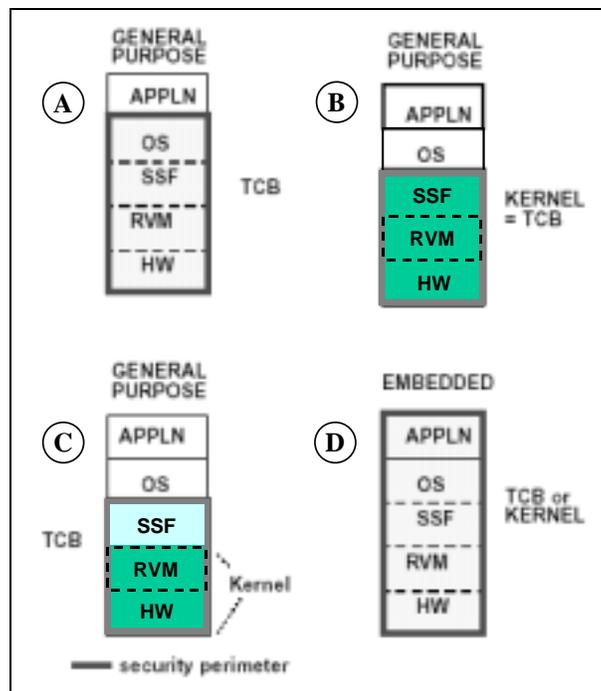


Figure 4. Sample Monolithic TCB Architectures.

Figure 4 depicts sample monolithic TCB architectures. The hash marks drawn around portions of the system delineate the security perimeter of the trusted computing base. This perimeter includes all of the elements of the system for supporting the security policy and for the support of the isolation of objects upon which protection is based. For instance, the TCB in general-purpose system A of Figure 4 includes the operating system, the security support functions (SSF), such as identification, authentication, and auditing, the reference validation mechanism, and the hardware, while the systems in parts B and C

exclude the general-purpose operating system functions from the TCB perimeter. In comparison, the embedded system in part D may include part of the application layer, thereby enclosing the entire system within the TCB perimeter. In this embedded system example, the TCB is also called the kernel like the system observed in part B. The kernel can also be a subset of the TCB, as demonstrated in system C. However, unlike any of the architectures shown, the TCB term can include systems that do not contain a reference validation mechanism or a security kernel. A TCB always refers to an implemented system for a computer.

As defined in the Trusted Computer System Evaluation Criteria [DOD85], there are a certain number of architectural requirements that need to be met by the TCB in order for it to have a high level of assurance (i.e., level B3). The TCSEC requires that the TCB shall:

- Maintain a domain for its own execution that protects it from external interference or tampering.
- Maintain process isolation through the provision of distinct address spaces under its control.
- Be internally structured into well-defined independent modules.
- Structure modules such that the principle of least privilege is enforced.
- Identify the correct functionality of the TCB interface and elements.
- Incorporate significant use of layering, abstraction, and data hiding.
- Employ proper system engineering to minimize complexity and exclude modules that are not protection critical.

F. CONCLUSION

From our discussion, we have noted that the environment to be protected must be well understood. After an assessment of the threats to security is reviewed, a system must then be designed to provide the desired protection. A protection scheme may include a security kernel that is responsible for enforcing the security policy. This security kernel is an implementation of an abstract concept called the reference monitor that controls the accesses to objects. To encompass all of the parts of a trusted operating

system on which we depend for the correct enforcement of a security policy, the notion of a trusted computing base and its security perimeter were introduced. In the next chapter, the security policy and model, design principles for secure operating systems, the security kernel concept, and criteria for evaluation and assurance will be reviewed to demonstrate a proper approach in building a secure operating system. A case study from the Multics experience will also give examples of previous efforts from which we can learn.

III. BUILDING A SECURE OPERATING SYSTEM

A. INTRODUCTION

Ken Thompson [THO84] has commented that, by default, an application should not be trusted if the end user did not write the code himself. He further stated that no matter how much a posteriori source-level verification or scrutiny is completed, no amount will protect the user from untrusted code. He illustrated this point by describing an instance of an artifice first suggested by Karger and Schell [KAR74]. Thompson claimed to have introduced an artifice into the C compiler as subverted code. When this code in the compiler determined that it was compiling the “login” code for the Unix operating system, it would generate code to accept not only the valid password but also a fixed password he had previously selected and built into the artifice or trapdoor. Thompson then planted code in the compiler, which in turn added the code for the two artifices into the object code each time it recompiled subsequent versions of the compiler, removing the subverted code from the compiler source code altogether. With this program, Thompson gave a classic example of how to construct clandestine mechanisms that can be introduced into the software during the construction phase of the system life cycle. Previous efforts have demonstrated that it is possible to design, construct, and maintain a secure operating system throughout its life cycle to keep it free of such subversive threats. The XTS 200 and XTS 300 [FER92] and GEMSOS [FER95] are examples of these efforts.

In the Anderson Report [AND72], the steps to build a secure system involved creating a model or ideal description of a secure system followed by the development of the security portion of the system or the security kernel, which would later be integrated with the rest of the system. This task, however, is not as easy as it may sound. The design of a trusted system is a difficult process, involving the selection of the appropriate and consistent set of features together with an appropriate degree of assurance that the features have been assembled and implemented correctly.

B. THE SECURITY POLICY AND MODEL

1. The Policy

The Trusted Computer System Evaluation Criteria defined the security policy control objective as follows: “A statement of intent with regard to control over access to and dissemination of information, to be known as the security policy must be precisely defined and implemented for each system that is used to process sensitive information. The security policy must accurately reflect the laws, regulations, and general policies from which it is derived” [DOD85]. Hence, the security policy is an informal set of well defined, consistent, and implementable rules oriented towards an organization and their policies.

Categories and subcategories of different policies to control the access and dissemination of information are discussed at length in computer security texts. The major categories include the mandatory access control policy (MAC), the discretionary access control policy (DAC), and supporting policies. The subdivisions of categories are related to the further control of information with respect to secrecy and integrity.

2. The Model

Once the policy has been established, the organization designing the system needs a formal statement, oriented towards implementation, of what is meant by a secure system. This policy model provides for a number of advantages for the design, including (1) *completeness* by including all relevant assertions, (2) *clearness* by making the statement understandable and brief, and (3) *correctness* by ensuring the statement is accurate. The model also supports assurance by providing *constraints* on system behavior, *consistency* with no contradictions, and a *basis for validation*.

After constructing the model, the designer must study the different ways that the security can be enforced in the system. This will assist the designer in building confidence that the proposed system will meet its requirements prior to actually creating the trusted operating system. This high level of confidence is the objective when building a high assurance system.

C. TRUSTED SYSTEM DESIGN ELEMENTS

Prior to discussing the design of a trusted operating system, it would be beneficial to review the eight design principles that have been provided from the experiences of others who invested considerable time and energy in system research. These eight were first discussed by Saltzer and Schroeder [SAL75] and are as follows: economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability. The economy of mechanism and least common mechanism are more central to the theme of this thesis, but a brief discussion of all of these will be given here to preserve continuity and provide context.

In setting the *fail-safe defaults*, the designer, acting conservatively in the interest of the user, needs to identify what items should be accessible rather than those that should not. Therefore, the default condition should be denial of access.

To obtain *complete mediation*, every access attempt must be checked. This means that the mechanism for checking these access attempts needs to be positioned correctly so it can mediate direct access attempts. It must be designed so that attempts to circumvent the system will fail.

An *open design* simply means that the design should not be secret. With the exception of a few key items like password tables, the source code should be open to extensive public scrutiny to help provide criticism, new ideas, and independent confirmation of the security design. In the context of commercial systems, where protection of intellectual property is of great importance to the vendor, an independent evaluation with a memorandum of understanding could be established between the company and the reviewer.

Separation of privilege is common in the commercial business world and makes good sense here in the design of secure systems. Ideally, access to objects should depend on more than just an <identification, password> pair; for some systems, this could mean two person access control where both users need to enter keys, or for others, the user is required to have an <identification, password> pair accompanied by a token with a cryptographic key. In this way, if one protection system is defeated (i.e., the password

has been cracked or discovered using social engineering), the attacker will not have complete access. The “Defense in Depth” concept, which combines the capabilities of people, operations, and security technologies like firewalls, intrusion detection systems, and public key cryptography to establish multiple layers of protection, could be viewed as an instantiation of this separation of privilege principle.

Least privilege means that every program and every user of the system should operate using the fewest privileges possible. This helps minimize the extent of the damage caused by a malicious or inadvertent attack.

The notion of *psychological acceptability* relates to the ease of use issues that all software engineers contend with in human computer interface design. If the mechanism for security is not easy to use (i.e., users routinely and automatically apply the protection mechanisms), then it is more likely that the user would apply them incorrectly or not at all.

Use of *least common mechanism* will be discussed later in the Multics case study to demonstrate the importance of minimizing the amount of mechanism common to more than one user and depended on by all users, especially in the design of the security kernel. Since shared mechanisms provide for potential channels of information flow, systems should employ physical or logical separations to reduce the risk of sharing.

Economy of mechanism, which will be discussed later in Chapter 4, delineates that the design of a protection system should be simple, small as possible, and straightforward. This will enable the protection system to be more carefully analyzed, exhaustively tested, verified, and relied upon with a degree of confidence or assurance -- one of the characteristics of the reference monitor concept that we are striving to achieve.

Keeping these principles in mind can guide the design and contribute to an implementation that has no security flaws [SAL75]. Together with the proper software design principles that will be discussed next, the security engineer will have the appropriate tools to build and maintain a secure system that spans the system’s lifecycle.

D. SECURE SOFTWARE DESIGN PRINCIPLES

The system lifecycle that is taken into consideration by the security engineer includes the following steps: requirements engineering or specification, design, implementation, testing, documentation, configuration management, system distribution, and system maintenance [DOD85]. After completing the requirements specification, the software design needs to follow a systematic approach to identify the major components of the system, specifying what each component does and establishing the interfaces between these components. Modularity is one principle of software engineering that makes this approach possible.

1. Modular Programming

When commenting on proper system design, Gauthier and Pont [GAU70] have stated that a well-defined segmentation of the project effort ensures system modularity. This means that each task has its own separate and distinct program module, which is tested and maintained independently. With no confusion in the intended interface with other system modules, this approach helps the designer construct well-defined inputs and outputs at implementation time.

A module that has been well designed encapsulates a function or database and provides an interface that is always called upon by other modules for access to those functions or databases. In a sense, the module is a “database” containing entity that holds different types of information. To access this database, three types of interfaces can be defined: those that initialize the data, those that modify the data, and those that list the data. Therefore, when a software engineer decides to build a new system using this approach, the designer needs to define the problem, define the databases, and then decompose these into modules. This decomposition supports the understanding, analysis, and maintenance of the interfaces and can provide a basis for least privilege. As an added benefit, modular programming uses coding techniques that allow modules to be written with little to no knowledge of the code in other modules; this allows the modules to be replaced or reassembled without having any effect on the system as a whole. This is where modular programming becomes of value as it is applied to the production of large, complex systems such as operating systems.

Modular programming also provides some benefits to the software management team as illustrated by Parnas [PAR72]. From a managerial standpoint, the development time in producing a large system is shortened since separate groups of programmers can be assigned their own modules to work on with little to no communications with the others, and the interface of each module is well defined from the start. Another benefit is product flexibility. So long as the interface is stable, the module can undergo drastic internal changes without affecting the other modules. The last benefit, comprehensibility, goes hand in hand with the management of complexity that will be discussed later. In short, it should be possible for anyone to study the complete system one module at a time. This will help the person analyzing the system reach a better understanding of the system, its purpose and functionality, and its security. Layered design also aids in this task.

2. Hierarchical Structuring

To define a hierarchical structure, a partial ordering relation such as “uses” or “depends on” needs to be defined between the modules or programs [PAR72]. A partial ordering has the three properties of being 1) reflexive, 2) antisymmetric, and 3) transitive. For example, one can say that for a particular set of modules {x, y, and z} and the relation “depends on,” that these three conditions mean respectively that for all x, y, and z in the set: 1) *x “depends on” x*, 2) *x “depends on” y* and *y “depends on” x* implies *x = y*, and 3) *x “depends on” y* and *y “depends on” z* implies *x “depends on” z*.

The managerial, product flexibility, and comprehensibility benefits discussed earlier could all be obtained if the modules were all on the same level without partial ordering. However, if the modules are assembled on different levels, a partial ordering can be introduced giving two additional benefits to the system: 1) parts of the system are simplified because they use the services of the lower levels and 2) if the upper levels were cut off, the remaining lower levels are still considered as a usable and useful product [PAR72]. If the system were designed in such a way that the low level modules made use of the high level modules, no hierarchical structure would exist, making it very difficult to remove portions of the system and still have it be usable and useful. Another issue with this lack of hierarchical design is error state. Since higher layers “depend” on lower layers in a proper design, the inclusion of lower layers that “depend” on higher

layers creates a circular dependency problem that may loop indefinitely if an error is encountered. For instance, if an error occurs in the lower modules, this error will propagate into the high modules, which is again propagated to the lower layer. This creates an error state in which the system may never permit recovery. This concern was presented in the auditing facility of the VMM security kernel [SEI90].

The concepts of structuring and layering were also discussed by Dijkstra [DIJ68] who stated, "...the larger the project, the more essential the structuring!" Following the strict levels of abstraction approach he introduced, the T.H.E. system hierarchy was broken up into five layers. Since his system was small and straightforward, however, Dijkstra later commented that his layering decision might very well turn out to be of modest depth. Schroeder, Clark, and Saltzer [SCH77] in their attempt to rid Multics of dependency loops also decided to use levels of abstraction in their security kernel design as a means of reducing complexity and providing precise and understandable specifications. By making lower layers unaware of higher abstractions, the Multics kernel designers were able to reduce the total amount of interactions in the system and thereby reduce the overall complexity. In addition, these levels of abstraction simplified the correctness argument and kernel debugging since each layer could be tested in isolation from all higher layers.

3. Information Hiding

A protected subsystem, as defined by Saltzer and Schroeder [SAL75], is a collection of procedures and data objects that is encapsulated in a domain of its own. This encapsulation allows the data structure of the protected object to be interpretively accessed; for example, it does not allow its internal organization to be accessible except by the internal procedures of the protected subsystem, which may be called only through specifically designated, domain entry points. Parnas [PAR72] first introduced this idea of data hiding or information hiding as a design approach where the software is decomposed into modules that hide design decisions. Therefore, the attention shifts away from the code used to implement the module and concentrates more on the signature or interface of the module.

Using information hiding in module design, data can only be manipulated through a simple, high-level, well-defined interface. Consider the example of a file system that provides services to users and applications in the use of files. Typically, the only way that a user or application may access files is through the file management system. This relieves the user or programmer of the necessity of developing special-purpose software for each application and provides the system with a means of controlling its most important asset. The implementation details are hidden from the program. This concept could be applied to a kernel-based architecture, where the description should only include some of the kernel's properties, but the rest should remain non-public or secret.

E. THE SECURITY KERNEL

One notion of designing an operating system around a kernel is described by Lampson [LAM76]. A kernel, by definition, is the part of the operating system that performs the lowest-level functions. Today, in standard operating system design, the kernel implements operations such as synchronization, message passing, interprocess communication, and interrupt handling. In the case of a security kernel, the consistent application of the reference monitor concept prevents penetration or subversion of security mechanisms with a high degree of assurance [BRI93].

The reason behind kernel-based architectures is that since a kernel contains only that part of the system essential to meeting security requirements, it can be small, compared to the system as a whole, and therefore has a better chance of being correct. One of the main advantages of the kernel approach is the clear statement of purpose of the system. Since a kernel is meaningful only with respect to some explicit requirements, these requirements serve as the statement of purpose of the system. The other main advantage is the enhanced probability of correct operation. Since the programs that are critical to the correct operation of the system are isolated in the kernel, a great deal of attention can be paid to getting this code right, and less attention can be paid to other system code that may be important but is not critical. The relatively small size of the kernel significantly improves the chances of applying formal verification techniques freely to the programs in the kernel, where applying these techniques to the entire system would be difficult.

From the earlier discussions regarding the reference monitor concept, the security kernel is simply an implementation of the reference validation mechanism. Some of its functions include creating subjects and objects, performing low-level resource management (controlling access to system resources), virtualizing the hardware, enforcing policy on every access request, and protecting itself from tampering. Of the group of systems that have employed the security kernel concept, the SCOMP system was the first security kernel of its size and complexity to be formally verified, and the first to be evaluated against the Trusted Computer System Evaluation Criteria [BEN84]. Other security kernel implementations are found in the XTS [FER92] and GEMSOS [FER95].

F. EVALUATION

System evaluations need to be based on some set of criteria, and they need to be done by third party evaluators to remove any bias from the evaluation process. From Dijkstra's [DIJ68] experience, there exist only three stages in the building of a new system: *conception* to meet the specified requirements, *construction* to write the code, and *verification* to test all relevant states at each hierarchical level to provide for a flawless system. In the T.H.E.-Multiprogramming System, Dijkstra's [DIJ68] small team started at level 0, moving on to the next layer only after they were satisfied that the previous level had been thoroughly tested. This was done at each layer by forcing the system into all the different states, and, at the same time, verifying that the system reacted in accordance with the specification.

Taking from the work of Dijkstra as well as a number of others, evaluation criteria were formalized to help determine whether systems met a certain degree of assurance, intended to establish confidence that the systems as a whole are robust against attack. In reference to the highest level of assurance (i.e., A1), Brinkley and Shell [BRI93] conjectured that with the amount of verification and validation criteria required of an A1 system, a malicious programmer, who participates in any or all of the design, construction, distribution, and maintenance phases of the system, could never insert a trapdoor. The *Trusted Computer System Evaluation Criteria* (TCSEC) or the Orange Book and the *Common Criteria* (CC) are two examples of evaluation criteria that categorize these levels of assurance from highest to lowest.

The TCSEC (Orange Book) was developed by the National Security Agency (NSA) and adopted in 1985. Addressing secure operating system design, a list of functional requirements (security policy and accountability) as well as assurance requirements (assurance and documentation) were carefully developed and categorized by TCSEC [DOD85]. To specify the amount or level of assurance and functional requirements met by a system, seven classes (A1, B3, B2, B1, C2, C1, D) were defined as an evaluation measure, with class A1 meeting all requirements and class D meeting none. Common to every class except class D, which provides essentially no security, testing and analysis for obvious flaws that bypass system security is performed, expanding the rigor and intensity with each increase in level.

In class C, discretionary security protection and controlled access protection are the requirements for the respective C1 and C2 classes. To provide for *discretionary security* (C1), the systems must be capable of controlling access on an individual user basis. For *controlled access* (C2), the system must provide individual accountability through login procedures, auditing, and resource isolation. In addition, storage must be erased before being reassigned to another process.

With the next level incorporating all the criteria below it, class B systems require labeled security protection, structured protection, and security domains as per the criteria for the respective B1, B2, and B3 class systems. *Labeled security* (B1) is provided by systems that support the assignment of sensitivity (classification) labels to subjects and objects, and they must provide mandatory access controls. For *structured protection* (B2), the systems must have a TCB based on a clearly defined formal security policy model, and it must be structured into protection-critical and non-protection-critical elements. The system also must be subject to more stringent configuration management and trusted facility management, and must address covert channels and provide a trusted path between the user and the TCB for login authentication. To meet the criteria of *security domains* (B3), the TCB must mediate all accesses, be tamperproof, and be engineered for analysis and testing. Additionally, the system must incorporate significant use of layering, abstraction, and information hiding, auditing to signal alerts, and system recovery procedures.

Class A or A1 systems, the highest class that satisfies all the TCSEC criteria, requires *verified protection* to provide high assurance against subversion of the security mechanisms. This criterion stipulates that formal methods of specification and verification must be used throughout the entire design process.

After a careful review of the assurance requirements, one could note that many of these reflect the secure system design principles, including (1) system architecture (hierarchical structuring and information hiding), (2) security testing, (3) formal verification (use of security models), (4) trusted facility management (support for separation of operator privileges), (5) configuration management during development (code reviews before implementation into the baseline product), (6) trusted recovery (system returns to secure state after power failure), and (7) trusted distribution (hardware by trusted shippers and software by cryptographic checksums). As illustrated in Figure 5, these criteria are broken down, giving a total of seven levels of assurance. For each level, a symbol with an 'X' enclosed by a circle signifies that new requirements in a given criteria class must be satisfied to be in compliance with TCSEC. For instance, a lower class such as C1 need only satisfy a given subset of the system architecture criterion that is required of a higher class like C2. If an arrow is observed as an entry, the corresponding criteria required of the level below is the same for the given level in question. For example, the system integrity criteria that class C1 systems need to satisfy are the same for all higher TCSEC levels.

Criteria Class		D	C1	C2	B1	B2	B3	A1
Assurance	System Architecture		⊗	⊗	⊗	⊗	⊗	⇒
	System Integrity		⊗	⇒	⇒	⇒	⇒	⇒
	Security Testing		⊗	⊗	⊗	⊗	⊗	⊗
	Design Specification and Verification				⊗	⊗	⊗	⊗
	Covert Channel Analysis					⊗	⊗	⊗
	Trusted Facility Management					⊗	⊗	⇒
	Configuration Management					⊗	⇒	⊗
	Trusted Recovery						⊗	⇒
	Trusted Distribution							⊗

Figure 5. TCSEC Requirements for Assurance.

The Common Criteria [CC99], which was developed by an international community and adopted in 1998, has a similar approach to assurance requirements that

guarantees the hierarchy of requirement sets as displayed by TCSEC. In this evaluation method, a *Protection Profile* (PP), defined as a specific requirement set, is built from a “menu” of assurance requirements to fit the needs of the user. This menu has 58 primitive assurance requirements, which are called assurance components. These are then grouped into 22 families, which are in turn grouped into six classes, including (1) configuration management, (2) delivery and operation, (3) development, (4) guidance documents, (5) life support cycle, and (6) tests. To measure the amount of requirements met by a system, an evaluated assurance level (EAL1 – EAL7) is assigned. In Figure 6, the numbers under the EAL columns specify which components of the family are required. For example, EAL5 requires components 1 and 2 of the ate-cov (the assurance testing-coverage) family. These requirements serve as a guide for the development, procurement, and basis of evaluation of information technology with security features.

Assurance Classes	Assurance Families	EAL						
		1	2	3	4	5	6	7
Configuration Management	acm_aut				1	1	2	2
	acm_cap	1	2	3	4	4	5	5
	acm_scp			1	2	3	3	3
Tests	ate_cov		1	2	2	2	3	3
	ate_dpt			1	1	2	2	3
	ate_fun		1	1	1	1	2	2
	ate_ind	1	2	2	2	2	2	3

Figure 6. CC Evaluated Assurance Levels (EAL).

G. ASSURANCE

If an operating system has no assurance of correctness of policy enforcement or penetration resistance, can a user be held accountable for the unpreventable actions of software behaving maliciously? A user needs to have confidence that both the policy and accountability objectives are being met by the system. Hence, to establish that a protection mechanism is effective and self-protecting, one needs to check the accuracy, correctness, and completeness of the implementation [SAL75].

The Trusted Computer System Evaluation Criteria defines the assurance control objective as follows: “Systems that are used to process or handle classified or other sensitive information must be designed to guarantee correct and accurate interpretation of

the security policy and must not distort the intent of that policy. Assurance must be provided that correct implementation and operation of the policy exists throughout the system's lifecycle" [DOD85]. Providing the highest levels of assurance will reveal intentional flaws (trapdoors) that have been deliberately introduced into the operating system. Since it is undecidable to conduct searches of arbitrary code for such artifices of subversion that may be present somewhere in a million lines of code, the construction process must minimize this possibility to conclude that such artifices do not exist, while at the same time discovering and correcting accidental errors that could lead to system compromise.

From all of the topics introduced thus far in this chapter, the main takeaway is that to establish a level of assurance, the operating system needs to be implemented in such a way that the user has confidence that it will enforce the security policy, and it needs to meet the expectations of the user with a certain degree of trust. In the next section, a case study about the Multics Project will be reviewed to give an example of a successful attempt at building a secure operating system.

H. THE MULTICS EXPERIENCE

Saltzer and Schroeder [SAL75] have written that contemporary operating systems have a well-known tendency to be extraordinarily large and complex. Intuitively, this size and complexity make the process of building a secure system hard due to the negative nature of security requirements. These requirements, which are necessary and sufficient, demand that all unauthorized actions, whether accidental or malicious, need to be blocked (always invoked), the protection mechanisms in the design cannot be circumvented (tamperproof), and the implementation that reflects the security specification needs to be proven correct with a certain level of confidence (verifiable). The Multiplexed Information and Computing Service (Multics) Project team, which started as a cooperative effort among the Massachusetts Institute of Technology (MIT), the Bell Telephone Laboratories, and the Computer Department of General Electric later acquired by Honeywell [COR91], set out to tackle these issues as well as others in the pursuit of a secure, time-sharing system.

Parnas [PAR96] stated that if a group is given the opportunity to start a new project, their time spent studying previous efforts and identifying the reasons for their poor structure is likely to pay off in a far better, easier to maintain product. Instead of reinventing the wheel, it makes more sense to avoid the same mistakes made before and take heed of those solutions that others have found. Over the last 25 years, the lessons learned from the Multics experience have propagated throughout academia and government agencies affiliated with security and have come to be widely acknowledged as the proper way of building a secure system. To distribute the knowledge gained from Multics, members of the project team documented the overall experience in a series of research papers.

While working on the Multics Project at MIT, Schroeder [SCH75] discussed three primary goals focusing on the reduction of the size and complexity of security-relevant portions of the Multics system. First, the team wanted to identify a minimum mechanism that must be correct to guarantee computer enforcement of the desired information access constraints. Second, they wanted to simplify the structure of that mechanism to make the verification of its correctness auditable. And third, they wanted to demonstrate by test implementation that the security kernel developed is capable of supporting all the functionality required of the Multics system completely and efficiently. Later research would take a step deeper into the design aspects of the kernel.

Schroeder, Clark, and Saltzer [SAL77] later redefined the design proposals to again concentrate on size and complexity reductions needed in the security kernel. First, they wanted to remove protected supervisory functions that did not belong in the security kernel. Second, they wanted to take advantage of the natural separation afforded by independent processes to help implement protected functions. And third, they wanted to use a more structured programming technique to implement the security kernel in a way that would make it easier to verify. With these goals in mind, the project team set out to define the proper way to construct a secure operating system.

1. Constructing a Secure System

In the early research phase of Multics, two different approaches were discussed as a methodology for constructing a secure system [SCH75]. The first that was explored

involved designing a formal specification for the desired system security properties followed by a top-down design and implementation process. To prove the correspondence between the code implementation and the formal specification, formal program verification techniques would need to be employed to achieve a level of confidence comparable to that of a mathematician's rigorous proof. The step of fitting the security properties defined in the formal specification to that of the desired real-world properties (security policy) would be left to human intuition. The second approach was related to the methodology of "penetrate and patch" that would be conducted by tiger teams. With the goals of finding, cataloging, and repairing security flaws, this approach hoped to convince others of the computer security problem, create an understanding of the type of flaws that exist and can be exploited, and to assist in hardening the available systems from penetration attempts.

In later research, the team created a multi-pronged approach to assist in the achievement of the redefined design goals discussed earlier in the introduction (the removal of functions that did not belong in the kernel, the implementation of protected functions that take advantage of the natural process separation, and the use of more structured programming techniques to ease code verification) [SCH77]. This approach had the following four concurrent processes: (1) use program verification wherever feasible, (2) assign small, expert teams of programmers to audit portions of the code for understandability and possible errors, (3) test the system for operational use to observe the system reliability, and (4) assign a tiger team the task of breaking into the system.

Once the approach was decided, other design issues needed to be researched for the proper solution. The amount of common mechanism, which was discussed earlier as one of secure system design principles, was one of these that demanded much attention.

2. Common Mechanism Risks

To implement any explicit or implicit communication among computations, a common mechanism is required, such as those found in memory sharing, interprocess communication, or physical resource sharing. These common mechanisms provide channels of information flow between different sensitivity levels of the system. For instance, if operation A at the secret level can influence the value of some set of data

items that operation B at the confidential level notices, then the common mechanism that is said to exist between these two operations allows the conveying of sensitive information. With this information channel open, operation A may *directly* write new information into a specific data item so that operation B can read it, or operation A may call a procedure at the confidential level that *indirectly* changes the state of the procedure's internals, enabling operation B to take note of this change [SCH75]. If the intent of the system is to prohibit or constrain unauthorized communication between processes, then unconstrained access to these common mechanisms represents a risk.

The functions that do incorporate common mechanisms in their implementation carry a built-in risk, which malicious users could work to their advantage if they can find an exploitable flaw in the mechanism. As discussed above, these common mechanisms make it possible for the operation of one user to exert unauthorized observation or influence over the operations of another. The simplifying conclusion to this problem has been that if a function does not require any type of outside communication, then its implementation should not use any common mechanisms.

To avoid the exploitation of such vulnerabilities from the start of system design, the designers of the system must ensure common mechanisms are used only when necessary and that each common mechanism has no implementation flaws and no exploitable design vulnerabilities. Additionally, the mechanism implementation needs to protect itself from any external tampering. To provide for these requirements, a security kernel should be the least amount of common mechanism necessary to implement the sharing of information and resources that are desired in the system, or, more importantly, only those mechanisms that are necessary and sufficient. But to verify the correctness and completeness of this implementation, it needs to be understandable, straightforward, and of good software design.

Parnas [PAR96] commented that a program needs to be well structured, written in a consistent style, free of errors, and developed in such a way that each component is simple and organized. Complexity and good engineering was also important for the designers of Multics. "Reduced size and complexity of security-relevant software is a prerequisite to performing a convincing logical verification that a system correctly

implements the claimed access constraints, no matter what verification techniques are used. Without such verification of correctness, a system cannot be considered secure” [SCH75].

3. Complexity and Dependencies

Much of the complexity of a system implementation can arise from the implementation of only a few of its features. When one realizes that a particular feature causes complexity, it is time to review the importance of the feature and to see if a slight variation in its semantics might lead to a simpler implementation. These minor adjustments or variations in the semantics of the user interface can make major differences in the complexity of the kernel implementation.

The file system, memory management, and processor management portions of the supervisor of Multics, the bulk of the supervisor, were organized in six large modules [SCH77]. The problem that existed with these modules was the complicated cycles of dependency that were created in the kernel. To eliminate many of these dependency loops, a complete restructuring of the file system, memory management, and process management portions of the supervisor was needed. This required that the structure generated by the “depends on” relation between modules be a partially ordered set as discussed earlier in the argument first exploited in Dijkstra’s T.H.E. System. Requiring a loop-free dependency structure allows system correctness to be established one module at a time. In Dijkstra’s system, the correctness proof was done one layer at a time, which was simple to complete since T.H.E. was a deterministic system. To get the loops out of Multics, type extension was found to be the best solution.

Type extension, as defined by Schroeder, Clark, and Saltzer [SCH77], involves making all the modules of the system act as object managers, categorizing all the ways one module can depend on another, and organizing the modules in a loop-free dependency structure. An object manager and the modules it depends on are solely responsible for maintaining the integrity of the managed objects. Wirth [WIR95] describes using type extension as a way of providing a system with flexible extensibility, which guarantees that modules can be added later to the system without requiring

changes or recompilation. Therefore, new modules could be added that incorporate new procedures based on calling existing ones.

In the Multics system, a partition of dependencies into five categories (module component dependencies, name mapping dependencies, program storage dependencies, address space dependencies, and module interpreter dependencies) was considered complete and fairly intuitive for systems designed according to the rationale of type extension [SCH77]. Studies of Multics later showed that the impact on performance of the system from the kernel re-design, and related changes, was minimal. Considering the modules that were redesigned for clarity and size reduction that had to remain in the kernel, such as memory management and process management, this small performance impact was especially interesting to members of the project team.

To address the size issue, the Multics team found at the time that the number of source lines, independent of the language being used, was the most useful and consistent measure of the kernel size. In Chapter 4, a discussion will demonstrate how archaic counting lines of code is in comparison to other complexity metrics that have been introduced recently and are quickly becoming widely used.

With a review completed of the construction approaches, implementation issues, and complexity and size concerns, a discussion of the Multics security kernel design, goals, activities, and redesign conclusions will close this lessons learned, case study.

4. Security Kernel Design

Schroeder [SCH75] describes a security kernel as a minimal, protected core of software whose correct operation is sufficient to guarantee enforcement of the claimed constraints on access. This is what he considers to be the structural basis for organizing a secure system. The kernel must contain all those parts of the system that pertain to meeting the security requirements. Therefore, there is no part of the system outside the security kernel that can cause the system not to meet its security requirements. Also, the security kernel can contain only those parts of the system that are necessary to meet the security requirements. Hence, the security kernel should not contain anything that does not pertain to the meeting of the security requirements. With this architecture, the

security kernel can be small, compared to the system as a whole, and therefore has a better chance of being correct.

a. Non-kernel Software

Once the security kernel contains only what is necessary and sufficient to meet the security requirements, correctness of the remaining portions of the system (i.e., the non-kernel software) became an issue of debate for Multics. Of these, four categories were discussed by Schroeder [SCH75], including (1) system-provided programs, (2) programs constructed by a user, (3) programs borrowed from other users, and (4) common mechanisms that a group of users sets up to implement some function. It is true that these applications could still cause undesired release of information, modification of information, or denial of its use. But, as commented by Schroeder [SCH75], if the kernel is correct, then these undesired results will not be unauthorized. Therefore, it is apparent that the only mechanisms that are essential to verify correct are the common mechanisms of the security kernel.

b. Multics Security Kernel Goals

When work first began in engineering the security kernel for Multics, Schroeder [SCH75] constructed a list of goals that strictly dealt with the security kernel. First, the kernel should be sufficiently small, well structured, and easy to understand so that an expert could verify, with relative ease, its correctness through manual auditing. Second, since a common mechanism is required if one operation is to influence another, the kernel should embody all system-provided mechanisms that are common to more than one operation. And third, to obtain all the mechanisms that should be included in the security kernel, the security specification should be viewed as a set of constraints on the interaction of the various operations that occur in a computer system.

Therefore, the plan to produce a security kernel for Multics was viewed as three different activities, including (1) the removing of non-kernel mechanisms from the supervisor, (2) the restructuring of the remaining kernel, and (3) the partitioning of the kernel into multiple protection domains [SCH75].

c. Categories of Security Kernel Activities

The first activity involved taking functions not requiring implementation as common mechanisms out of the supervisor to be implemented in the user domains of a

process. An example of this in Multics was removing file system functions from the supervisor. The second dealt with the restructuring of the mechanisms that must remain in the kernel. Such activities can reduce both the size and complexity of the kernel. In some cases, a piece of the kernel can be eliminated and its function assumed by another kernel mechanism. An example of this in Multics was the restructuring of the file system mechanisms that had to remain in the kernel. And the third activity was the partitioning of the kernel into differently protected functional domains or security policy domains. This effort helped modularize the job of matching the kernel to the system security specification.

d. Redefining Key Aspects of the Project

After the work began on the Multics Project, many problems from a security point-of-view redefined key aspects of the project as well as raised some new interesting questions [SCH77]. The first problem was with the size of the central supervisor. For example, the source code was over 54,000 lines of code in length and the number of programmers that contributed to the project ranged in the area of a hundred or more. This made the task of conducting an integrity audit very difficult since it was necessary to inspect and understand every line of code in each and every program. The second problem was related to the ad hoc security mechanisms provided by the first design. Since some of these mechanisms, such as access control lists, rings of protection, and passwords, were not previously defined in any simple, underlying model of security, the job of auditing the supervisor was considered more difficult than that of the first problem because of the absence of a specification. Presented with this situation, the auditor would have no idea what to look for since no model ever existed to verify the implementation.

To redirect the project into solving these two concerns, the team decided to redefine the key aspects of the design. First, simplify the supervisor so as to make it more feasible to audit and understand. And second, provide a set of security functions that can be described by a simple, more straightforward formal model. The only new concerns these brought about were the achievability of a successful audit, the usability of a security function defined by a formal model, and the changes in system performance when these new aspects were implemented.

e. Conclusions of the Kernel Design and Redesign

After the initial design, the redefined aspects of the project concentrated specifically on the security kernel's reduction in size and complexity. As stated by Schroeder, Clark, and Saltzer [SCH77], "The initial projects of removing mechanisms from the Multics supervisor helped us understand what mechanisms needed to be present in a security kernel, but they did not help us understand how these pieces should be organized." Therefore, they concluded that in order to simplify the security kernel, it was important to develop an organizational rationale, like type extension as discussed above, for modularizing the required functions and fitting them into an understandable overall structure.

From their accumulated work and research on Multics, Schroeder, Clark, and Saltzer [SCH77] agreed that the primary conclusion of the project was that a kernel of a general-purpose operating system can be made significantly simpler by first imposing a clear standard that defines what it should include. This kernel concept, along with the type extension design discipline, appears to be a viable approach to security in large-scale systems. And as an added benefit, there does not seem to be a significant performance loss arising from using simpler more modular designs. Therefore, from their observations, a secure system need have no performance penalty.

Parnas [PAR96] decided that the most important lesson is "up-front investment." In any software that he considered a jewel, the designers had spent obvious amounts of time thinking about the structure before writing the code. In the case with Multics, the project spanned over four years and involved the work of several faculty members, graduate students, and staff members. It proved to be a success by later being evaluated as a Class B2 [DOD85] high assurance system. But as we will discuss in the next chapter, it was remarkable that such an ambitious system, which involved so many sources of complexity, could have ever been built.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. MANAGING COMPLEXITY IN OPERATING SYSTEM DESIGN

A. INTRODUCTION

Most of the software we see or buy, in the opinion of Parnas [PAR96], is ugly, unreliable, hard to change, and certainly not something that Dijkstra would admire. But why would commercial vendors place such undependable software on the market? Wirth [WIR95] believes that the reason uncontrolled software growth has been widely accepted over the last two decades is because customers have trouble distinguishing between essential features and those that are just nice to have. “Software vendors uncritically adopt almost any feature that users want,” claims Wirth. “When a system’s power is measured by the number of features, quantity becomes more important than quality.” To stay competitive, the vendors add those features the market demands to sell their products. Can vendors start over again and build an operating system that rids itself of such functionality, making it less complex, easier to understand, and easier to test to help provide the level of assurance demanded of a secure operating system evaluated under the Common Criteria or Orange Book? The Unix operating system, which gathered the ideas of many previous efforts, is one such example of starting over and developing a well-structured system.

As part of the Bell Laboratories’ Multics effort, Ken Thompson frustrated with the complexities of developing a large system under the cooperation of three large organizations (M.I.T., Bell Labs, GE) decided to start over on his own. His strategy was clear and straightforward. He wanted to start small and build up the ideas one by one as he saw how to implement them. Since its advent, Unix has evolved and become immensely successful as the system of choice for workstations, giving rise to other open system variants such as Linux, OpenBSD, and FreeBSD and commercial descendents such as Solaris and OS/X. From a security perspective, however, UNIX has several of its own flaws and vulnerabilities.

Managing complexity while building a secure operating system requires the attention of all participants involved in every portion of the system lifecycle. This chapter discusses complexity from a more abstract point-of-view followed by what

complexity exists in software. Testing and the McCabe Cyclomatic Complexity metric will then be introduced and discussed to give an understanding of how they are coupled to help give the evaluator and user more confidence that the system behaves according to the requirements specification.

B. BUILDING SYSTEMS THAT FAIL

From his experience of working on the Compatible Time-Sharing System (CTSS) and the Multiplexed Information and Computing Service (Multics) system, Corbató [COR91] commented on the task of building ambitious systems: “It almost goes without saying that ambitious systems never quite work as expected.” The question to ask when designing such systems, he pointed out, is not “*if* something will go wrong, but *when* it will go wrong?” With this recurring problem affiliated with complex systems, Corbató presented his view that when a design team has a multitude of novel issues to contend with while building a system, mistakes are inevitable. Corbató contributes these mistakes to what he defines as the properties of ambitious systems. These properties can be summarized in five points:

- They are often vast and have significant organizational structures going beyond that of simple replication.
- They are frequently complicated or elaborate and are too much for even a small group to develop.
- They are pushing the envelope of what people know how to do, and as a result there is always a level of uncertainty about when completion is possible.
- They often break new ground, offer new services, and soon become indispensable.
- They invite a flood of improvements and changes by virtue of having opened up a new domain of usage.

Gathering from his experience, Corbató [COR91] abstracted the major causes of these properties of ambitious systems into five categories he referenced as the main sources of complexity, including (1) scale, (2) new design domains, (3) human usage of computer systems, (4) rapid change, and (5) the frailty of human users.

Scale in system design and construction is introduced when a project, which requires a large number of personnel, requires many levels of management. With this company hierarchy, communication problems grow and add to the scale of the project as the size of the design team grows.

New design domains become an issue when software engineers or designers of a system need to make a large transition in the way they think about solving a given problem related to the design and construction. This could be related to language selection, the integrated development environment, or the method of design.

Another complexity that needs to be addressed in system design and construction arises from *human usage of computer systems*, where if an individual places total trust in every computer user, he is vulnerable to the antisocial behavior of any malicious user. For instance, a complicated trade-off zone is presented to the user of a system when offered a various arrangement of trust and security mechanisms. When considering productivity and convenience, for example, most users will select not to use passwords that are a nuisance to remember.

Rapid change, which is often driven by technology improvements, causes changes in procedures or usage. These quick modifications in standard operating procedures could create new vulnerabilities that were not considered in the original design; therefore this source of complexity also needs to be considered in system design and construction.

Finally, the *frailty of human users* is considered a source of complexity in system design and construction since users are forced to deal with the multiplicity of technologies in modern life, which produce vast changes in their life-styles. Because of the amount of stress that is created from this over stimulation of inputs, the natural defense of many is to increasingly depend on others to act as information filters. Hence, instead of creating a knowledgeable user base that understands the inter-workings of a computer and the threats to its security, the end result is a population of “ignorant users” that only demand functionality and ease of use.

C. SOFTWARE COMPLEXITY

Parnas [PAR96] and Wirth [WIR95] have both argued that if a consumer is offered a system that provides “essential features” or a more useful tool with all the “bells

and whistles,” most users will choose the latter. This has driven software vendors in the market to offer newer and better features, which in many cases offer no additional functionality, with each new release. “Often,” Parnas notes, “software has grown large and its structure has degraded because designers have repeatedly modified it to integrate new systems or add new features.” The disadvantage to this software design approach is the negative effect the added features have on the system complexity. Since the code is repeatedly changed and modified to add the previously unanticipated features demanded by the market, the software design becomes sloppy, slow, and unreliable. Processor speed and memory size used to be the way out for some vendors to compensate for this poor design, but considering the computer security problem we face today, the software needs to be as Wirth refers more “lean,” which will likely turn out to be smaller and faster.

As opposed to indirect software measures such as project milestone status and reported system failures, software complexity is one branch of software metrics that is focused on the direct measurement of software attributes related to module design. As pointed out by McCabe [MCC96], there are literally hundreds of software complexity measures, ranging from the simple, such as source lines of code, to the esoteric, such as the number of variable definition and usage associations. The important criterion for the selection of these different metrics, however, is the uniformity of application. This is what McCabe calls the *open engineering* criterion, where abstract models used to represent software systems should be as independent as possible of implementation characteristics such as source code formatting and program language. The objective of this approach is to set certain complexity standards and interpret the resultant numbers uniformly across all projects and languages. For instance, McCabe points out that the number-of-lines-of-code metric does not meet this criterion since it is extremely sensitive to programming language, coding style, and textual formatting of the source code. However, his *cyclomatic complexity metric*, which measures the amount of decision logic in a source code function, is independent of text formatting and nearly independent of programming language. Unlike the basic complexity measure of line counting, the McCabe Complexity is possible since the same fundamental decision structures are uniformly applied in all function-oriented, programming languages.

Thirty years ago, Anderson [AND72] believed that the sheer size of contemporary operating systems, ranging on the order of 100,000 instructions, created an amount of complexity that made the static design and implementation virtually impossible to validate. Moreover, when considering the dynamic behavior of the system, there is no practical way to check all of the possible control paths to verify that they produce correct results that are error-free. Since complexity is a common source of error in software, McCabe [MCC96] decided to use this relationship directly to allocate testing effort. Leveraging this connection between complexity and error thereby concentrates the testing effort on the most error-prone software.

D. COMPLEXITY AND TESTING

According to the definition provided by McCabe [MCC96], software testing is the process of executing software and comparing the observed behavior to the desired behavior. As commented by C.S. Lewis, “No issue is meaningful unless it can be put to the test of decisive verification” [PET00]. The major goal of this verification process, as viewed by McCabe, is the discovery of errors in software. With the absence of such errors, the secondary goal is to help build confidence in the proper operation of the software. But what does it mean if a testing process detects no errors? As pointed out by Dijkstra [DIJ68], “Testing can show the presence, but never the absence of errors in software.” When verifying and validating an operating system, this makes the problem of addressing the malicious software threat hard since testing cannot demonstrate the absence of artifices or trapdoors. Does this mean that the software is high quality or the testing process is low quality? From McCabe’s observations, many errors are found at the beginning of the testing process, with the observed error rate decreasing as errors are fixed. As the rate approaches zero, statistical techniques are used to determine a reasonable point to stop. In reality, however, this stopping point is usually dictated by a vendor’s delivery date to the market. This common approach by commercial vendors has its weaknesses, as argued by McCabe. First, since the testing effort cannot be predicted in advance, the schedule can expire long before the error rate drops to an acceptable level. And second, since the statistical model only predicts the estimated error rate for the underlying test case distribution being used, it may have little or no connection to the

likelihood of errors that are manifesting or present when the system is delivered. Other methodical approaches to software testing may help detect the errors more effectively.

Requirements analysis is another approach discussed by McCabe [MCC96] to detect the errors in the software. Using this technique, each requirement specification is converted into a set of test cases. Then, at least one of these test cases within the scope of each requirement is executed to verify the system behavior. According to McCabe, however, this procedure does not provide a complete solution. Since the requirements are written at a much higher level of abstraction than the code that implements it, much more detail can exist in the latter. Therefore, testing only at the requirements level may miss many sources of error in the software since only a portion of the implementation is actually being tested.

Another approach called *white box testing* uses the software implementation itself rather than the requirement specifications to guide the testing efforts [MCC96]. This technique, also called glass box, code-based, or structural testing, focuses on the detailed design of the software rather than on the functions. This is accomplished by checking primary statements, paths, and branches for correct execution. *Black box testing*, in comparison, hides the contents of the box and focuses only on the inputs, outputs, and principle functions of the software module. A benefit to the white box approach is that the entire software implementation is considered throughout the testing effort; however, if the software does not implement one or more requirements, the testing effort will not detect the portions of the code that are missing [MCC96]. This convinced McCabe that in order to account for these concerns, white box and requirements analysis testing would need to be coupled together to verify the system behavior.

Although there are many effective testing techniques that exist or are being developed through research, it is also important to design and construct software that can be effectively tested. By limiting the decision logic of each program module during development, the amount of complexity will be reduced directly versus the common, inadequate approach of controlling the number-of-lines-of-code. McCabe [MCC96] states that the key requirement of white box or structured testing is that all decision outcomes must be exercised independently during testing. To measure the work

associated with this requirement, he introduced his cyclomatic complexity metric for modules, which is proportional to the number of tests required for a software module.

E. MCCABE CYCLOMATIC COMPLEXITY METRIC

1. Definition and Applications

A straightforward way to assess the complexity of a method is to count the number of decisions in the flow diagram representation of the diagram. The measure proposed by McCabe [MCC96] computes a number that represents the amount of decision logic in a single software module based entirely on the structure of the software's control flow graph. The form of these decision paths and their number strongly relate to the anticipated difficulties encountered when testing and maintaining the software. Also since the cyclomatic complexity is intended to be independent of language and language format, this measure provides a single ordinal number that can be compared to the complexity of any other program. To understand the definition of cyclomatic complexity, a brief review of some graph theory will be presented.

Using graph theory as a mathematical description, the cyclomatic complexity represents the same property as the cyclomatic number utilized in a directed graph [MCC96]. A directed graph (or digraph) G is a pair (V, E) , where V is a finite set and E is a binary relation on V . The set V is called the vertex set of G , and its elements are called vertices. The set E is called the edge set of G , and its elements are called edges. Hence, the definition in graph theory of the cyclomatic number, ν , comes in the form $\nu(G) = e - n + 2p$, where G indicates that the complexity is a function of the graph, e is the number of edges, n is the number of nodes, and p is the number of connected entry and exit nodes in the graph.

Since each software module corresponds to a single function or subroutine in a typical language, they can be used effectively as a design component via a single entry and exit point mechanism [MCC96]. Therefore, in a control flow graph describing the logical structure of software modules, the graph consists of nodes, which represent computational statements or expressions, and edges, which represent transfer of control between nodes. Additionally, as explained by McCabe, these program control flow graphs are connected via a directed "virtual edge," which associates the exit node to the

entry node. From the formula of the cyclomatic number, the existence of one virtual edge means that p is equal to one, reducing the previous form to $v(G) = e - n + 2$, which defines the cyclomatic complexity. Figure 7 illustrates how this cyclomatic complexity is calculated from the Euclid module, control flow graph. The source code is provided for amplification.

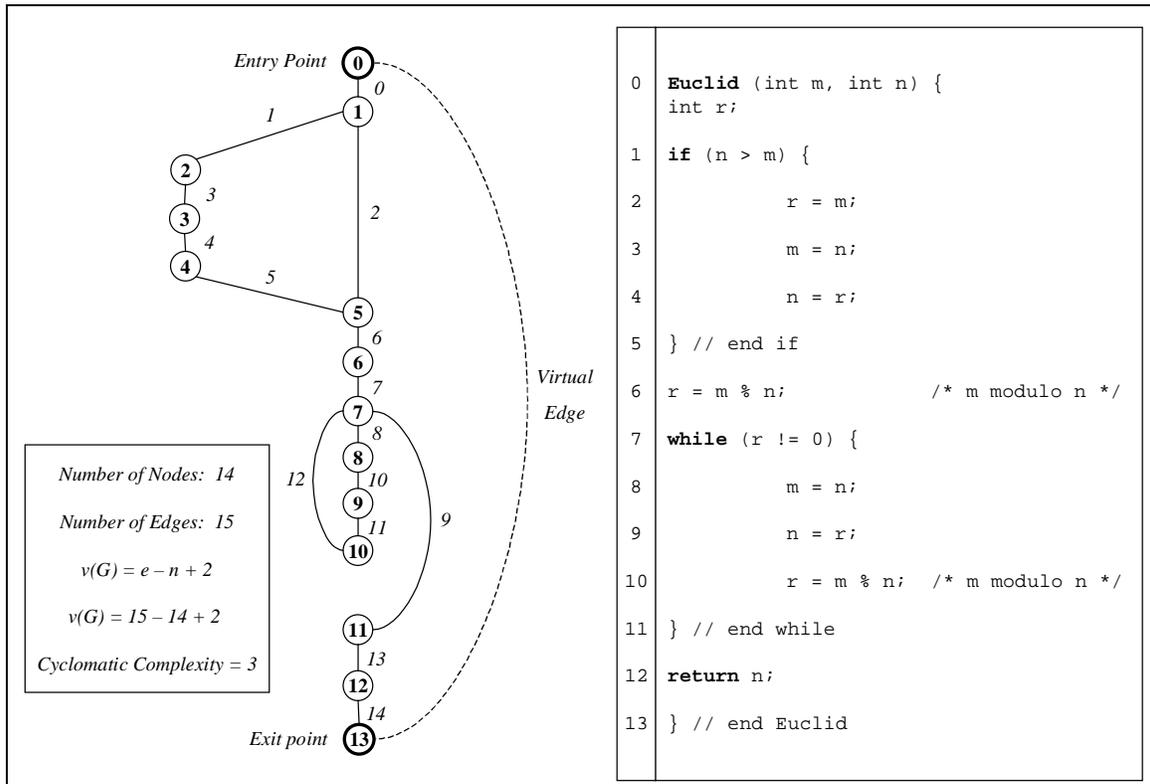


Figure 7. Euclid Module Code With Cyclomatic Complexity [MCC96].

As observed from Figure 7, each possible execution path of a software module has a corresponding path from the entry to the exit node of the control flow graph of the module. As defined by McCabe [MCC96], the cyclomatic complexity is the minimum number of paths that can, in linear combination, generate all possible paths through the module. With regard to testing, therefore, one only requires the cyclomatic complexity number of paths to cover all of the relevant edges of each software module. This resulting measure can be used in development, maintenance, and reengineering situations to develop estimates of risk, cost, or program stability.

As demonstrated by McCabe [MCC96], the correlation that exists between a module's cyclomatic complexity and its error frequency can be applied as a strong indicator of its testability, as well as its understandability and receptiveness to modification. From a software life-cycle standpoint, McCabe noted that cyclomatic complexity could be applied in several ways, including (1) code development risk analysis, (2) change risk analysis in maintenance, (3) test planning, and (4) reengineering.

2. Limiting the Cyclomatic Complexity

The Waterfall Model, described as a sequence of activities in a software lifecycle that begins with concept exploration and concludes with maintenance and eventual replacement, is one example of a software lifecycle process. From the model's various activities as described by McCabe [MCC96], maintenance turns out to be the most expensive of all these, with typical estimates ranging from 60% to 80% of the total cost. In this phase, complexity tends to increase due to the increased number of error corrections and functional enhancements, which are accomplished more often by adding code rather than deleting it. McCabe also makes note that these changes also increase the complexity of each module since it is usually easier to "patch" the logic in an existing module rather than introducing a new module altogether.

The maintenance phase, however, may not be the only step in the life cycle that is affected by this increase in complexity. The design and implementation phases could also create overly complex modules that are error-prone, hard to understand, hard to test, and hard to modify. Therefore, deliberately limiting complexity at all stages of software development helps avoid the pitfalls associated with high complexity software. From his research in software testing and measuring complexity, McCabe [MCC96] explains that the most effective policy for developers with a solid understanding of both the mechanics and the intent of the complexity limitation is the following: "For each module, either limit cyclomatic complexity to 10...or provide a written explanation of why the limit was exceeded." An organization can pick a complexity limit greater than 10, but only if (1) it is sure it knows what it is doing, and (2) it is willing to devote the additional design analysis and testing effort required by the more complex modules. A chart of threshold values is provided in Figure 8 on the next page to serve as a guideline for developers. The original limit of 10, as proposed by McCabe [MCC96], has significant supporting

evidence, but limits as high as 15 have been used successfully as well. Limits greater than 10 should be reserved for projects that have several operational advantages over typical projects. Some of these advantages include experienced staff, formal design, a modern programming language, structured programming, code walkthroughs, and a comprehensive test plan.

Cyclomatic Complexity	Risk Evaluation
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk program
greater than 50	untestable program (very high risk)

Figure 8. Threshold Values.

An attempt to limit complexity affects the allocation of code among individual software modules. For instance, limiting the amount of code in any one module might tend to create more modules for the same application. From a software engineering standpoint, however, the usual factors to consider when allocating code among modules are the cohesion and coupling principles of structured design as discussed in Chapter 3. This means that the ideal module needs to perform a single conceptual function (high cohesion), and do so in a self-contained manner without interacting with other modules except to use them as subroutines (low coupling). Thus, complexity limitation attempts to quantify an “except where doing so would render a module too complex to understand, test, or maintain” clause to the structure design principles [MCC96].

Other archaic rationale for limiting complexity, like reducing a software module to one page to make it more manageable, have no direct connection to a module’s actual complexity and should not be used as a measurement of such [MCC96]. For instance, one control flow graph may have 282 lines of code with a cyclomatic complexity of one; whereby a second flow graph represented by a 30-line module may have a cyclomatic complexity of 28. As this example makes clear, there is no consistent relationship

between the number-of-lines-of-code and complexity. Although the number-of-lines-of-code is an important size measure, it is still considered a crude measurement for complexity.

With a review of its definition and applications completed, some practical and simplified cyclomatic complexity calculations will be introduced to aid in module design and implementation.

3. Simplified Complexity Calculations

McCabe [MCC96] explains two ways to manually calculate the cyclomatic complexity of a program: (1) counting the decision predicates and (2) counting the flow graph regions.

On a control-flow graph, a binary decision predicate (i.e. “yes” or “no”) appears as a node with exactly two edges flowing out of it. Starting with one and adding the number of such nodes yields the cyclomatic complexity. For multi-way decisions (i.e. switch or case statements), the number added is one less than the number of edges out of the decision node.

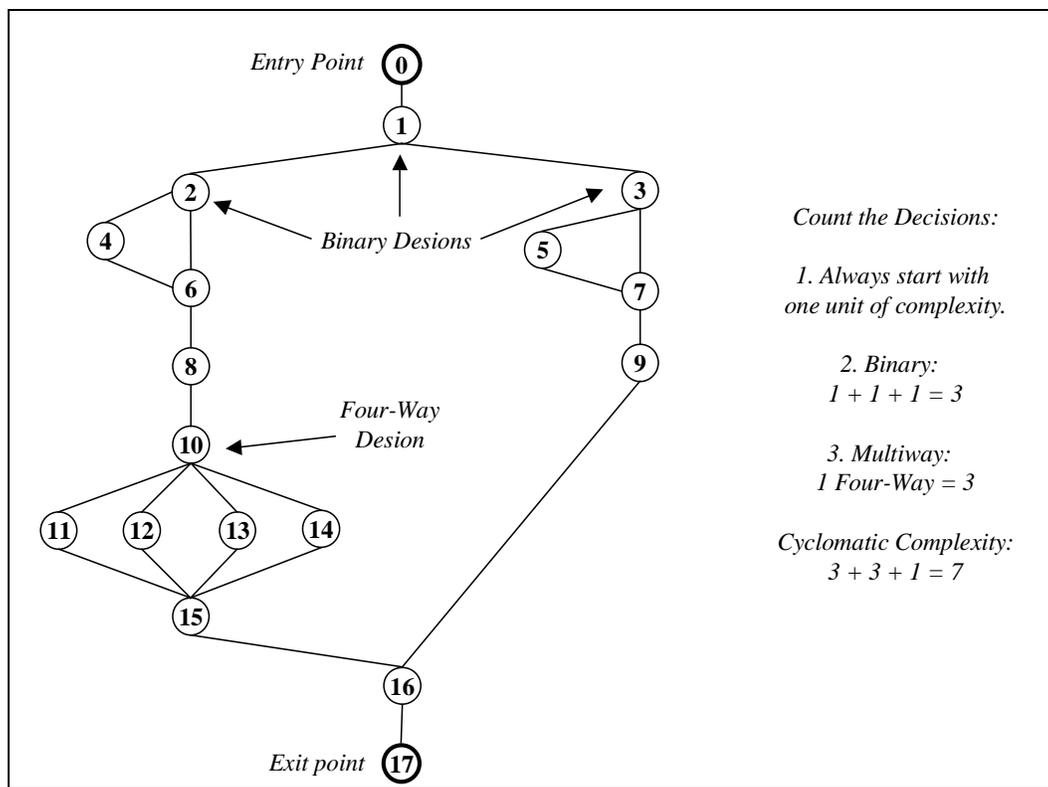


Figure 9. Binary and Multi-way Decisions [MCC96].

vertices, E is the number of edges, and F is the number of faces, can be applied to any three-dimensional shape. For instance, a tetrahedron has $V = 4$ vertices, $E = 6$ edges, and $F = 4$ faces, yielding $4 - 6 + 4 = 2$. As described by McCabe [MCC96], using R (the number of regions) to represent F (the number of faces), Euler's Formula can be rewritten for application to all planar graphs $n - e + R = 2$. Rearranging this formula gives the form $R = e - n + 2$, which matches the previous definition of the cyclomatic complexity. Thus, for a planar flow graph, counting regions gives a quick visual method for determining complexity. Using the flow graph in Figure 10 as an example, we start by counting the infinite region outside of the graph followed by counting the total number of internal regions, giving a cyclomatic complexity equal to seven. Since the control flow graphs in Figures 9 and 10 are identical, the method of counting the regions agrees with the method of counting the decision predicates used in the previous example.

Although the cyclomatic complexity can be calculated manually for small program suites, automated tools are preferable for most operational environments.

4. Automated Tool

Automated tools are the most efficient and reliable way to determine complexity. In a matter of minutes, complexity can be determined for hundreds of modules spanning thousands of lines of code. McCabe [MCC96], however, notes that in order for automated graphing and complexity calculation to be possible, the technology needs to be language-sensitive. Therefore, there must be a front-end source parser for each language, with variants for dialectic differences. This type of technology exists today and will be described in Chapter 5 with a description of the Imagix4D reverse engineering and documentation tool.

Another example of this technology is a plug-in, which calculates the cyclomatic complexity metric for Pascal source code, designed by Derrick [DER97] for the Code Warrior integrated development environment (IDE). The score that the program provides is the number of decision points gathered from the number of conditional statements (i.e., if, while, repeat, for, and, or, and case statements) in a routine plus one. Figure 11 shows the design and implementation scores that a programmer could use to limit complexity using the Code Warrior, cyclomatic complexity plug-in.

Scores	<i>Design</i>	<i>Plug-in Implementation</i>
8 - 10	Fairly simple	Gives user a note
11 - 19	Need to start thinking about simplifying	Gives user a warning
>= 20	Need to simplify the routine	Gives user a error

Figure 11. Complexity Plug-in Design and Implementation Scores.

The tools that are available for studying complexity are useful in software development, but McCabe [MCC96] makes an important observation regarding the use of any of these methods after the design and implementation phases are already complete: “The feedback from an automated tool may come too late for effective development of new software. Once the code for a software module (or file, or subsystem) is finished and processed by automated tools, reworking it to reduce complexity is more costly and error-prone than developing the module with complexity in mind from the outset.” Therefore, instead of pushing complexity-related issues to later phases of the software life cycle like the maintenance phase, manual and automated techniques as discussed in this chapter need to be applied for complexity analysis to help design and build easy to understand software.

Wirth [WIR95] states that reducing complexity and size must be the goal in every step of the design, including system specification, design, and detailed programming. This would bring one to logically infer that if a system is already constructed, it may be too late to go back and try to fix the complexity problems. Would it benefit the vendor and its design team to start again with a new software project that confirms and controls complexity throughout all of its development? After a discussion of the Imagix4D’s capabilities, an experiment that compares the complexity of different operating systems will help provide the answer to this question.

V. IMAGIX 4D REVERSE ENGINEERING TOOL

A. INTRODUCTION

There are several ways to calculate complexity of applications and operating systems in practice, ranging from counting decision predicates by hand, as discussed in the last chapter, to using a tool that performs the calculations in an automated, more error-free manner. When searching for a tool that specifically automates cyclomatic complexity calculations, Imagix 4D is the ideal application. From its description, Imagix 4D is a reverse engineering and documentation tool for legacy C and C++ software. As part of its wide functionality, the tool can be used on any level to rapidly check or systematically study software, including architectural details, functional dependencies, control structures, and data usage. By automating the browsing and analysis of the code, Imagix 4D speeds the comprehension of programs that are large, complex, or unfamiliar.

The following chapter will give an overview of the Imagix 4D tool starting with a description of its underlying database model, the entity-relationship-attribute symbols utilized by the database, the views and displays that present the information stored in the database, and the various modes that are available to browse, explore, and analyze the software code. Throughout this synopsis, screen shots will demonstrate the tool's functionality by analyzing a simple C++ program called "Vehicle Watch," which was developed as the final project in a beginner's programming class. This program tracks vehicle information such as speed and heading. The next chapter will present an analysis of other software that is not as easy to understand – operating system source code. From this study, we plan to show a complexity comparison between an open source, general operating system like Linux and a commercially available, embedded operating system like Talisker.

B. DATA MODEL

The underlying database, which is employed by the Imagix software, uses an entity-relationship data model. The symbols (entities) in the program code, their dependencies to other symbols (relationships), and specific information about any of the symbols (attributes) constitute the most natural way of handling and presenting the data about the program being analyzed.

When parsing the source code, all of the entities, attributes, and relationships are collected and updated in the database. The Imagix 4D database recognizes a wide range of symbol types, including file directories, program files (binary, C, C++, executable, header, and library), classes, functions, macros, variables, and data types. Figure 12, which depicts the “Set View” options for Imagix 4D’s Graph window, illustrates the various types of symbols available. The purpose and functionality of the Graph window will be explained in the next section.

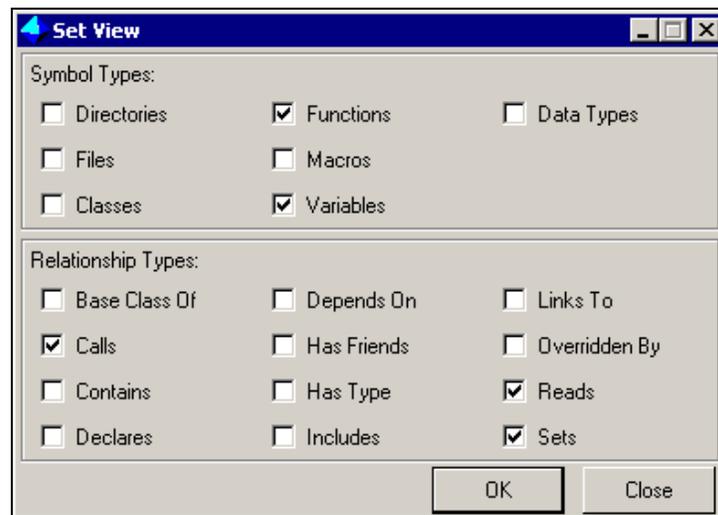


Figure 12. Graph Window - Set View Options.

Associated with each of these symbols is a set of relationships to other symbols. A wide variety of relationship types are collected and updated, including (1) function “calls” function, (2) file “declares” symbol, (3) file “contains” symbol, (4) variable “has type” data type, (5) file1 “includes” file2, (6) functions “reads” variable, and (7) function “sets” variable relations. Figure 12 also shows a list of these relationships that are possible to model in Imagix 4D.

For each symbol in the database, the reverse engineering tool collects a number of attributes as well. Of interest are the *function metric attributes*, which are illustrated in Figure 13 showing the different attributes available from the “List Options” menu. Of the metric options, “callers” is defined as the number of other functions that call a particular function. “Complexity” is based on the cyclomatic complexity metric as defined by McCabe. The total number-of-lines in a function is given by the “lines” metric, while the “LOC” metric is the number-of-lines-of-code containing tokens in the

function. And concerning the length of a file, the number-of-lines containing comments are given by the “LOCmmts” metric. To generate profile data, a “frequency” metric gives the number of times a function was called, a “coverage” metric gives the percentage of blocks of code that were executed, and a “time” metric gives the time spent executing the function. Besides functional metrics, other attributes such as file size and modification date can also be included in the Imagix 4D database.

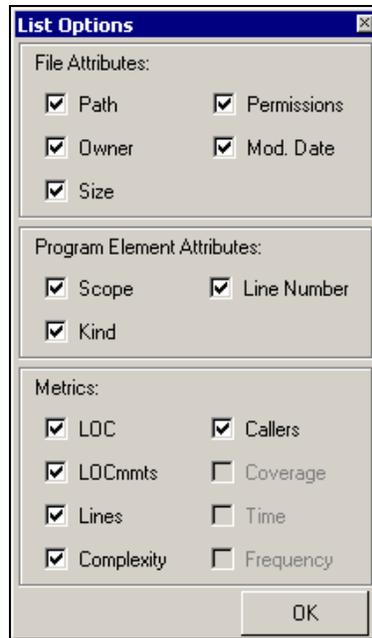


Figure 13. List Options Menu for Imagix 4D Database.

Information from the database is used to generate the various displays as well as support Imagix’s knowledge-based querying.

C. VIEWS AND DISPLAY WINDOWS

The display windows are the central focus of Imagix 4D’s user interface. Taking advantages of visualization technologies to convey information from Imagix 4D’s comprehensive database, these windows enable a user to achieve a fast and accurate understanding of the software. Since the views and displays present several different perspectives of the program simultaneously, their complementary nature enables a user to grasp the inherent design of the software.

1. Graph Window

The central window for displaying information is the Graph window. The window shows a group of symbols (such as particular functions, classes, variables, and/or

types) in the software, and how they relate to each other. For a given symbol, one is able to see at a glance how it interacts with other symbols.

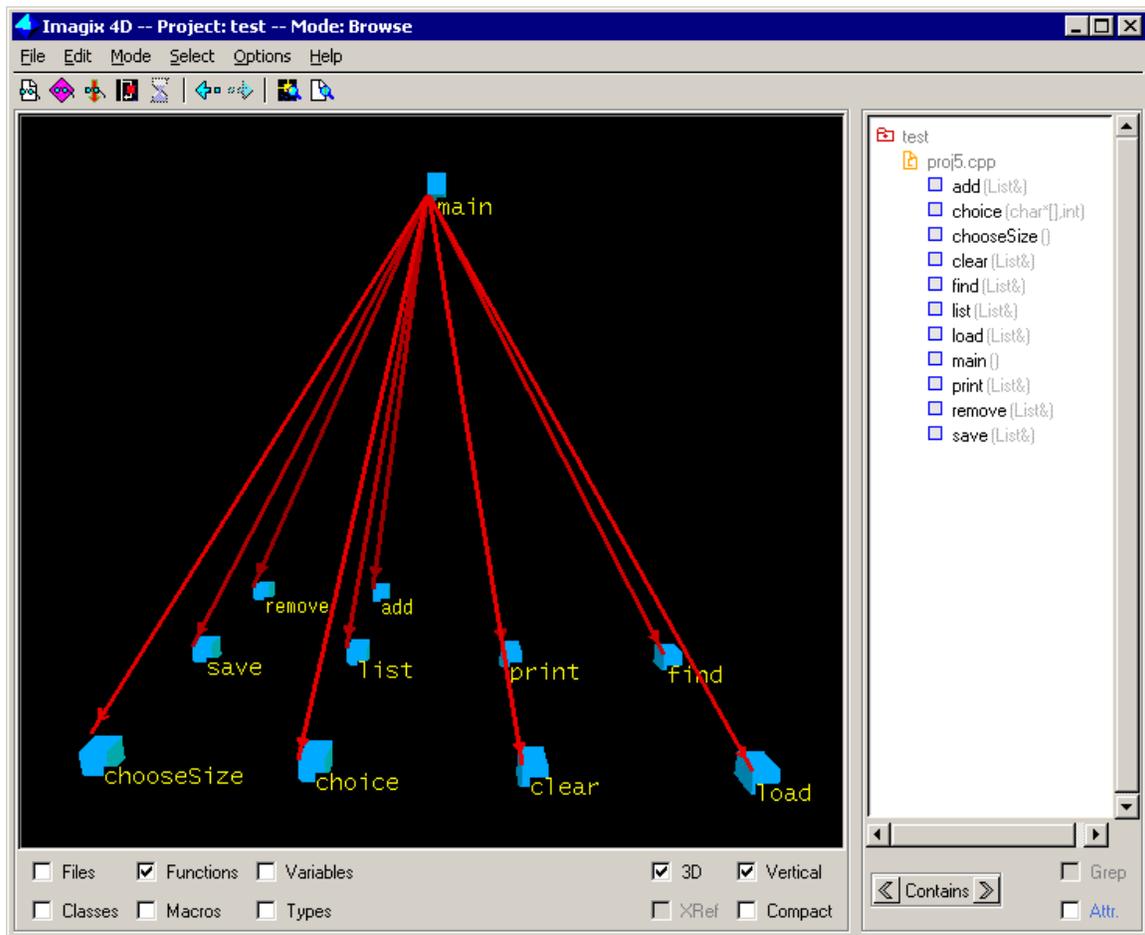


Figure 14. Graph Window and List Window.

The Graph window helps a user visualize and understand the software by pictorially representing the symbol and relationship data contained in the Imagix 4D database. In the view presented in Figure 14, functions are displayed as indicated by the selected radio button shown at the bottom left hand corner of the Graph window. This example shows the hierarchy of calls between functions, which will be utilized in our comparative analysis later in Chapter 6. The setting and reading of variables can also be visualized by selecting the appropriate checkbox.

In the Graph window and the Graph Symbols Key (shown in Figure 15), all the lines drawn between symbols representing the relationships have arrowheads. These indicate the direction of the “dependency” as directed graphs as discussed in the previous

chapter. For instance, when a function is said to call another function, an arrow from the calling function to the called function so indicates. This dependency direction is also given by the layout of the graph. In general, the symbols at the top of the graph tend to have dependencies upon the symbols lower in the graph. There are times, however, when a user will find a traditional 2D layout more appropriate for the data being observed. This 2D layout, which can be selected at the bottom of the Graph window as illustrated in Figure 14, is similar to the 3D view except that the dependencies tend to be displayed left to right, rather than from top to bottom.

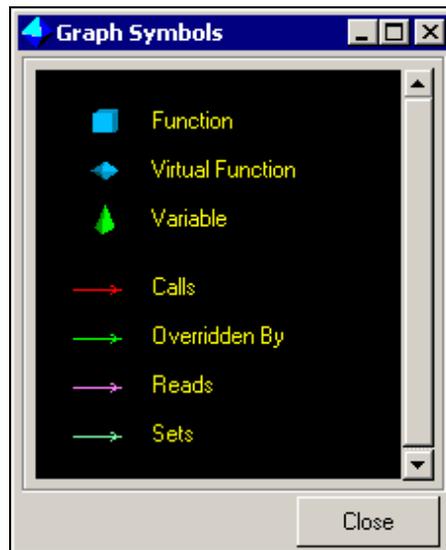


Figure 15. Graph Symbols Key.

One option available with the 2D view is the cross-reference layout, also shown as an option in the Graph window of Figure 14. When this option is checked, only the relationships that have a direct dependency with a pre-selected symbol are drawn, while suppressing all other relationships from the graph. For instance, if “main” were selected by clicking on its function symbol in the graph of Figure 14, it would become the center of focus when the cross-reference layout is selected in the 2D view. In the resulting display, symbols that directly depend on “main” will have arrows drawn *into* the selected symbol, appearing to the left of “main,” while those that “main” directly depends on will have arrows drawn *out of* “main,” appearing to the right. In this example, “main” has no “into” dependencies; therefore, it would only have arrows drawn in the “out of” direction, demonstrating what functions and variables it directly depends on.

2. Graph Symbols Key

The Graph Symbols Key, similar to the legend of a map, is a useful tool for keeping track of the symbols and the relationships that can be displayed in the current view. As illustrated in Figure 15, the key uses shapes and colors to indicate these different types. For instance, blue cubes are used for functions, and green pyramids represent data variables. Color is also used to indicate the type of relationship between two symbols. For example, the arrow representing a function calling another function is drawn in red, a function reading a variable appears as light purple, and a function that is setting a particular variable is shown in aquamarine.

3. List Window

While the Graph window shows the symbols and their relationships, the List window, as shown on the right side of Figure 14, complements the Graph window by displaying various characteristics about the symbols themselves. In particular, a user can utilize the List window to understand the context of a symbol and examine its attributes. For example, since the Imagix 4D database tracks where each symbol is declared, a user can use the List window to examine the hierarchy of containers. Hence, it can be observed in Figure 14 that variable “car” used in function “add” is defined in file “proj5.cpp” located in directory “test”. The “contains” adjust buttons below the List window enables one to control how many levels of containers are shown. Other symbol attributes like the number-of-lines-of-code in a function and its complexity, the scope of a certain variable, and the date a file was last modified can also be presented in the List window by selecting the “attr” checkbox located in the bottom right of the window.

4. File and Class Browsers

The Imagix 4D browsers provide information about files or classes from different perspectives. As shown in Figure 16, which depicts a typical file browser, the “Index” panel contains a list of all the files that have been loaded into the database, including source files and header files. The “Members” panel contains a listing of all the members defined in the current file as well as those that are only declared. These can be presented in the order they appear in the file, by name, or by type. A user of the application can also use the browser filter capabilities to limit the list to just those members of particular types, scopes, and origins. Finally, the “Relationships” panel shows how the current file

is related to other files in the database. During the examination of a file, a user may want to know what other files may include this one, or what other files have functions that call members of this file. As shown in Figure 16 with the “includes” relationship, a user can select other types that are available, including (1) calls, (2) sets, (3) reads, and (4) has type.

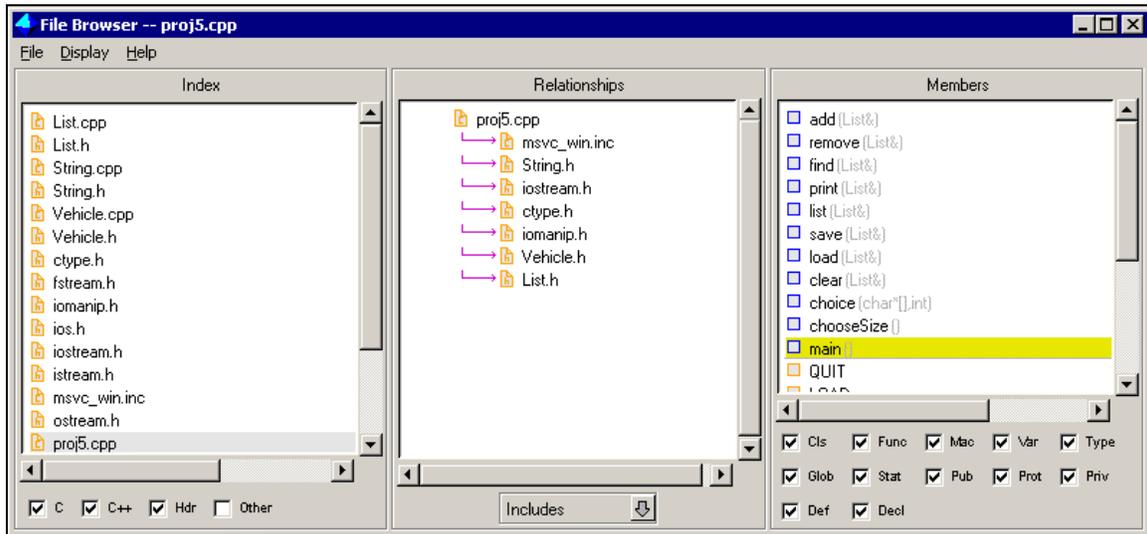


Figure 16. File Browser.

5. Flow Chart

The Flow Chart, as illustrated in Figure 17, shows the internal flow of control within a function. In this example, the decision logic of the “main” function is presented. For other more complex functions, which may consist of hundreds of lines-of-code, this capability can help the user visualize the internal logic of the routine. As also shown in Figure 17, the Flow Chart Symbols that can be selected help explain the symbols used in the control flow diagram. For instance, diamonds represent decision points; in-line blocks of code are shown as squares; and lines that connect the symbols show control flow.

An additional feature that helps the user see where a specific line of code fits in the overall logic flow of a function is the Flow Chart’s link to the File Editor. When the user clicks on a specific symbol, the cursor in the File Editor goes to that location in the source code. Likewise, clicking on the source code causes the matching symbol in the Flow Chart to be highlighted in red. By being able to see the source code in combination

with the logic of the function, one is able to understand and modify the function much more quickly and accurately.

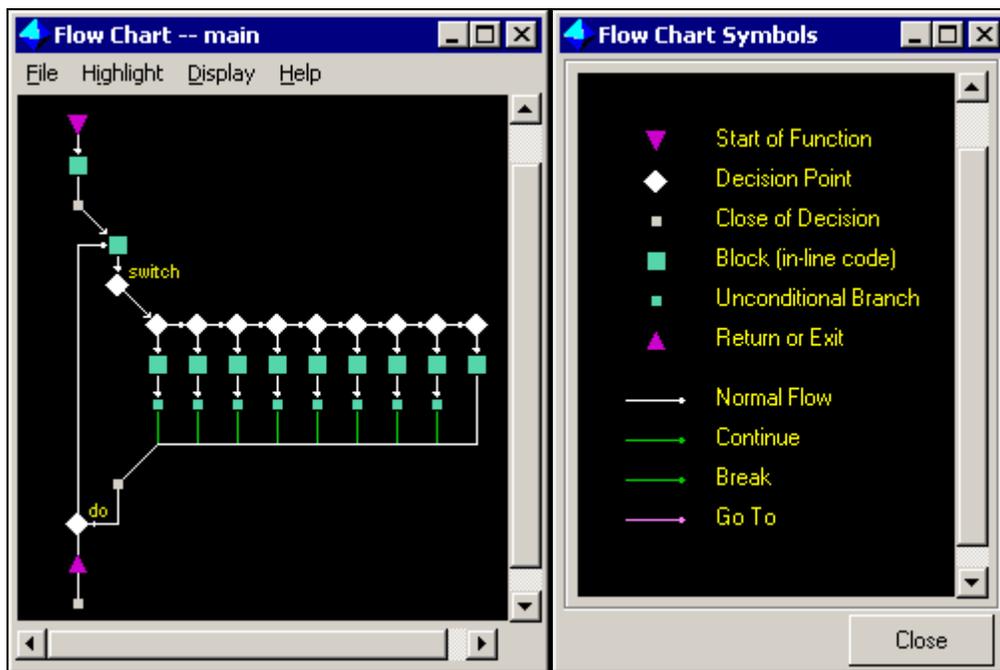
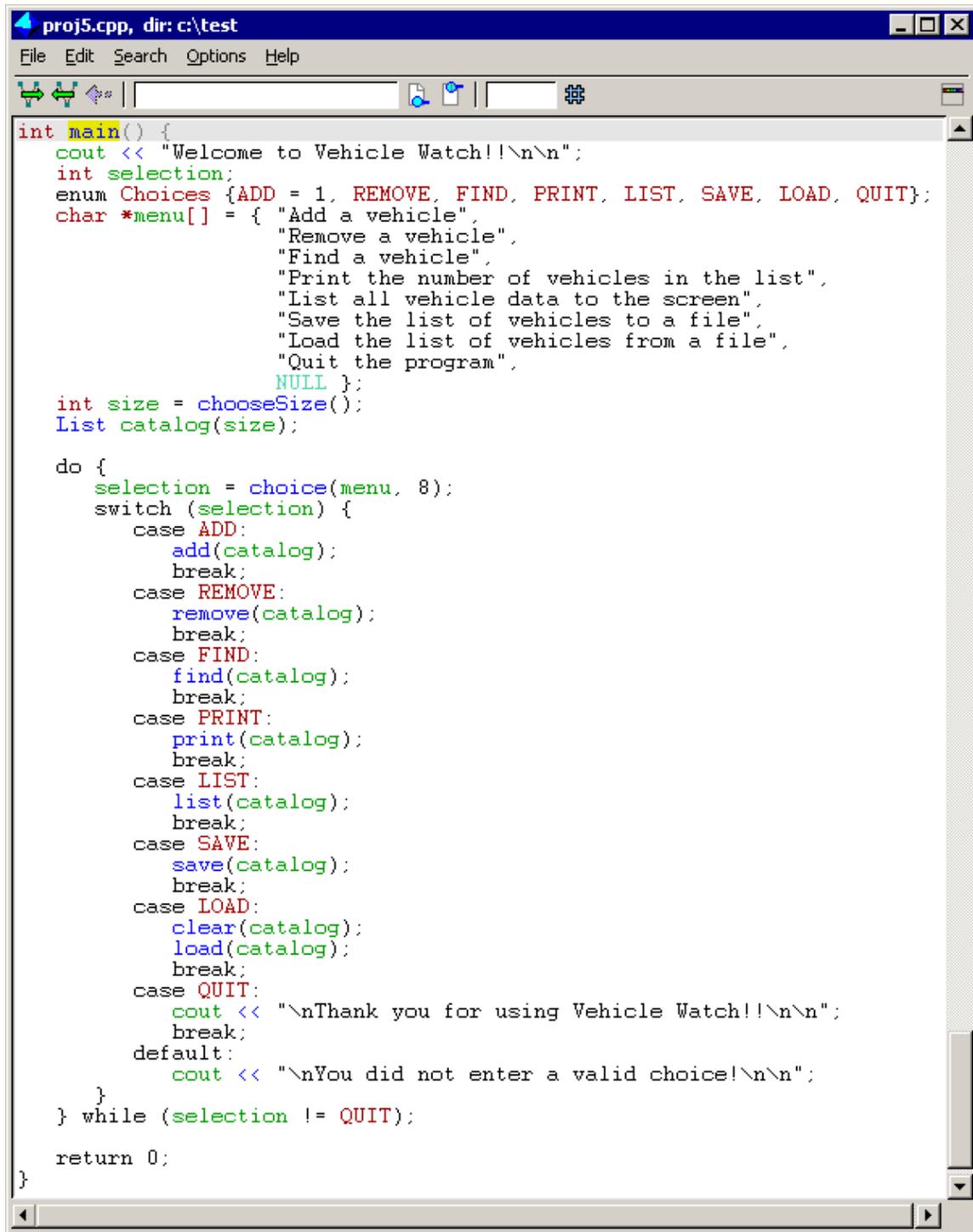


Figure 17. Flow Chart and Symbols.

6. File Editor

Imagix 4D's built-in file editor enables the user to achieve a better understanding of a program while examining and modifying its source code. An example of this capability is shown in Figure 18 where the "main" function is listed in its entirety.

To visually improve the level of understanding, color-coding is employed to distinguish between comments and actual code. This color-coding is also extended to symbols to aid in determining how they are defined and used in the code. To make browsing of the code more straightforward, hypertext-like navigation based on the information contained in the underlying database is achieved by tightly linking the file editor to other displays. For instance, by double-clicking on a colored symbol, the file editor immediately browses to the file and line number where that symbol is defined. This is also possible from the Graph window, the List window, the Browsers, the Flow Chart or a given Report that can be generated by the application.



The image shows a screenshot of a File Editor window titled "proj5.cpp, dir: c:\test". The window has a menu bar with "File", "Edit", "Search", "Options", and "Help". Below the menu bar is an icon bar with several icons, including a green arrow pointing right, a green arrow pointing left, a magnifying glass, and a search icon. The main area of the window contains C++ code for a program named "Vehicle Watch". The code defines a main function that prints a welcome message, initializes a menu, and enters a loop where the user can select from various options like "Add a vehicle", "Remove a vehicle", "Find a vehicle", "Print the number of vehicles in the list", "List all vehicle data to the screen", "Save the list of vehicles to a file", "Load the list of vehicles from a file", and "Quit the program". The code uses a switch statement to handle these options and a while loop to keep the program running until the user chooses to quit.

```
int main() {
    cout << "Welcome to Vehicle Watch!!\n\n";
    int selection;
    enum Choices {ADD = 1, REMOVE, FIND, PRINT, LIST, SAVE, LOAD, QUIT};
    char *menu[] = { "Add a vehicle",
                    "Remove a vehicle",
                    "Find a vehicle",
                    "Print the number of vehicles in the list",
                    "List all vehicle data to the screen",
                    "Save the list of vehicles to a file",
                    "Load the list of vehicles from a file",
                    "Quit the program",
                    NULL };
    int size = chooseSize();
    List catalog(size);

    do {
        selection = choice(menu, 8);
        switch (selection) {
            case ADD:
                add(catalog);
                break;
            case REMOVE:
                remove(catalog);
                break;
            case FIND:
                find(catalog);
                break;
            case PRINT:
                print(catalog);
                break;
            case LIST:
                list(catalog);
                break;
            case SAVE:
                save(catalog);
                break;
            case LOAD:
                clear(catalog);
                load(catalog);
                break;
            case QUIT:
                cout << "\nThank you for using Vehicle Watch!!\n\n";
                break;
            default:
                cout << "\nYou did not enter a valid choice!\n\n";
        }
    } while (selection != QUIT);

    return 0;
}
```

Figure 18. File Editor.

Another form of navigation, which cycles through all of the locations where a symbol is used, is achieved with the “Next” and “Prev” reference buttons located on the icon bar of the File Editor as shown in Figure 18. With these tools, the user avoids

having to spend the time and effort trying to remember where things are located and can focus instead on the understanding and possible modification of the source code.

7. Reports

Leveraging the information in its database, Imagix 4D can generate comprehensive reports and documentation about the software as shown in Figure 19. In this typical report, all of the function information is listed in order of its complexity. Other options for these information reports about functions include organizing the information: by name, by LOC, by lines, and by callers. Additionally, file and class summary as well as file information reports can be generated to include similar attributes as seen in the function report.

Function	Cmplx	LOC	Lines	Callers	File
List::fileLoad	17	62	67	1	List.cpp
add	16	73	78	1	proj5.cpp
main	11	49	51	0	proj5.cpp
List::fileSave	6	24	24	1	List.cpp
List::remove	5	19	19	1	List.cpp
load	5	15	16	1	proj5.cpp
List::listClear	4	12	12	1	List.cpp
save	3	12	13	1	proj5.cpp
remove	3	11	11	1	proj5.cpp
print	3	8	8	1	proj5.cpp
list	3	31	31	1	proj5.cpp
List::getNext	3	10	10	2	List.cpp
List::getFirst	3	11	11	2	List.cpp
find	3	15	15	1	proj5.cpp
List::find	3	9	9	1	List.cpp
chooseSize	3	17	17	1	proj5.cpp
List::add	3	11	11	2	List.cpp
List::~List	2	6	6	1	List.cpp
clear	2	5	5	1	proj5.cpp
choice	2	11	11	1	proj5.cpp

Figure 19. Typical Report – Function Information.

D. MODES OF OPERATION

To ease the task of program understanding, Imagix 4D operates in four different modes, including (1) browse, (2) explore, (3) analyze, and (4) control flow modes. Browse mode helps the user to quickly navigate through the code to examine the key relationships between functions, classes, variables, and types. The explore mode enables the systematic study of the software on any level from its high level architecture to the details of its build, procedural, data, and module dependencies. The analyze mode examines the quantitative characteristics of the software such as giving a visual indication of the complexity metric level of the design structure using predefined threshold values.

Finally, the control flow mode provides insight into the sequence and conditionality of function calls and variable usage.

All of the view and display functionality discussed thus far are all available in the four modes of operation. In the presentation that follows, additional capabilities affiliated with each mode in particular will be discussed for understanding and continuity.

1. Browsing the Source Code

When browsing the source code, the only other functionality permitted is the querying of the database using the “Find Query” and the “Grep Query” commands provided. When a “Find” is executed, the display windows focus on the results of the query.

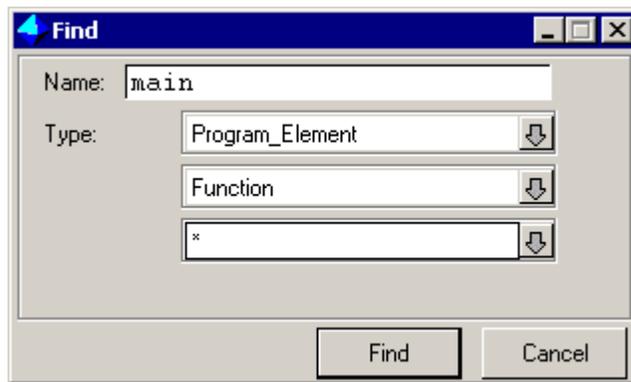


Figure 20. Find Query in Browse Mode.

The query, as illustrated in Figure 20, consists of a specific “Name,” a “Program_Element” type, and a “Function” subtype. “Files” is a second type that may be chosen where “Binary_File,” “Directory,” and “Source File” are some of its subtypes. Other “Program_Element” subtypes include the set {Class, Macro, Type, and Variable}. If a user executes the find query shown in Figure 20, the “main” function symbols in both the graph and list windows (illustrated in Figure 14) would become highlighted after the Imagix 4D tool searched its database for matches to the query.

The results of a “Grep” query, as illustrated in Figure 21, are listed in the search result window of the “Grep Tool” function available in all modes of Imagix 4D. In Unix, if “grep foo bar” was entered at the command-line, the system would return all the lines that *contain* a string matching the expression “foo” in the file “bar.” In Imagix 4D, this function enables a user to easily jump to the files where the “greped” strings are located.

After a grep operation is performed, the user can display these returned strings by selecting the “Grep” option available in the graph window as illustrated in Figure 14. This will result in a display of all the returned strings organized by location where they are declared. This is accomplished by listing them under a corresponding function, module, or file symbol. This list can be used as a basis for browsing to the actual occurrences of the string. In Figure 21, the tool is used to find all the occurrences of the expression “memory management” in the Linux operating system.

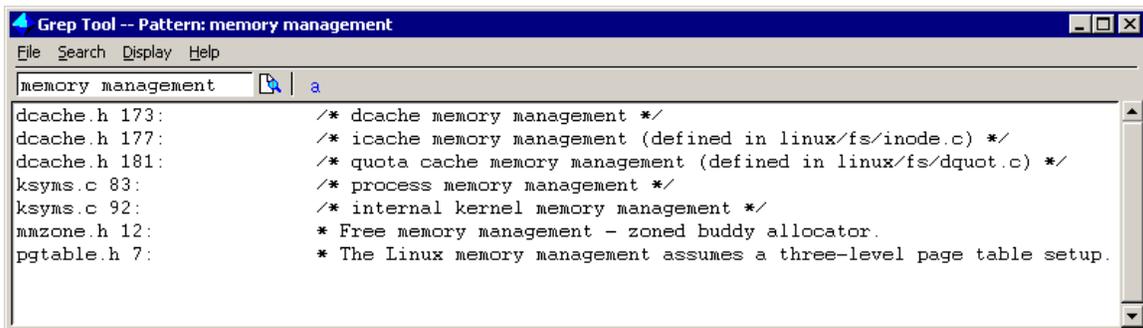


Figure 21. Grep Query in All Modes.

2. Exploration and Analysis of the Source Code

In *explore* and *analyze* modes, an additional capability available to the user is the tracing of software hierarchies in the source code. For instance, since the arrows on the relationship lines indicate the direction of the dependencies, a user can trace the hierarchy of dependencies quickly and simply by using the “Traverse Up” and “Traverse Down” functions available in these modes. These commands can also help find circular dependencies in the software hierarchies, as will be later demonstrated in Chapter 6.

Extending the database querying capabilities discussed earlier in the browse mode, the “Find” command in *explore* and *analyze* modes can quickly identify all symbols that meet a given criteria a user specifies by using the additional “Options...” menu item not available in browse mode. As shown in Figure 22, for instance, a user can base a “Find” query on certain attributes related to the symbol’s name, type, scope, file location, and quantitative characteristics such as lines-of-code or complexity. This command will determine what symbols meet the criteria, and where they exist within the context of the current view.

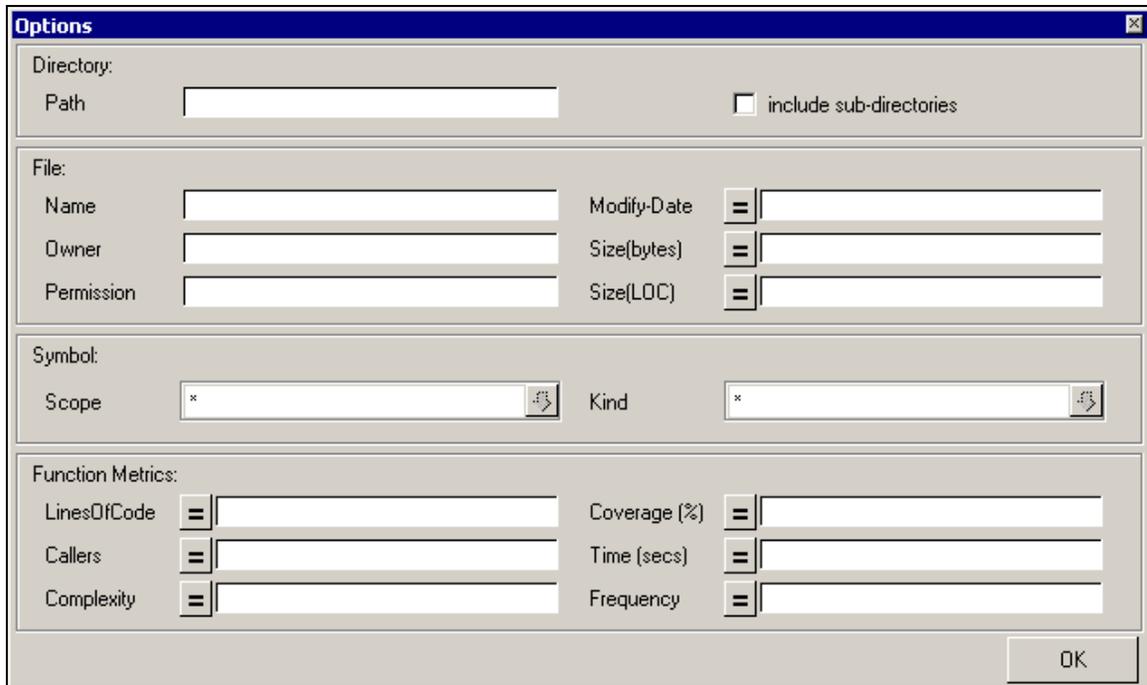


Figure 22. Find Query Options in Explore and Analyze Modes.

Another feature of explore and analyze modes is the capability to control and focus the content of inquiries into the source code to try and find more detailed information. For example, once a user has identified information of interest using the “Traverse”, “Find”, and “Grep” commands, he may want to remove unnecessary information and focus his view. To assist the user in accomplishing this simplification, data filtering functions (i.e., the “Isolate” function) are available. A complementary approach to this is to start with an overly simple view and add pertinent information (i.e., the “Add” function). Using either approach, the user can decide to focus his view in a way that follows his natural train of thought. In Figure 23, the menu for the “Add” command illustrates that symbols with a given name or type can be added “anywhere” in the graph; or relative to the symbol specified, symbols can be added in full or incremental steps in the up or down direction of the system hierarchy.

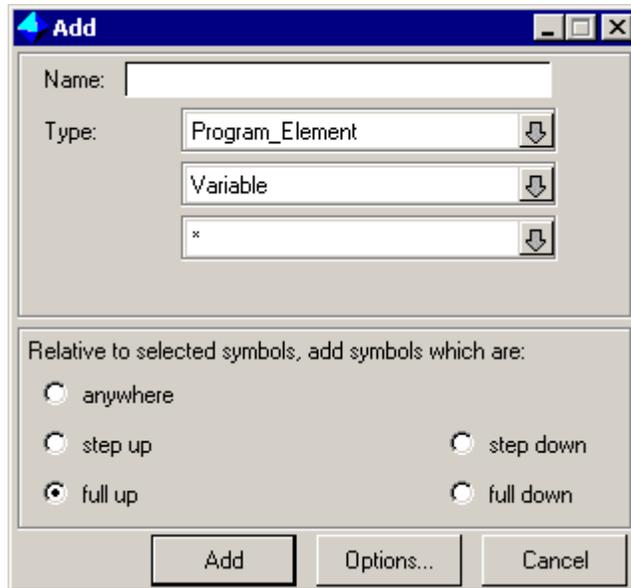


Figure 23. Add Function in Explore and Analyze Modes.

“Group” commands can also give the user a way to condense information without removing it. By collapsing a number of symbols together, for example, the information about all the relationships to the rest of the software can still be retained. This is useful when viewing software as a number of subsystems. When examining the sub-system interfaces, the user can see how one particular sub-tree interacts with the rest of the software without being burdened by the details of the sub-tree.

The views available in the analyze mode also provide insight into software metrics. When selecting the analyze mode, the user has a choice to view the source graph in relation to complexity, lines, LOC, or callers. In Figure 24, which shows the “main” function as the root, the symbols are color-coded to indicate whether they have a high, medium, or low value in relation to given complexity levels. By having a visual indication of the metric level as the user examines the design structure, he is able to more quickly identify areas that require attention for inspection, redesign, and modification. In this example, red indicates functions with the highest complexity and the least complex functions are green. In Figure 25, which depicts the complexity thresholds setting, a user is able to change the threshold values to meet his organization’s design policy.

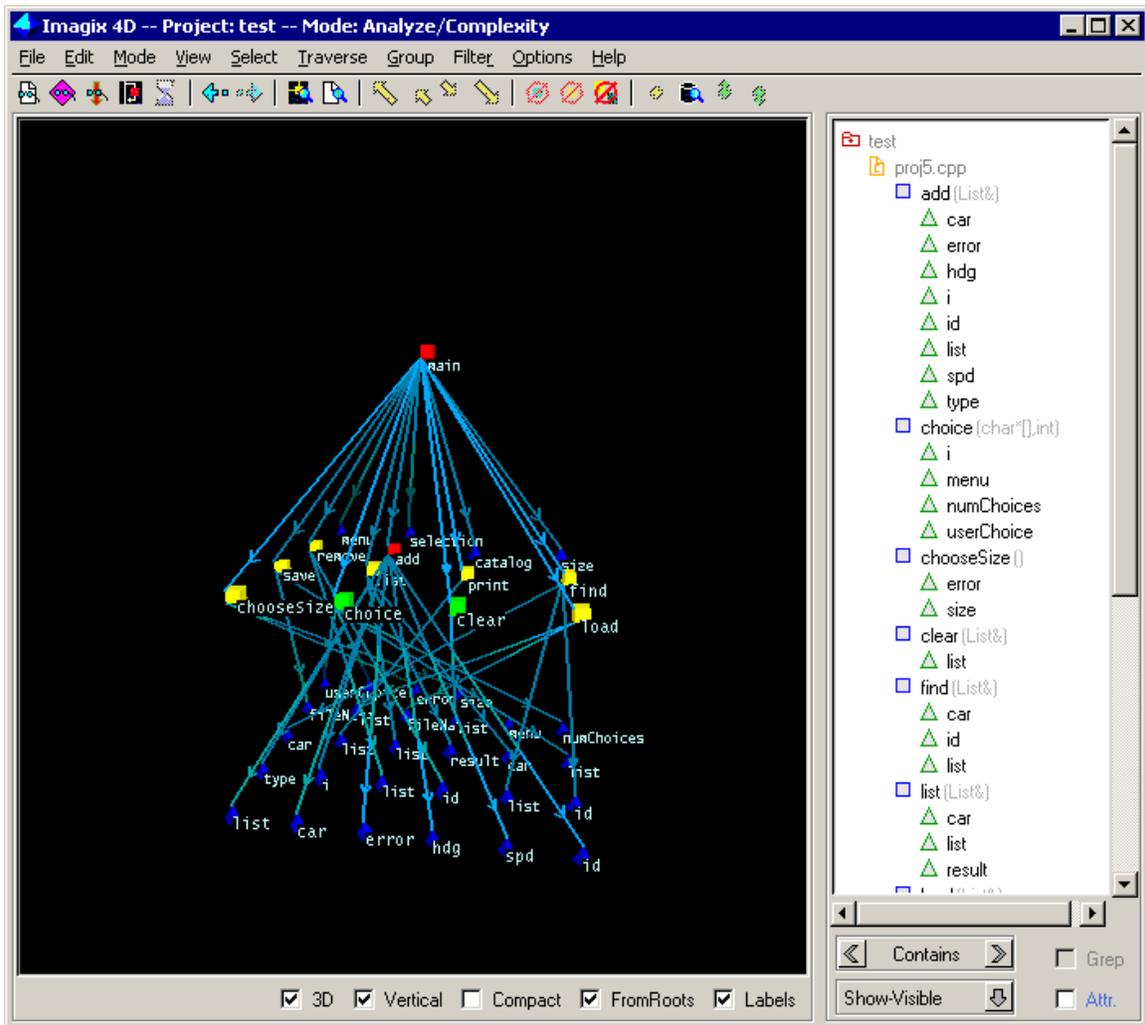


Figure 24. Analyze Mode - Software Complexity Analysis.

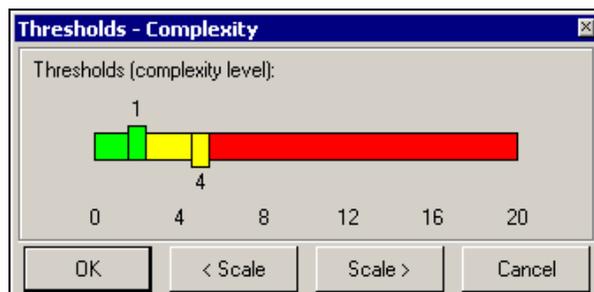


Figure 25. Threshold Complexity Level Setting.

3. Control Flow Analysis of the Source Code

There may be times when it is important to understand the sequence and conditionality of the function calls and variables usage in the source code. For this more detailed control flow analysis, Imagix 4D provides the control flow mode.

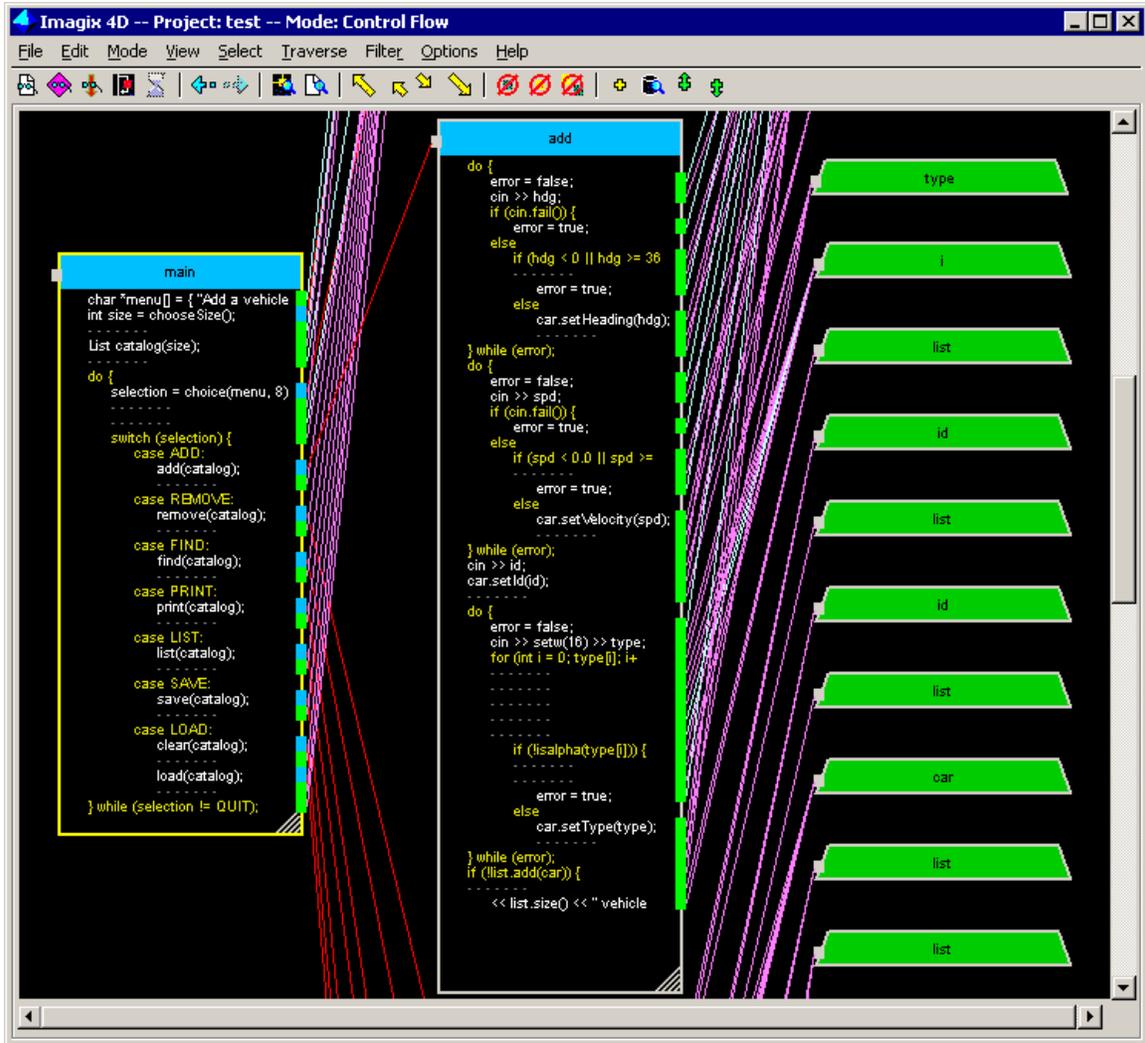


Figure 26. Control Flow Analysis.

For some symbols, their usage is the same in both the control flow mode and the standard graph. In both, blue, square-shaped symbols are associated with functions; and the symbols for variables are green and have sloped sides. The functions, however, are larger and contain additional information. From Figure 26, which depicts the “main” function and its interaction with functions and variables, it is observed that the functions enclose source code snippets with blue and green rectangles along their right edge. The snippets reflect the line of source code where the function call or variable usage occurs, the yellow text indicates that the source code was part of a decision, the white text is used for source code from a block of in-line code, and the blue and green rectangles along the right edge indicate whether a function is being called or a variable is being accessed. By

studying the control flow graph, one is able to learn about the order and conditions under which functions are called and variables are set and read.

D. CONCLUSION

With the goal of constructing source code that is straightforward and understandable, an automated tool such as Imagix 4D can be a valuable aid to help each programmer of the project team have a thorough comprehension of the software, thereby providing source code that is less complex, less error-prone, and easier to test.

With a review of Imagix 4D's capabilities completed, we will now apply its functionality to analyze the source code of the Linux, OpenBSD, and Talisker operating systems. In this context, we plan to observe the differences in complexity demonstrated by each of these systems.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. COMPLEXITY ANALYSIS OF LINUX, OPENBSD, AND TALISKER

A. INTRODUCTION

As this chapter will demonstrate, all three operational modes of the Imagix 4D tool (Browse, Explore, and Analyze) were used to study many different attributes of the Linux, OpenBSD, and Talisker operating systems. However, before discussing the basis of the analysis (the source code used, the data gathered using the Imagix 4D tool, and the final results from the comparative analysis) a brief background history of each operating system will be given to provide an overview of each system's origin, design goals or development process, and release distribution history. The experiment that will follow examines each operating system, its kernel, and its schedule module using the tool's various views, displays, and reports. After gathering enough data for analysis, the design and implementation of the three operating systems will then be compared in the context of the design principles previously presented, including hierarchical structuring, modular design, information hiding, the McCabe Cyclomatic Complexity, and the number-of-lines-of-code complexity metric.

B. BACKGROUND

1. Linux

Linux was initially developed in 1991 as an operating system for IBM-compatible personal computers [BOV01]. Linus Torvalds, the creator of the operating system and current owner of the Linux registered trademark, keeps the operating system up-to-date with various hardware developments and coordinates the activity of hundreds of other Linux developers around the world. As a result of this cooperation, developers have worked to make Linux available on other architectures, including Alpha, SPARC, Motorola MC680x0, PowerPC, and IBM System/390. An appealing benefit of Linux is the fact that it is an open source system. Under the GNU Public License, the source code is open and available for anyone to study. If the Linux kernel or its core utilities are modified or redistributed, the license stipulates that these changes must be released as open source. This has spawned many different activities in the operating system's continuous evolution.

One example of an activity shared by volunteer, non-commercial developers is the design and implementation of the Linux kernel source code, which is currently maintained at www.kernel.org. This web site and dozens of others are devoted to facilitating the simultaneous development of Linux. Of the versions presently available, each has two separate code trees to differentiate between a stable version and a development version. A simple numbering scheme is used to distinguish these kernel versions. The first number is the version number, the second number denotes whether it is a stable version (even) or development version (odd), and the third is the release number [BOV01]. For example, Linux 2.4.17, which will be used in the complexity analysis later in this chapter, is interpreted as Linux Version 2 - Stable Kernel 4 - Release 17.

A distribution of Linux provides a stable kernel version along with other standard utilities that the distribution creator feels is appropriate. Caldera, Debian, Slackware, TurboLinux, Red Hat, and SuSE are all examples of these distributions. Others include Corel, Mandrake, and variations on existing distributions such as Trustix, which is a pre-hardened version of Red Hat.

2. OpenBSD

In 1976, the Computer Science Research Group (CSRG) of the University of California in Berkley, CA started releasing an open source operating system called the *Berkley Software Distribution* or BSD [STA98]. This operating system was an open source derivative of AT&T's Research UNIX operating system, which is also the ancestor of the modern UNIX System V. Using the work initiated by CSRG, other projects like NetBSD and FreeBSD were created later with particular design goals specified. This caused the code base of the original version diverge to the point that the code of the multiple systems could no longer be merged. In 1996, another project called OpenBSD split off from NetBSD, making it the third version of BSD available today [STA98].

From a development standpoint, no one person or corporation owns BSD. It is created and distributed by a community of highly technical and committed contributors from all over the world. The BSD kernels are developed and updated following the open source development model [LEH99]. Each project maintains a publicly accessible source

tree under the Current Versions System (CVS), which contains all source files for the project, including documentation and other incidental files. CVS allows users to check out and extract copies of any desired version of the system available. The developers who contribute to the improvements of the OpenBSD project are divided into two categories, the contributors and the committers [LEH99].

Contributors write code or documentation. They are not permitted to commit or add code directly to the source tree. In order for their code to be included in the system, it must be reviewed and checked by a registered developer, known as a committer.

Committers are developers with write access to the source tree. In order to become a committer, an individual must show ability in the area in which he is active. It is at the individual committer's discretion whether he should consult with other committers prior to adding changes to the source tree.

Just like Linux, OpenBSD has a nomenclature for distinguishing the different versions of the operating system in the source tree. For instance, all new developments of the system are submitted into a branch called *CURRENT*. Unlike Linux, OpenBSD does not assign a number to this version (i.e., "OpenBSD-CURRENT"). At regular intervals, between two and four times a year, this branch of the OpenBSD project provides a new *RELEASE* version of the system, which is available on CD-ROM and via free download from mirrored ftp sites. The version that is released is intended for end users and is the normal stable version of the operating system. The version that will be used in the complexity analysis is "OpenBSD 2.9-RELEASE." As flaws are found in a *RELEASE* version, they are repaired and then added to the CVS tree.

In contrast to the numerous Linux distributions, there are only three open source BSDs: FreeBSD, NetBSD, and OpenBSD. Of these three, the design goals of OpenBSD are security and code purity. The project team meets these goals by using a combination of open source sharing and rigorous code reviews. Their objective is to create a system that is demonstrably correct, making it the choice of security-conscious organizations such as banks, stock exchanges, and US Government departments.

3. Talisker

The first device using Windows CE was the handheld PC 1.0, which was shipped in 1996 with the Windows CE 1.0 version (HAL01). Afterwards, new releases of the Windows CE operating system were given codenames based on tree types, including Alder (Windows CE versions 2.0/2.1), Birch (Windows CE Versions 2.11/2.12), and Cedar (Windows CE 3.0). However, the third generation of Windows CE has since separated from this tradition given the name Talisker, presumably after a Scotch whisky.

At the time of Version 1.0, no tools were publicly available to allow developers to build their own custom-embedded devices using Windows CE. However, Microsoft did ship a set of tools to the original equipment manufacturers (OEM) of hardware like Casio and Compaq, who were currently building the Handheld PC 1.0. These tools were known as the OEM Adaptation Kit, or “OAK.” The tools developed to help build the second generation of Windows CE, Versions 2.0-3.0, were named after the tools that could cut down a respective type of tree. For example, Axe was used for Adler, Buzzsaw for Birch, and Chainsaw for Cedar. But with the third generation, the operating systems and tools were no longer given separate names; hence, the combination is called Talisker.

Windows CE is an operating system built from scratch and specifically designed for the needs of embedded systems [HAL01]. Contrary to popular belief, it is not based on the Windows NT design nor is it compatible with NT applications. From the outset, the project had a number of design goals, including (1) building an OS suitable for small systems, (2) supporting a broad range of hardware, (3) using standard APIs, programming models, and tools, (4) making the OS componentized and ROM-able to allow for the best fit of memory and features, (5) enabling compelling products with built-in features like communications, user interface, database, and file systems, (6) real-time support, and (7) enabling battery-powered products by including unobtrusive power management.

Componentization, one of the core design goals of the Windows CE operating system, enables an embedded developer to choose the blocks or components of Windows CE that are appropriate for the device he is building [HAL01]. This approach makes it

possible to build a variety of devices using the Windows CE operating system, including industrial control systems, handheld PCs, and Pocket PCs. Since the Windows CE 3.0 operating system contains approximately 220 modules, the developer can use the Platform Builder 3.0 tool to configure a custom image of Windows CE 3.0 by selecting the components that are necessary for the device [HAL01]. Talisker developers will use the same approach using the updated Platform Builder 4.0.

With a review of system background completed, we can now proceed with the comparative analysis of the operating systems using an automated tool.

C. BASIS FOR ANALYSIS

Using Imagix 4D, six different “projects” to browse, explore, and analyze were created for the experiment, including (1) Linux_OS_X86, (2) OpenBSD_OS_X86, (3) Talisker_OS_X86, (4) Linux_Kernel_X86, (5) OpenBSD_Kernel_X86, and (6) Talisker_Kernel_X86. In the experiment conducted using these six projects, three design principles (hierarchical structuring, modularity, and information hiding) and two complexity measures (cyclomatic complexity and number-of-lines-of-code) were used as a basis for comparison. This section will present a brief overview of what these principles and measures represent, their importance to proper software design, and how they were used in the comparative analysis experiment.

1. Hierarchical Structuring

Recall from earlier discussion that to define a hierarchical structure, a partial ordering relation such as “uses” or “depends on” needs to be defined between the modules or programs. If the modules are assembled on different levels, a partial ordering can be introduced giving three additional benefits to the system: (1) parts of the system are simplified because they use the services of the lower levels, (2) if the upper levels were cut off, the remaining lower levels are still considered as a usable and useful product, and (3) common pitfalls of circular dependency such as live lock and deadlock are avoided.

If the system were designed in such a way that the low level modules made use of the high level modules, no hierarchical structure would exist, making it very difficult to remove portions of the system and still have it be usable and useful. Recall from the

Multics experience that in their attempt to rid the system of dependency loops, they decided to use levels of abstraction in their security kernel design as a means of reducing complexity and providing precise and understandable specifications. By making lower layers unaware of higher abstractions, the Multics kernel designers were able to reduce the total amount of interactions in the system and thereby reduce the overall complexity. In addition, these levels of abstraction simplified the correctness argument and kernel debugging since each layer could be tested in isolation from all higher layers.

Using the information gathered in its database after parsing the operating system source code, Imagix 4D can present 3-D graphical layouts that show a system's hierarchical structuring and dependencies. Using the tool's different graphical layout modes, we are able to see if the source code of each operating system is designed and written in a manner that exhibits partial ordering.

2. Modular Programming

A module that has been well designed encapsulates a function or database and it is always called upon by other modules for access to the module's services. In a sense, the module is a "database" containing entity that holds different types of information. To access this database, three types of interfaces can be defined: those that initialize the data, those that modify the data, and those that list the data.

When a software engineer decides to build a new system using modularity, the designer needs to define the problem, define the databases, and then decompose these into modules. This decomposition supports the understanding, analysis, and maintenance of the interfaces and can provide a basis for least privilege. As an added benefit, modular programming uses coding techniques that allow modules to be written with little to no knowledge of the code in other modules; this allows the modules to be replaced or reassembled without having any effect on the system as a whole. Recall that the other benefits of modular design include improved development time, product flexibility (i.e., if the interface is stable, the module can undergo drastic internal changes without affecting the other modules), and comprehensibility (i.e., it should be possible for anyone to study the complete system one module at a time).

Using the information gathered in its database, Imagix 4D can present file summary and function information reports that provide module characteristics, including the total number of modules, the number of functions included in each module, and the number of other functions that call a particular function given by the “callers” metric. From the data gathered, some inferences can be made regarding modularity. For instance, if an operating system uses a large number of modules, it may be inferred that the design properly supports the understanding, analysis, and maintenance of the interfaces. However, if a given module handles a plethora of different functions and manages many databases, its modularity is questionable. In addition, the “callers” metric may reveal how well this given module’s interface is defined and used. If the average number of callers per function is high, then other modules call upon the encapsulated functions of a given module through a well-defined interface. If this average is low, then there exists a wide number of functions that are called rarely, which may demonstrate an interface that is broad and not well-defined.

3. Information Hiding

A protected subsystem, as defined in earlier discussion, is a collection of procedures and data objects that is encapsulated in a domain of its own. This encapsulation allows the data structure of the protected object to be interpretively accessed; for example, it does not allow its internal organization to be accessible except by the internal procedures of the protected subsystem, which may be called only through specifically designated, domain entry points.

Information hiding is such a design approach where the software is decomposed into modules that hide design decisions. This helps shift attention away from the code used to implement the module and concentrate more on the signature or interface of the module. Therefore, the member variables used by each module should be private. To manipulate this private data, a simple, high-level, well-defined interface should be provided in the design to hide the implementation details. For instance, if a program tracks vehicle velocity, the module that encapsulates this data should include “set velocity” and “get velocity” functions rather than permitting an interface that directly manipulates the variable. In comparison, the use of global variables in a software design goes against the principle of information hiding.

From its database, Imagix 4D can present file summary reports that show the total number of variables included in each source file as well as the total number of global variables included in the header files. Using the explore features of Imagix 4D, all of the variables in the source code can be queried to see which of these are also considered to be global by the automated tool. A variable is categorized as global if it is manipulated by more than one module.

4. McCabe Cyclomatic Complexity

The McCabe Cyclomatic Complexity metric, which measures the amount of decision logic in a source code function, is independent of text formatting and nearly independent of programming language. When considering the dynamic behavior of the system, there is no practical way to check all of the possible control paths to verify whether they produce correct results that are error-free. Since complexity is a common source of error in software, McCabe decided to use his complexity metric to directly allocate testing effort. Since the cyclomatic complexity is the minimum number of paths that can, in linear combination, generate all possible paths through the module, one only requires this complexity number to cover all of the relevant edges of each software module giving a strong indicator of its testability, as well as its understandability and receptiveness to modification. This resulting measure can be used in development, maintenance, and reengineering situations to develop estimates of risk, cost, or program stability.

For our analysis, recall McCabe's threshold points for the cyclomatic complexity as it pertains to software design. A value of less than 10 is ideal for good design; a value greater than 10 but less than 15 requires that the designers know what behavior is desired, and that they know how to achieve this mapping of specifications to the code implementation properly. However, a value greater than 15 should only be attempted by an experienced and knowledgeable staff that is using formal design methods, a modern programming language, structured programming, code walkthroughs, and a comprehensive test plan. From the Imagix 4D database, function information reports can generate the cyclomatic complexity numbers for functions included in each module. These values can be compared against McCabe's threshold points to see the relative complexity of each operating system.

5. Number-of-Lines-of-Code

Although the McCabe Cyclomatic Complexity metric is a more accurate measure of the complexity exhibited by a module's decision logic, Imagix 4D also includes other software metrics such as the total lines of code, which are included as metric data in the file summary and function information reports. From this archaic measure, the average number of lines per function can be calculated giving a rough estimate of the relative complexity of the operating system. For instance, if a function exceeds a full page or 60 lines of code, it may be considered to be too complex.

D. SOURCE CODE FILES INCLUDED IN THE ANALYSIS

The source trees as illustrated in these tables represent the actual amount of operating system source code loaded into the automated tool and parsed into the Imagix 4D database. Once this information is available to the tool, the source code can be browsed, explored, and analyzed.

1. Operating System Source Code

The source code directories for each operating system are shown in Tables 1, 2, and 3. Some source code files related to networking, non-X86 architectures, and drivers as well as any documentation directories were removed from the Linux, OpenBSD, and Talisker projects to focus the study on the core i386 architecture, including the file system, process scheduling, management of input/output (I/O) devices, memory management, and the necessary library files.

From the directories shown, a few differences in the design of each operating system can be seen. For instance, the OpenBSD system has a much larger number of directories that are more specific to a set of functionality (i.e., different file system directories), while Talisker has a broader categorization of its source code. This difference may reflect OpenBSD's design goal to build an operating system that is highly modular.

All three operating systems show many similarities. For example, each system has directories for file system management. Library and header files are in wide use since all three operating systems are written in C. Each system appears to employ the kernel concept in its system architecture. Besides the graphics, window manager, and event manager, the one other difference in Talisker is the absence of a memory

management directory. Rather than keeping it separate, Talisker instead includes this source inside its *new kernel* (nk) directory. As we will see later in our analysis, the designers of Talisker decided to include much of the operating system, including the large GWE source, in the kernel apparently for performance reasons.

asm	fs	i386	init	ipc
kernel	lib	linux	mm	

Table 1. Linux 2.4.17 Source Directories – X86 Architecture.

adofs	i386	isofs	kern	lib
misfs	msdosfs	nfs	stand	sys
ufs	uvm	vm	xf	

Table 2. OpenBSD 2.9 Source Directories – X86 Architecture.

core	device	fsdmgr	gwe	nk
-------------	---------------	---------------	------------	-----------

Table 3. Talisker Source Directories – X86 Architecture.

2. Kernel Source Code

The kernel source code modules for the Linux operating system are listed in Tables 4 and 5. The reason for this separation of the kernel source is in accordance with the way Linux handles portability and reusability of their code. Table 4 lists the modules that are applicable to all platform architectures, while the modules in Table 5 are specific to the X86 Intel architecture. The kernel source code files for OpenBSD in Table 6 and Talisker in Table 7 also lists the modules that were loaded into the Imagix 4D database.

With the total number of modules totaling 60 in Linux and OpenBSD at 82, one could infer from the names of the modules that both of these systems may use some form of modular programming in their kernel designs. Talisker, in comparison, with only 40 modules in its kernel may fail to demonstrate whether the design employs a modular approach or not. As previously discussed in the basis of the analysis, a module should encapsulate one database, whether real or abstract, that handles a specific portion of the problem decomposed and shared by several specific modules.

asm	dma.c	kmod.c	ptrace.c	sysctl.c
kernel	exec_domain.c	ksyms.c	resource.c	time.c
linux	exit.c	module.c	sched.c	timer.c
acct.c	fork.c	panic.c	signal.c	uid16.c
capability.c	info.c	pm.c	softirq.c	user.c
context.c	itimer.c	printk.c	sys.c	

Table 4. Linux Kernel Source Files – All Platforms.

acpitable.c	i8259.c	mpparse.c	pci-visws.c	sys_i386.c
apic.c	init_task.c	msr.c	process.c	time.c
apm.c	io_apic.c	mtrr.c	ptrace.c	traps.c
bluesmoke.c	ioport.c	nmi.c	semaphore.c	visws_apic.c
cpuid.c	irq.c	pci-dma.c	setup.c	vm86.c
dmi_scan.c	ldt.c	pci-i386.c	signal.c	
i386_ksyms.c	mca.c	pci-irq.c	smp.c	
i387c	microcode.c	pci-pc.c	smpboot.c	

Table 5. Linux Kernel Source Files – X86 Architecture.

CVS	kern_fork.c	kern_timeout.c	syscalls.c	sysctl.c
sys	kern_fthread.c	kern_xxx.c	sysv_ipc.c	uipc_syscalls.c
exec_aout.c	kern_kthread.c	kgdb_stub.c	sysv_msg.c	uipc_userreq.c
exec_conf.c	kern_ktrace.c	subr_autoconf.c	sysv_sem.c	vfs_bio.c
exec_ecoff.c	kern_lkm.c	subr_disk.c	sysv_shm.c	vfs_cache.c
exec_elf.c	kern_lock.c	subr_extent.c	tty.c	vfs_cluster.c
exec_elf64.c	kern_malloc.c	subr_log.c	tty_conf.c	vfs_conf.c
exec_script.c	kern_ntptime.c	subr_pool.c	tty_pty.c	vfs_default.c
exec_subr.c	kern_physio.c	subr_prf.c	tty_subr.c	vfs_init.c
init_main.c	kern_proc.c	subr_prof.c	tty_tb.c	vfs_lockf.c
init_sysent.c	kern_prot.c	subr_rmap.c	tty_tty.c	vfs_lookup.c
kern_acct.c	kern_resource.c	subr_userconf.c	uipc_domain	vfs_subr.c
kern_clock.c	kern_sig.c	subr_xxx.c	uipc_mbuf.c	vfs_sync.c
kern_descrip.c	kern_subr.c	sys_generic.c	uipc_mbuf2.c	vfs_syscalls.c
kern_event.c	kern_synch.c	sys_pipe.c	uipc_proto.c	vfs_vnops.c
kern_exec.c	kern_sysctl.c	sys_process.c	uipc_socket.c	vnode_if.c
kern_exit.c	kern_time.c	sys_socket.c	uipc_socket2.c	

Table 6. OpenBSD Kernel Source Files.

celog.c	heap.c	kdpacket.c	mdx86.c	profiler.c
compr2.c	intrapi.c	kdriver.c	memtrk.c	resource.c
dbg.c	kdapi.c	kdtrap.c	objdisp.c	schedule.c
dbgasync.c	kdbreak.c	kdkernel.c	odebug.c	sprofile.c
debug.c	kdctrl.c	kmisc.c	physmem.c	strings.c
exdsptch.c	kddata.c	kwin32.c	ppfs.c	stubs.c
exsup.c	kdinit.c	loader.c	printf.c	sysinit.c
fault.c	kdmov.c	mapfile.c	profile.c	virtmem.c

Table 7. Talisker Kernel Source Files.

E. INFORMATION GATHERED FROM IMAGIX 4D

1. Function Call Graphs

Using the different graphical layout modes available with Imagix 4D, we are able to see if the source code of each operating system is designed and written in a manner that exhibits partial ordering. To observe whether this property is satisfied, Imagix 4D includes in all of its operating modes the compact graph layout option to help visualize the layering of the system design.

Under the *compact* layout, all of functions at the top-most layer are considered the root function symbols, which we will call “Layer 1.” In the operating system architecture, “Layer 1” is the application-programming interface (API). The next level or “Layer 2” contains all of the function symbols *directly related* to the root function symbols in “Layer 1.” The third level (Layer 3) contains all of the functions that are directly related to the second level (Layer 2) but not the initial root level (Layer 1), and so on. Hence, the functions in “Layer 1” call the functions in “Layer 2,” which in turn call the functions in “Layer 3,” and so on. This direct relationship is a “depends on” relation between the software layers. However, the “depends on” arrows can point in either direction for function calls or reads. Recall that if the operating system source code is designed and written in a manner that exhibits partial ordering, then there should be no arrows in the up direction between the different software layers.

The Linux, OpenBSD, and Talisker operating systems, kernels, and schedule modules were viewed using the compact layout mode. In the operating system and kernel projects, the following procedure was used to see whether any “depends on” arrows were pointing up toward higher layers in the hierarchy:

- In “Browse” mode, select the function checkbox in the bottom left hand corner of the “Graph” window. (We only wish to view the operating system functions.)
- Shift to the “Explore” mode and select “Restore All” from the “Filter” pull-down menu. (This will display all of the operating system’s function symbols.)
- Select “Root Symbols” from the “Select” pull-down menu. (This will highlight Layer 1 in the graph.)
- Select “Group Selected...” from the “Group” pull-down menu and enter “Layer 1” for the group name. (Grouping the functions of Layer 1 into one graph symbol.)
- Now select “Step Down” from the “Traverse” pull-down menu. (Proceeds to the next layer in the hierarchy by highlighting all of the second layer, function symbols).
- Before grouping “Layer 2,” de-select the “Layer 1” symbol by clicking on its name in the “List” window. (This prevents the layers from becoming grouped rather than keeping them separate.)
- Continue this process until all of the layers are grouped.
- View the new graph in 2-D, horizontal, normal mode by de-selecting the “3-D,” “Compact,” and “Vertical” checkboxes in the bottom of the “Graph” window. (From this new presentation, we can now see what layers make calls in the upward direction.)

In the screen shots that follow, the kernel function hierarchies that were used at the start of this process are presented for illustration. The operating system graphs are not included since they replicate the kernel graphs with detail that is hard to display. The previous procedure was applied to all six of the operating system and kernel graphs to see if any calls are in fact made in the up direction. These six layering screen shots follow the six kernel and schedule module function hierarchy graphs.

a. Kernel Function Hierarchies

Figures 27, 28, and 29, which illustrate the Linux, OpenBSD, and Talisker kernels respectively, are examples of compact graph layouts.

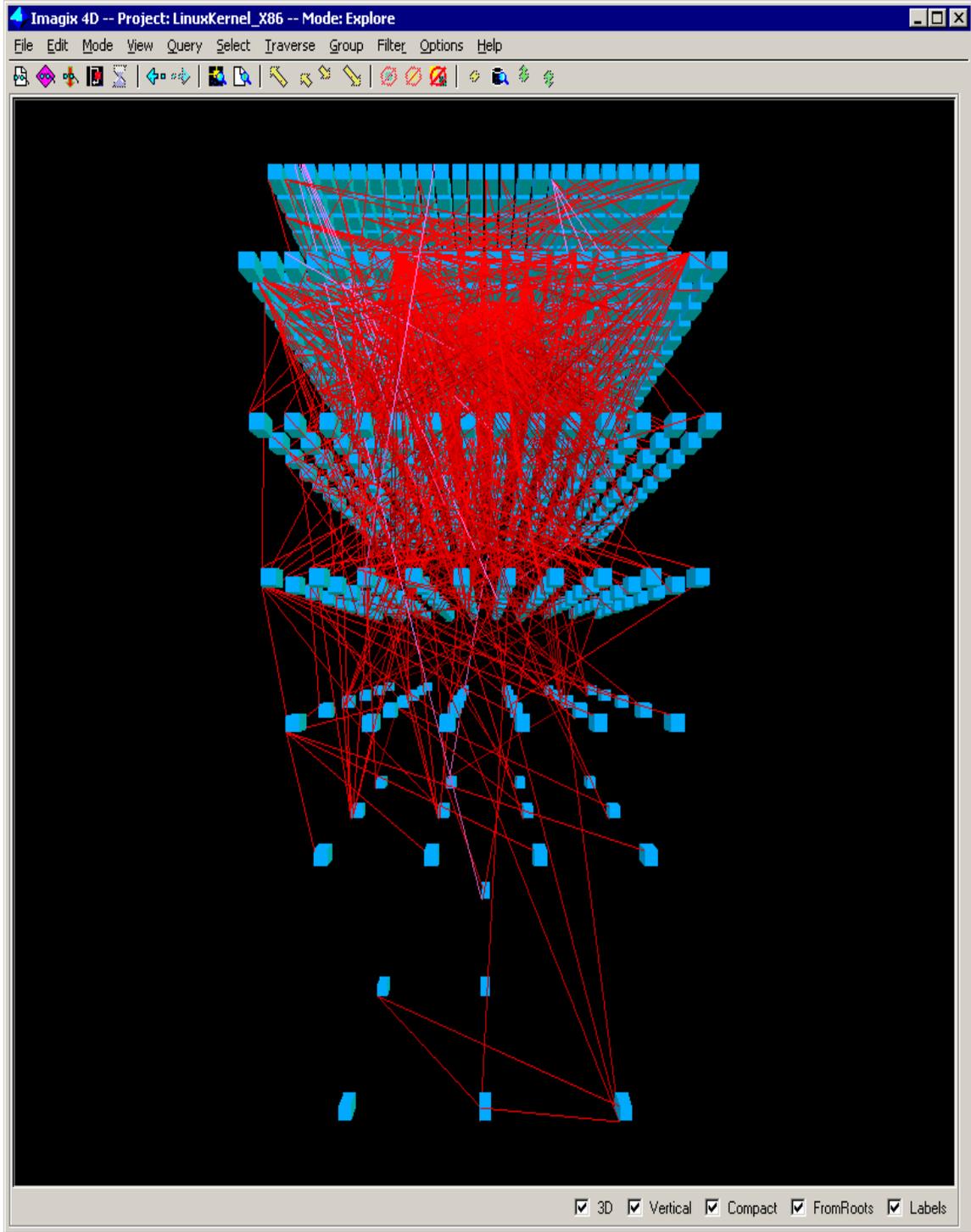


Figure 27. Linux Kernel – Function Hierarchy (Compact View).

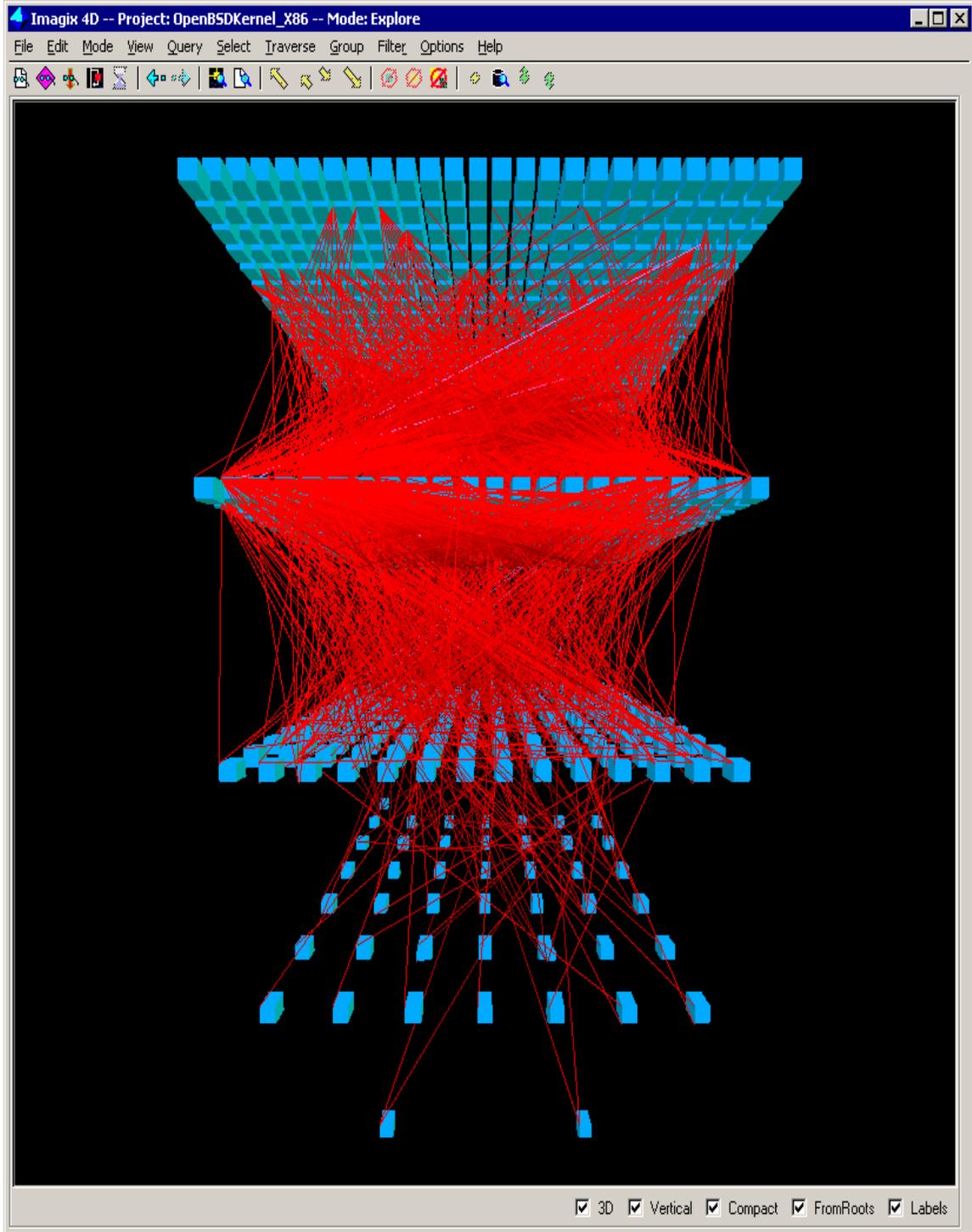


Figure 28. OpenBSD Kernel – Function Hierarchy (Compact View).

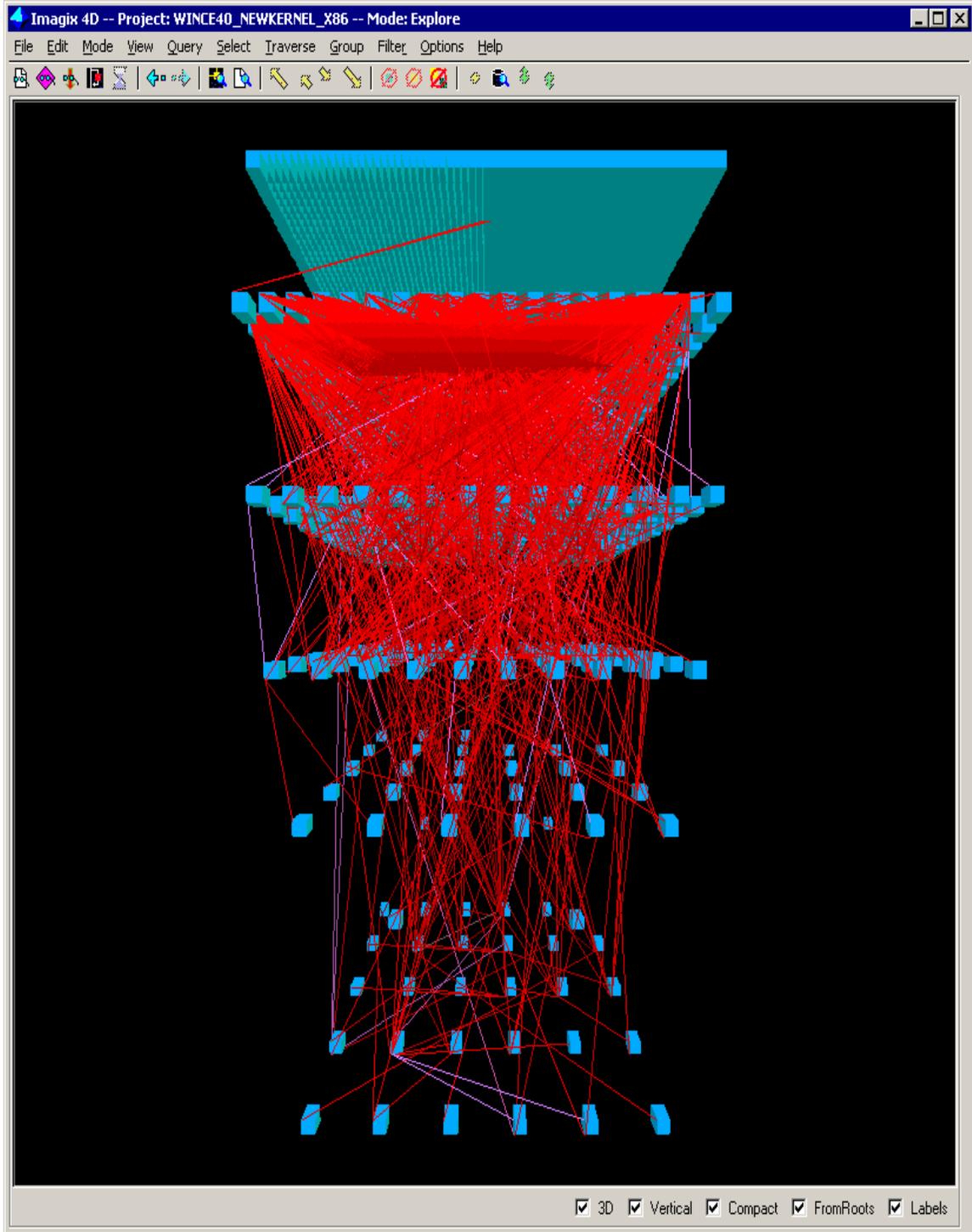


Figure 29. Talisker Kernel – Function Hierarchy (Compact View).

b. Schedule Module Function Hierarchies

Before proceeding, it is necessary to point out the naming convention for the scheduling source code used by each operating system. In Linux and Talisker, the

source code files are called “sched.c” and “schedule.c,” respectively. In contrast, OpenBSD uses “kern_synch.c” as the name of its scheduling module.

The schedule module hierarchies are shown to see the layout of the Linux, OpenBSD, and Talisker schedule source files. The functions included in each schedule module were then studied to determine whether any of these should not be included in the module. Recall from the modularity discussion that if a given module handles a plethora of different functions and manages many databases, its modularity is questionable. The process of finding upward “depends on” arrows does not apply to the schedule module functions. Only a module’s interface with the rest of the operating system is considered part of the hierarchical structuring.

In Figure 30, the Linux schedule module is shown in the compact layout with a total of five levels being visible in the hierarchy. Figure 31, which illustrates the synchronization module used in OpenBSD, and Figure 32, which shows the Talisker version of the scheduling module, also provide the same information as observed in Linux, except OpenBSD and Talisker both show four levels.

Other observations can be made about the number of functions in each schedule module regarding modularity. For instance, Linux and OpenBSD both have 17 functions that are included inside each module. From the names of the functions in Linux, such as “add_to_runqueue,” “goodness,” and “schedule,” it can be concluded that these handle process scheduling. With function names like “preempt,” “roundrobin,” and “schedcpu,” the same conclusions can be drawn for the OpenBSD module. Talisker, however, with over 143 different functions contained within its “schedule.c” module, appears to have more than just functions related to scheduling. For instance, several other functions handled in separate modules in Linux and OpenBSD are shown to be included in Talisker’s schedule module, including semaphore and mutual exclusion functions, memory paging functions, kernel and user time functions, and thread creation (forking) functions. Since the module handles a plethora of different functions and manages many databases, its modularity is questionable.

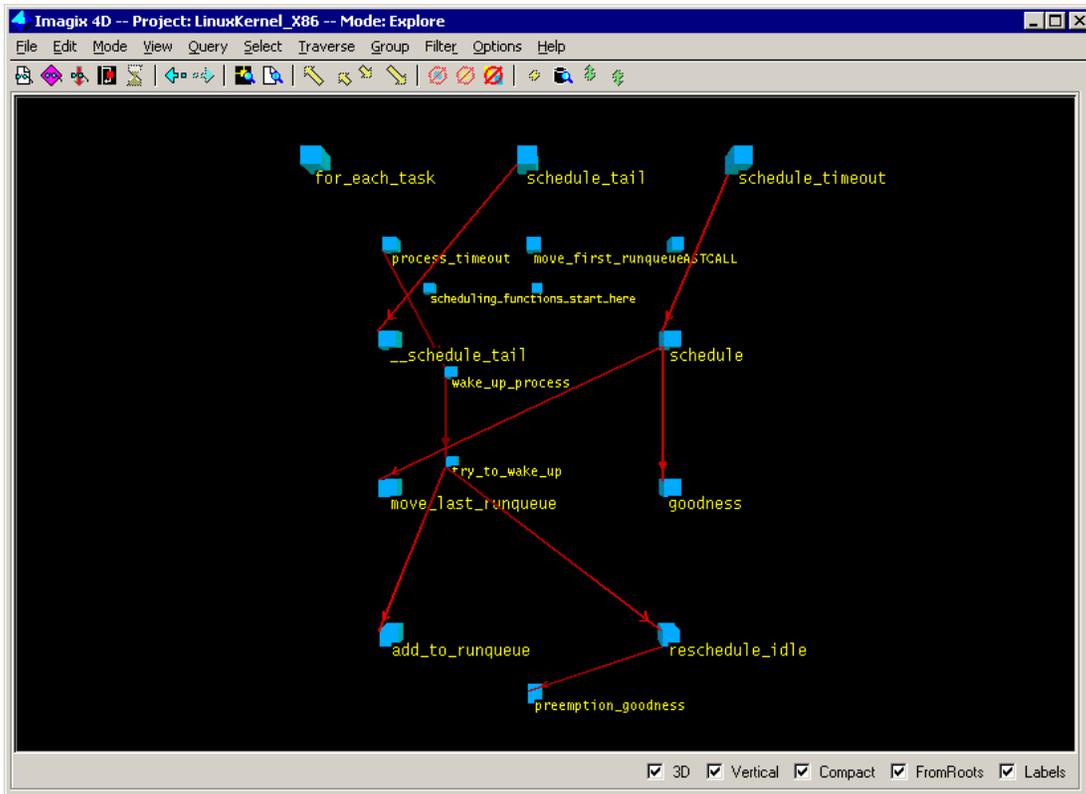


Figure 30. Linux Scheduling – Function Hierarchy (Compact View).

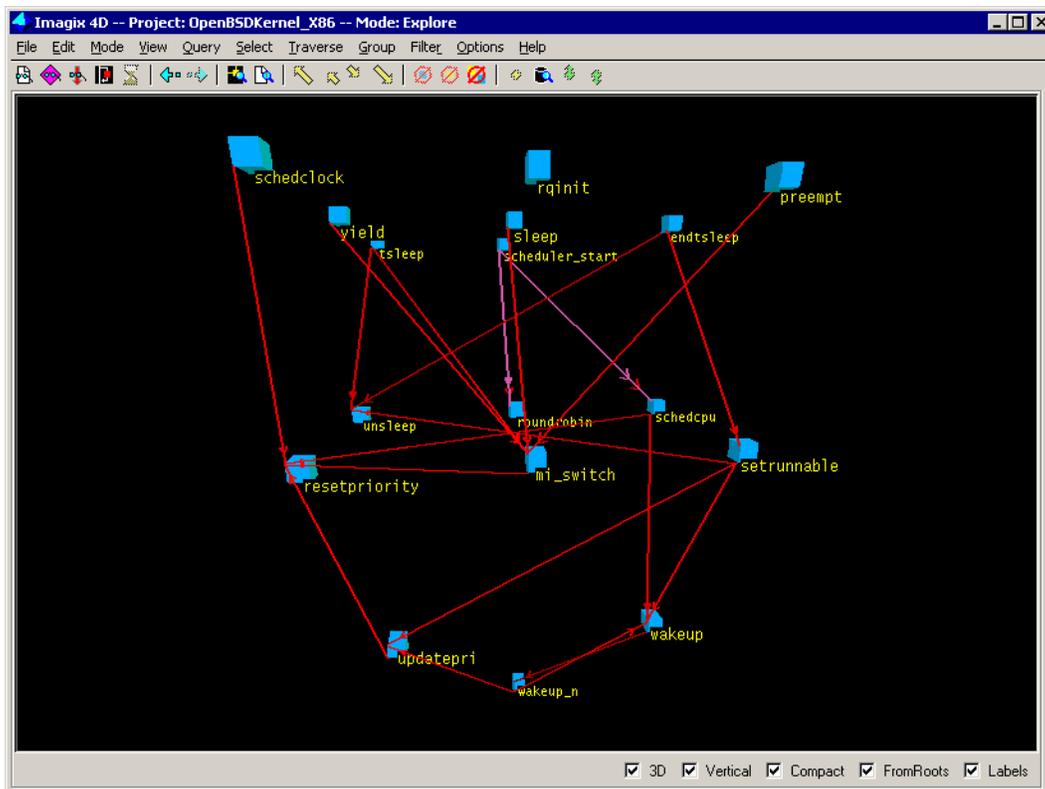


Figure 31. OpenBSD Scheduling – Function Hierarchy (Compact View).

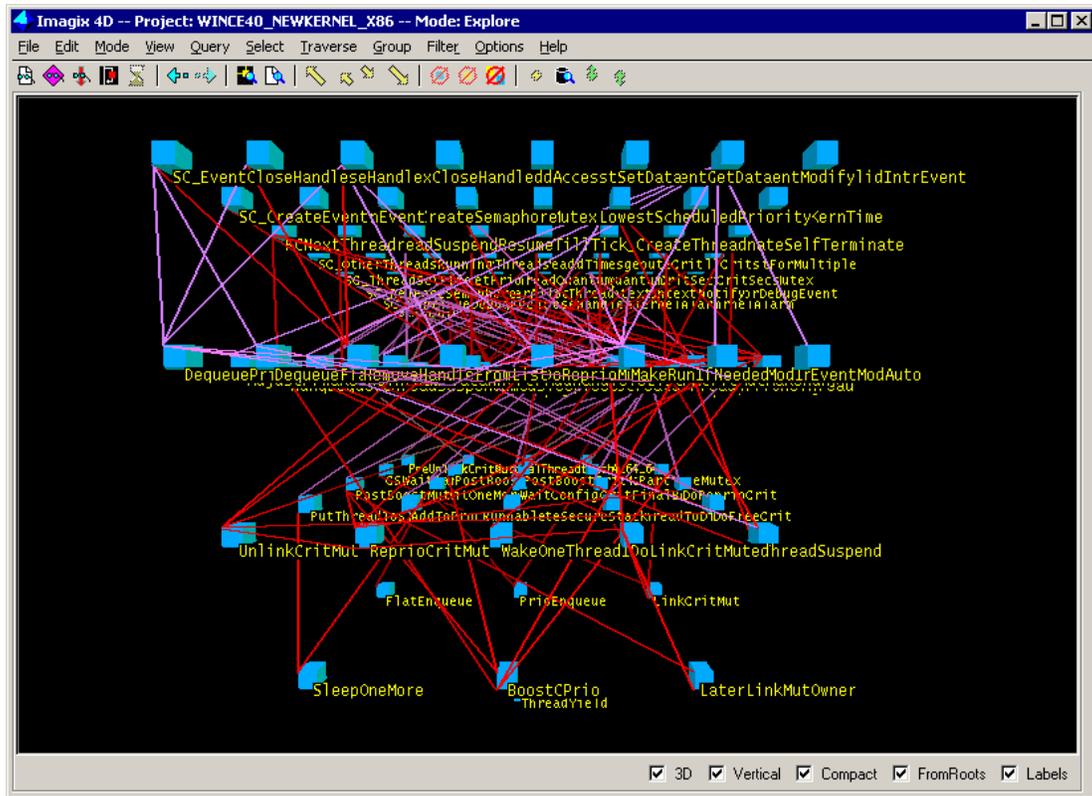


Figure 32. Talisker Scheduling – Function Hierarchy (Compact View).

c. Operating System Layering

The results from the procedure to find upward “depends on” arrows in the operating system layers are shown in the next three screen shots. In Figure 33, Linux shows eight layers in its hierarchical structuring. The red arrows represent the “calls” relations while the light purple arrows represent the “reads” relations. From the illustration, many “depends on” arrows are pointing in upward direction. This means that the lower layers are making “calls” to higher layers, which is not in agreement with partial ordering. OpenBSD, in Figure 34 with eight layers, and Talisker, in Figure 35 with nine layers, also exhibit a breakdown in partial ordering with dependencies in the upward direction. In the analysis section, the number of layers and number of upward dependencies will be used to determine which operating system most closely meets the criteria for partial ordering.

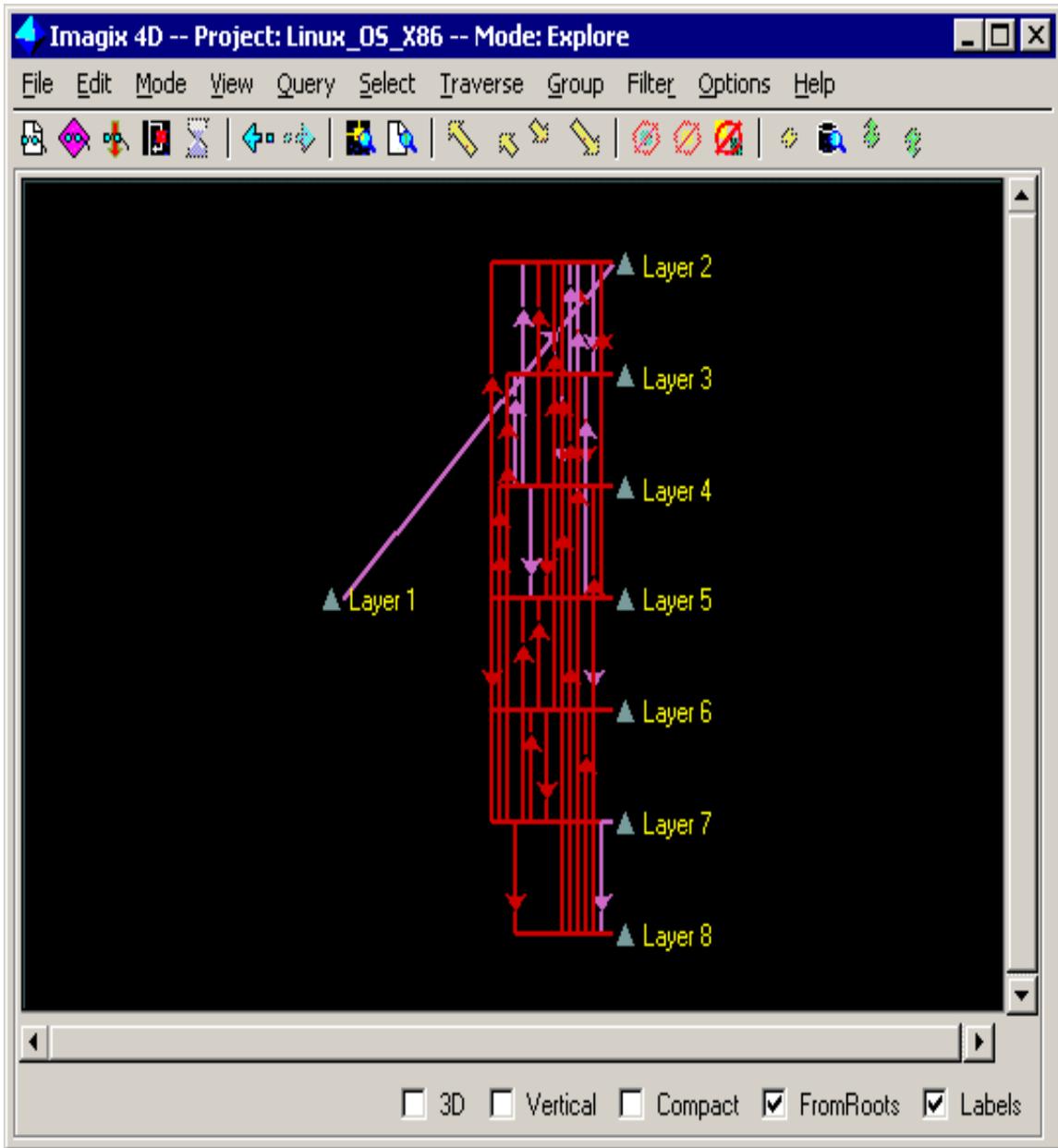


Figure 33. Linux Operating System Layering.

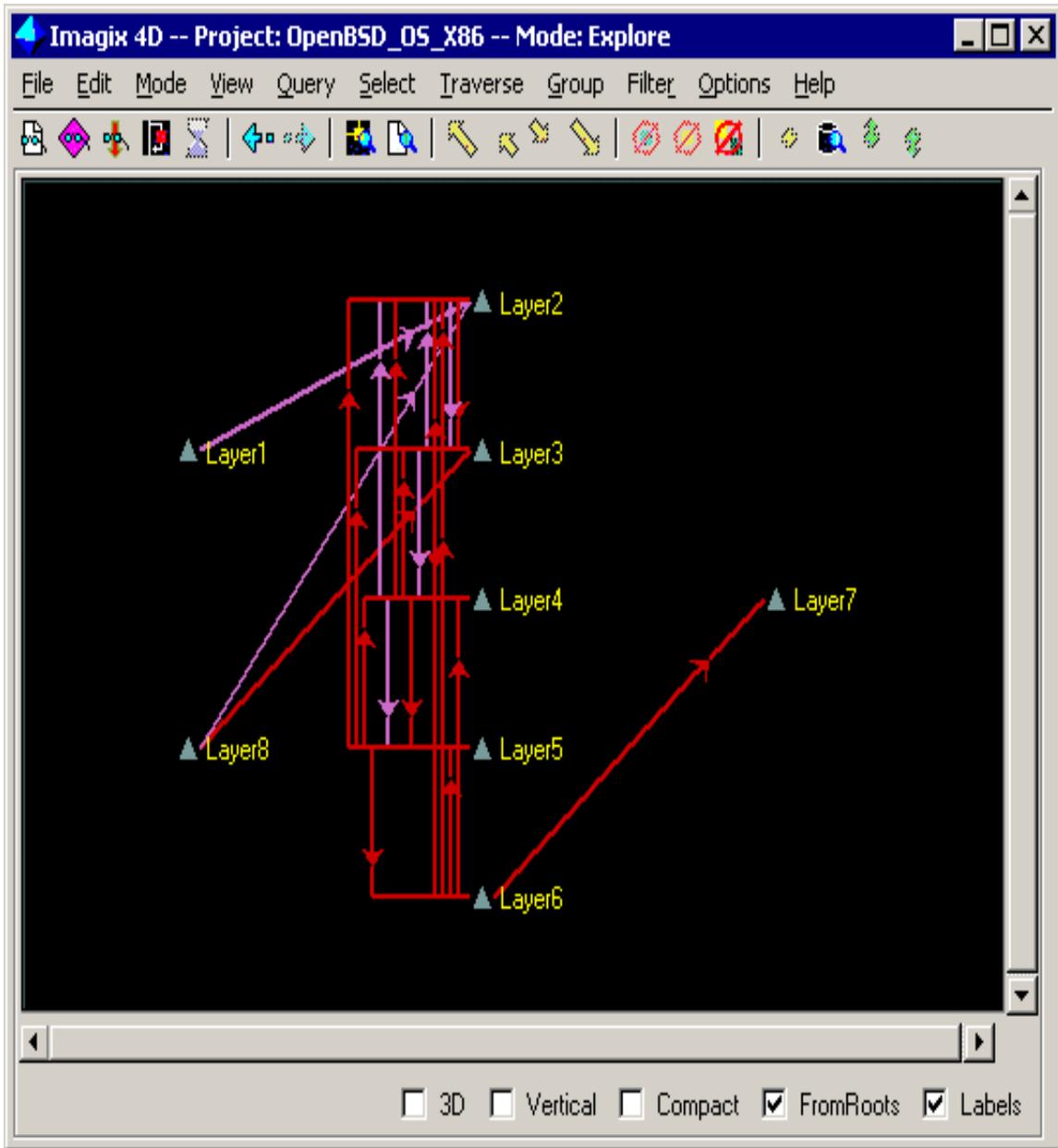


Figure 34. OpenBSD Operating System Layering.

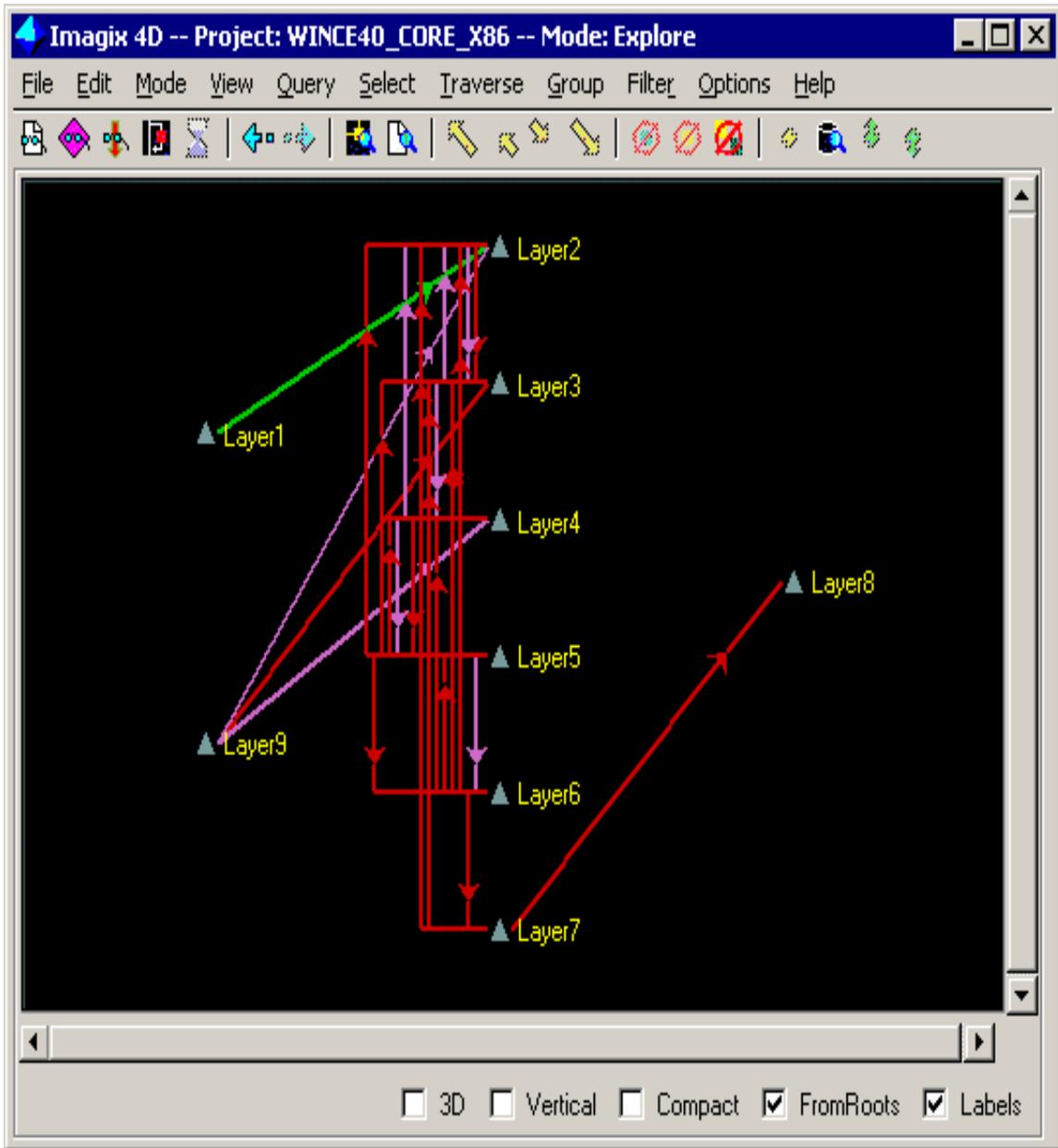


Figure 35. Talisker Operating System Layering.

d. Kernel Layering

The following three screen shots are the results from the procedure to find upward “depends on” arrows in the kernel layers. Linux, in Figure 36, shows six layers, OpenBSD, in Figure 37, has five layers, and Talisker, shown in Figure 38, displays seven layers in their kernel hierarchies. As illustrated in the operating system graphs, many “depends on” arrows are pointing in upward direction. The analysis of this data will be

presented in the next section. From visual observation, however, it appears that OpenBSD has fewer circular dependencies with Linux next in line followed by Talisker.

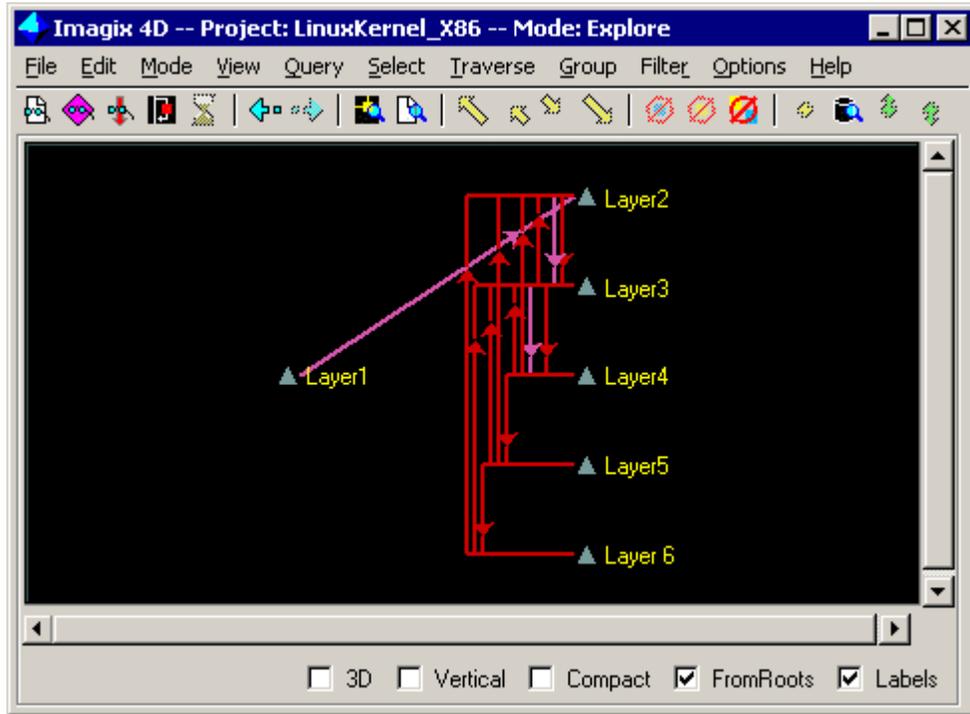


Figure 36. Linux Kernel Layering.

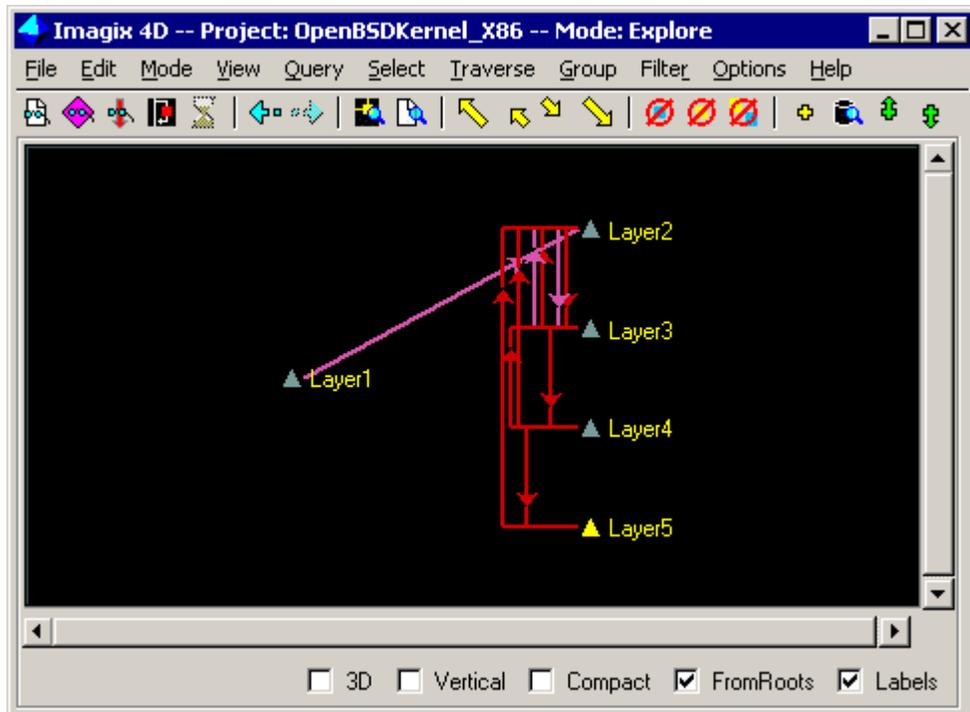


Figure 37. OpenBSD Kernel Layering.

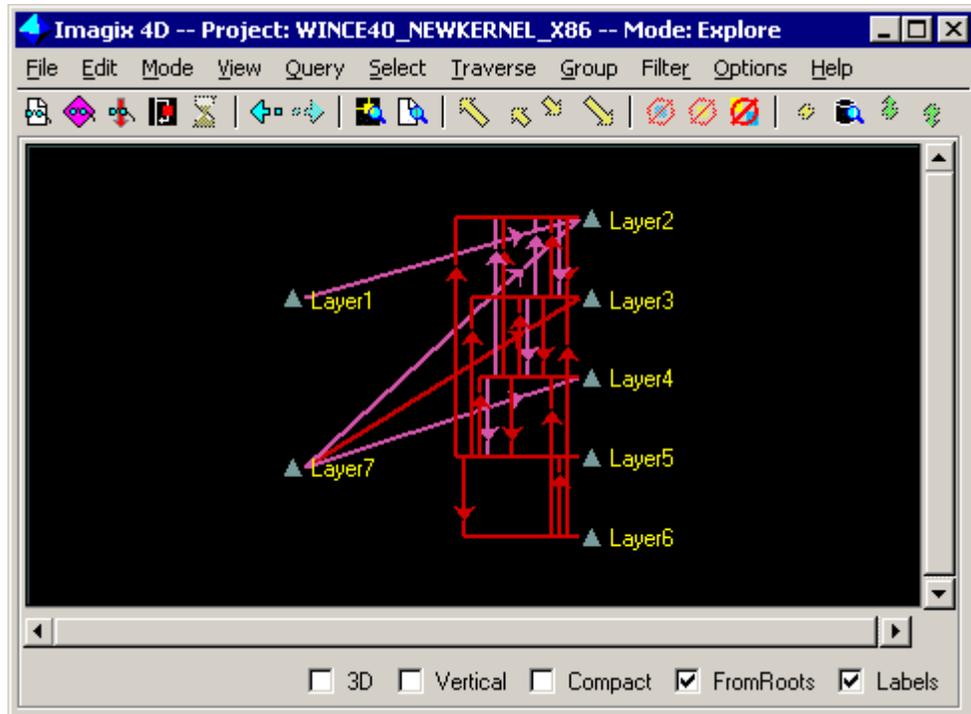


Figure 38. Talisker Kernel Layering.

2. File Summary Reports

File summary reports provide an overview of the files in the project. By tabulating these reports by type (i.e., source and header files) and by directory location, information about lines of code, lines of comments, number of functions, and the number of variables are listed. These statistics are useful in helping software designers make improvements in the implementation of their particular software package. In our comparative analysis, where software design principles are being considered, the information provided in these reports can help determine which system better satisfies modularity and information hiding.

During our data gathering process using Imagix 4D, file summary reports were generated for each of the six projects (viz, the three operating systems and the three kernels). Since the file summary report is not specific to each module, information about the three different kernel schedule modules had to be gathered from other sources, primarily the function information reports, which will be discussed in the next subsection. The following screen shots reflect a summary of the six different reports that were gathered for our analysis.

a. Operating System Files

Figures 39, 40, and 41 show the top portion of the file summary report generated for the respective Linux, OpenBSD, and Talisker operating systems.

	Total Files	KLOC	KLOCmt	KLines	Members Funcs	Macros	Vars	Types
total	802	193	102	308	4455	10763	28560	2443
C files:	409	166	82	245	4015	1137	21580	317
header files:	393	26	20	63	440	9626	6980	2126
c:/LINUX2.4.17/Linux_OS_X86/asm	0	0	0	0	0	0	0	0

Figure 39. Linux Operating System – File Summary Report.

	Total Files	KLOC	KLOCmt	KLines	Members Funcs	Macros	Vars	Types
total	744	151	94	293	3743	7434	26294	1965
C files:	468	134	68	245	3710	1177	21165	320
header files:	276	16	25	47	33	6257	5129	1645
c:/OPENBSD2.9/OpenBSD_OS_X86/adosfs	744	151	94	293	3743	7434	26294	1965

Figure 40. OpenBSD Operating System – File Summary Report.

	Total Files	KLOC	KLOCmt	KLines	Members Classes	Funcs	Macros	Vars	Types
total	161	76	57	169	17	1125	13649	11404	4443
C files:	56	29	11	52	0	972	281	4854	126
C++ files:	8	3	0	4	10	103	42	643	7
header files:	97	42	45	112	7	50	13326	5907	4310
c:/Program Files/Imagix/user/cc_cfg	161	76	57	169	17	1125	13649	11404	4443

Figure 41. Talisker Operating System – File Summary Report.

b. Kernel Files

Summary information reports on all the kernel modules for the Linux, OpenBSD, and Talisker operating systems are shown in Figures 42, 43, and 44, respectively. The parts of the reports that were used to create the summaries in the analysis section included “Total Files,” “Total KLines,” “Members Funcs”, and “Members Vars”. These measures were extracted from the source file lines only.

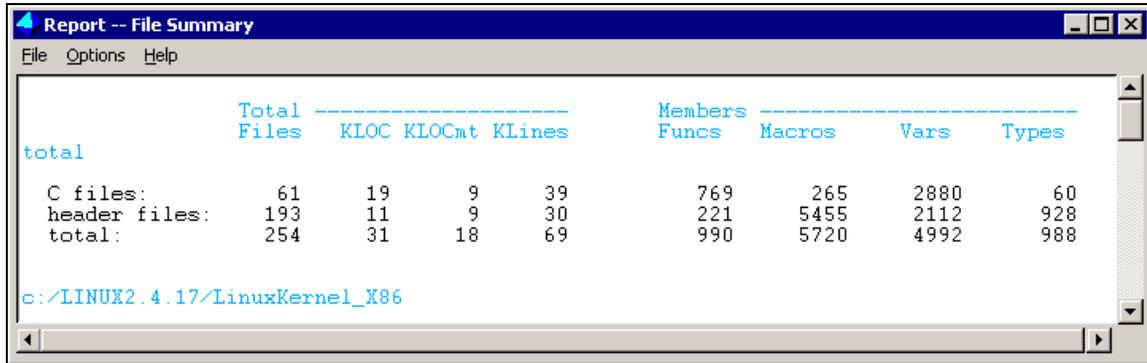


Figure 42. Linux Kernel – File Summary Report.

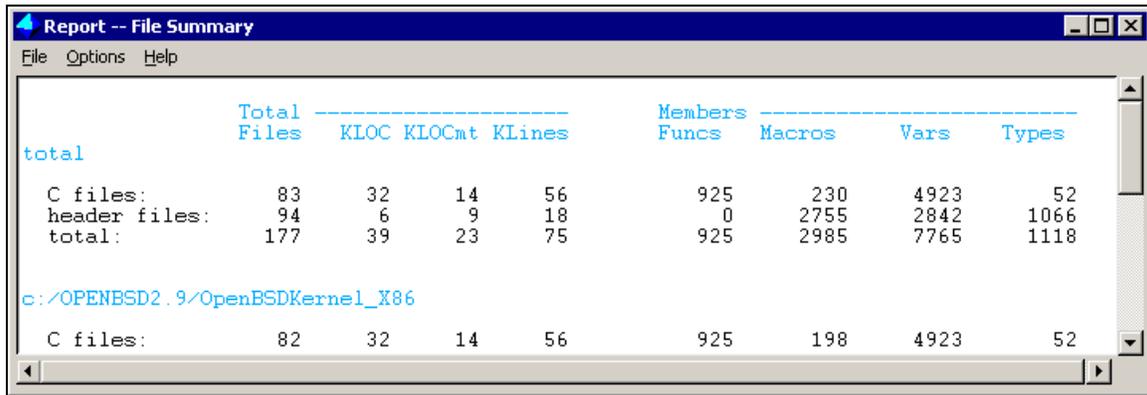


Figure 43. OpenBSD Kernel – File Summary Report.

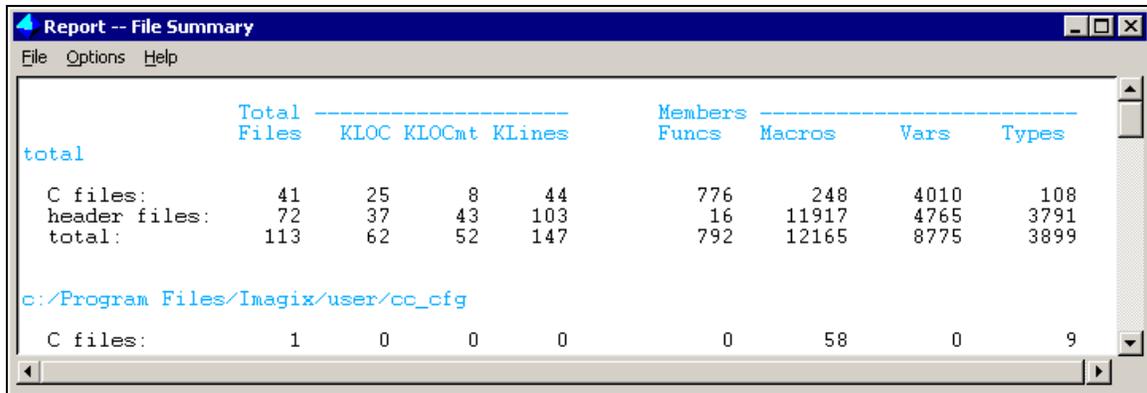


Figure 44. Talisker Kernel – File Summary Report.

3. Function Information Reports

Imagix 4D provides another type of report that lists the functions in a project, showing complexity, lines of code inside the function, total lines of code, number of callers, and the file location. The information in this report can be sorted in decreasing order using any one of these five function characteristics. Software designers can use these statistics to study the complexity of their source code during the implementation and maintenance phases of an operating system lifecycle. In our comparative analysis, the information provided in these reports can help determine which operating system is more complex in its design, and how the average number of callers per function affects our modularity comparison.

The data gathered for our study and analysis include the generation of nine different function information reports. In each report, the functions were listed in order of complexity starting from the highest measure. The other sorting options were not viewed as valuable since the reports were printed to a text file and imported into corresponding spreadsheets, where sorting could be controlled at a higher level beyond the capabilities of Imagix 4D. Using the spreadsheet application, a complexity distribution related to McCabe's threshold values for complexity (i.e., less than or equal to 10 and greater than 10) can be calculated using the spreadsheet's numbering scheme. The number of functions that have callers greater than zero can also be found by sorting these numbers in decreasing order. Taking the total number of callers and dividing by the total number of called functions can provide an average for comparison value. The first three function information reports that were generated using this process included the Linux, OpenBSD, and Talisker operating systems. The three respective operating system kernels followed, but the three scheduling module reports due to its manageable size did not require importation into corresponding spreadsheets. From the analysis that will follow in the next section, the complexity distribution for each of these sets of three will help determine which system is more complex. However, the "number of callers" data will only be shown in the analysis of the schedule modules to support an argument regarding modularity. The following nine screen shots reflect a summary of the nine different reports that were gathered for our analysis.

a. *Operating System Functions*

As stated previously, the function information reports that are shown in Figures 45, 46, and 47 give the first 20 functions of the Linux, OpenBSD, and Talisker operating systems respectively.

Function	Cmplx	LOC	Lines	Callers	File
create_elf_tables	232	80	113	1	binfmt_elf.c
balance_leaf	145	596	930	1	do_balan.c
presto_psdev_ioctl	108	761	1004	0	psdev.c
math_emulate	99	336	442	0	fpu_entry.c
do_syslog	87	116	126	4	printk.c
fldenv	80	75	89	2	reg_ld_str.c
hfs_rvret_t	79	262	334	0	file_hdr.c
load_elf_binary	77	254	393	1	binfmt_elf.c
vsprintf	76	187	226	4	vsprintf.c
parse_options	72	143	166	2	fs.c
sys_swapon	70	234	256	0	swapfile.c
inflate_blocks	65	246	267	1	zlib.c
hfs_cat_move	64	200	261	0	catalog.c
fy12x	63	161	203	1	fpu_trig.c
FPU_load_store	63	171	197	1	load_store.c
zlib_fs_inflate_blocks	62	214	223	0	infblock.c
fy12xpl	62	136	191	1	fpu_trig.c
jifs_scan_flash	59	334	523	1	intrep.c
ufs_read_super	58	287	374	0	super.c
do_fprem	58	200	259	2	fpu_trig.c

Figure 45. Linux OS – Function Information Sorted By Complexity.

Function	Cmplx	LOC	Lines	Callers	File
nfsrv_create	208	230	237	0	nfs_serv.c
nfs_create	201	90	100	1	nfs_vnops.c
nfs_mknodrpc	196	83	83	2	nfs_vnops.c
nfs_symlink	188	54	62	1	nfs_vnops.c
nfs_mkdir	183	69	77	1	nfs_vnops.c
kprintf	171	307	476	3	subr_prf.c
vm_fault	164	301	746	6	vm_fault.c
nfsrv_readdir	164	226	260	0	nfs_serv.c
nfsrv_mkdir	153	110	110	0	nfs_serv.c
nfsrv_mknod	149	141	145	0	nfs_serv.c
nfsrv_setattr	148	105	126	0	nfs_serv.c
nfsrv_readdirplus	148	241	293	0	nfs_serv.c
sput	137	447	487	2	pcccons.c
nfs_setattrrpc	137	48	48	2	nfs_vnops.c
nfs_readdirplusrpc	130	189	221	1	nfs_vnops.c
math_emulate	126	404	410	0	math_emulate.c
nfsrv_writegather	121	258	301	0	nfs_serv.c
ttyinput	118	261	352	1	tty.c
soreceive	117	266	313	5	uipc_socket.c
nfs_lookup	116	136	151	1	nfs_vnops.c

Figure 46. OpenBSD OS – Function Information Sorted By Complexity.

Function	Cmplx	LOC	Lines	Callers	File
NKvvsprintfW	146	202	223	5	printf.c
GPE::EmulatedBlt	138	586	681	1	swblt.cpp
SC_CreateFileMapping	100	363	503	1	mapfile.c
LoadOneLibraryPart2	92	315	385	4	loader.c
GPE::EmulatedLine	75	217	258	0	swline.cpp
DrvStrokePath	65	353	616	1	drvstrok.cpp
DoVirtualAlloc	64	239	317	5	virtmem.c
SafeOpenExe	57	255	271	2	loader.c
WaitOneMore	56	169	171	1	schedule.c
FlushMapBuffersLogged	53	213	345	4	mapfile.c
PageInModule	50	171	197	1	loader.c
KCNextThread	50	170	243	0	schedule.c
SC_CreateProc	49	210	228	1	schedule.c
CreateNewProc	49	187	198	2	loader.c
ExceptionDispatch	48	169	206	0	mdx86.c
GetProcessAndThreadInfo	47	274	338	1	dbg.c
DoLockPages	42	133	165	1	virtmem.c
LoadO32	39	115	139	2	loader.c
RegisterDeviceEx	37	138	196	2	device.c
MappedPageIn	37	141	174	0	mapfile.c

Figure 47. Talisker OS – Function Information Sorted By Complexity.

From these three function information reports, one can conclude that OpenBSD’s complexity is far above the McCabe threshold values with measures in the hundreds. In comparison, Talisker shown in Figure 47 appears to have a better handle on complexity, although it surpasses the threshold McCabe considers un-testable and high risk. Linux falls somewhere in between of these two. An analysis of the data gathered will be presented in the next section.

b. Kernel Functions

After inspection of Figures 48, 49, and 50 of the Linux, OpenBSD, and Talisker kernels respectively, all three appear to have nearly the same level of complexity at first glance. If one was to order these from best to worst, it appears that Linux would be first, followed by Talisker, and then OpenBSD. Trying to figure out which system is more complex using only the function reports is difficult. As mentioned earlier, this data needs to be imported into a spreadsheet and studied further. The results of this procedure will be referenced in the summary tables included in the analysis section. The parts of the reports that were used to create the summaries included the “Cmplx” values for complexity. In the schedule module analysis, however, the “Lines” and “Callers” metrics were included in the summaries to aid in the modularity analysis.

Function	Cmplx	LOC	Lines	Callers	File
setup_sigcontext	135	36	41	2	signal.c
restore_sigcontext	89	38	59	2	signal.c
setup_rt_frame	88	70	74	1	signal.c
do_syslog	87	116	126	1	printk.c
copy_siginfo_to_user	54	26	34	1	signal.c
sys_sigaction	49	27	30	0	signal.c
setup_frame	49	58	69	1	signal.c
do_signal	38	91	136	0	signal.c
do_fork	38	121	190	4	fork.c
sys_ptrace	32	62	75	0	ptrace.c
sys_olduname	32	22	28	0	sys_i386.c
smp_boot_cpus	32	130	242	0	smpboot.c
sys_ipc	31	72	73	0	sys_i386.c
MP_processor_info	31	89	102	3	mpparse.c
mtrr_add_page	30	119	158	1	mtrr.c
handle_vm86_fault	27	72	110	2	vm86.c
check_events	27	73	97	1	apm.c
apm	27	84	120	1	apm.c
second_overflow	24	72	113	1	timer.c
setup_ioapic_ids_from_mpc	23	58	89	1	io_apic.c

Figure 48. Linux Kernel – Function Information Sorted By Complexity.

Function	Cmplx	LOC	Lines	Callers	File
kprintf	171	307	476	3	subr_prf.c
ttyinput	118	261	352	1	tty.c
soreceive	117	266	313	2	uipc_socket.c
ttioctl	80	257	300	1	tty.c
userconf_parse	76	156	165	1	subr_userconf.c
uipc_usrreq	62	189	238	0	uipc_usrreq.c
sosend	62	141	163	2	uipc_socket.c
sys_execve	60	253	464	2	kern_exec.c
lookup	57	201	316	1	vfs_lookup.c
lockmgr	54	172	268	7	kern_lock.c
psignal	47	94	196	23	kern_sig.c
fork1	47	158	323	4	kern_fork.c
kqueue_scan	46	118	125	1	kern_event.c
kern_sysctl	46	124	152	0	kern_sysctl.c
tthread	44	127	185	0	tty.c
sys__semctl	43	139	161	0	sysv_sem.c
sys__osemctl	43	139	161	0	sysv_sem.c
sosetopt	43	112	124	1	uipc_socket.c
sys_semop	42	132	267	0	sysv_sem.c
sys_ptrace	42	117	302	1	sys_process.c

Figure 49. OpenBSD Kernel – Function Information Sorted By Complexity.

Function	Cmplx	LOC	Lines	Callers	File
NKwvsprintfW	146	202	223	5	printf.c
SC_CreateFileMapping	100	363	503	1	mapfile.c
LoadOneLibraryPart2	92	315	385	4	loader.c
DoVirtualAlloc	64	239	317	5	virtmem.c
SafeOpenExe	57	255	271	2	loader.c
WaitOneMore	56	169	171	1	schedule.c
FlushMapBuffersLogged	53	213	345	4	mapfile.c
PageInModule	50	171	197	1	loader.c
KCNextThread	50	170	243	0	schedule.c
SC_CreateProc	49	210	228	1	schedule.c
CreateNewProc	49	187	198	2	loader.c
ExceptionDispatch	48	169	206	0	mdx86.c
GetProcessAndThreadInfo	47	274	338	1	dbg.c
DoLockPages	42	133	165	1	virtmem.c
Load032	39	115	139	2	loader.c
MappedPageIn	37	141	174	0	mapfile.c
KdpAddBreakpoint	37	174	270	3	kdbreak.c
KdpTrap	35	167	314	2	kdtrap.c
KdpSendWaitContinue	35	176	258	3	kdapi.c
ProfilerReport	34	162	186	0	profile.c

Figure 50. Talisker Kernel – Function Information Sorted By Complexity.

c. Schedule Module Functions

The Linux, OpenBSD, and Talisker function reports for each schedule module are displayed in Figures 51, 52, and 53, respectively. As stated earlier, the “Cmplx,” “Lines,” and “Callers” data was included in the summary tables created and presented in the analysis section.

Function	Cmplx	LOC	Lines	Callers	File
schedule	13	45	67	16	sched.c
goodness	8	19	51	2	sched.c
try_to_wake_up	7	16	20	1	sched.c
schedule_timeout	6	31	51	2	sched.c
reschedule_idle	2	8	105	1	sched.c
wake_up_process	1	4	4	3	sched.c
scheduling_functions_star	1	1	1	0	sched.c
schedule_tail	1	4	4	0	sched.c
process_timeout	1	5	6	1	sched.c
preemption_goodness	1	4	4	1	sched.c
move_last_runqueue	1	5	5	1	sched.c
move_first_runqueue	1	5	5	0	sched.c
add_to_runqueue	1	5	5	1	sched.c
__schedule_tail	1	4	66	1	sched.c
for_each_task	-	-	-	0	sched.c
LIST_HEAD	-	-	-	0	sched.c
FASTCALL	-	-	-	0	sched.c

Figure 51. Linux Kernel Schedule Module – Function Information Report.

Function	Cmplx	LOC	Lines	Callers	File
tsleep	18	64	103	44	kern_synch.c
setrunnable	15	34	39	5	kern_synch.c
wakeup_n	12	39	53	1	kern_synch.c
mi_switch	9	38	62	5	kern_synch.c
schedcpu	7	45	73	1	kern_synch.c
updatepri	4	16	16	2	kern_synch.c
unsleep	4	19	19	5	kern_synch.c
sleep	4	31	60	1	kern_synch.c
roundrobin	4	19	24	1	kern_synch.c
endtsleep	4	17	17	2	kern_synch.c
schedclock	2	9	8	1	kern_synch.c
rqinit	2	7	7	1	kern_synch.c
resetpriority	2	11	11	5	kern_synch.c
preempt	2	15	19	1	kern_synch.c
yield	1	12	12	0	kern_synch.c
wakeup	1	6	5	50	kern_synch.c
scheduler_start	1	10	18	1	kern_synch.c

Figure 52. OpenBSD Schedule Module – Function Information Report.

Function	Cmplx	LOC	Lines	Callers	File
WaitOneMore	56	169	171	1	schedule.c
KCNextThread	50	170	243	0	schedule.c
SC_CreateProc	49	210	228	1	schedule.c
SC_NKTerminateThread	28	146	160	2	schedule.c
RunqDequeue	28	91	111	5	schedule.c
SetThreadToDie	25	91	111	1	schedule.c
SC_WaitForMultiple	25	115	118	8	schedule.c
ThreadSuspend	23	68	87	5	schedule.c
FinishRemoveThread	20	78	91	2	schedule.c
SC_EventModify	19	62	66	2	schedule.c
DoCreateThread	19	98	136	4	schedule.c
SystemStartupFunc	18	90	117	1	schedule.c
SC_CreateSemaphore	18	77	80	1	schedule.c
SC_CreateMutex	18	77	81	1	schedule.c
PrioEnqueue	18	56	58	2	schedule.c
OpenOrCreateEvent	18	80	88	2	schedule.c
NextThread	18	90	111	0	schedule.c
SC_ProcDebug	17	92	99	1	schedule.c

Figure 53. Talisker Kernel Schedule Module – Function Information Report.

From these function reports, Talisker in Figure 53 is the most complex regarding the cyclomatic complexity and lines of code; in addition, it has the most number of functions included inside its schedule module equaling 143. As will be shown in the analysis section, the average number of callers per function result will support the argument that Talisker’s schedule module interface is not well defined. In comparison, Linux and OpenBSD have the same number of functions, but Linux is the least complex regarding cyclomatic complexity and lines of code. From a brief glance at the number of

callers, OpenBSD has a higher number of callers than Linux. The actual numbers will be reviewed in the analysis regarding modularity.

4. Global Variable Graphs

In this portion of the data gathering process of the experiment, the “Explore” mode features of the Imagix 4D tool were used to help find all of the global variables in the source code found in each of the six projects as well as the three schedule modules. Recall that the definition of a global variable, as observed from using Imagix 4D, is any variable that is shared by two or more modules (source files) in the system. If a global variable exists between modules, then the principle of information hiding is no longer valid in the design, thereby affecting its modularity. To assist those designing, implementing, and maintaining an operating system over its lifecycle, the Imagix 4D tool can help determine if global variables are ever introduced.

The process that was used to find the global variables is straightforward. For instance, while in the “Browse” mode in one of the six Imagix 4D projects, the “Variables” checkbox in the bottom left hand corner of the graph window should be the only one selected. Then one needs to proceed to the “Explore” mode and select from the “Filter” pull-down menu the “Restore All” selection, which displays all of the variables stored in the database. The Graph window will show a forest of green variable symbols, and the List window will correspond to each variable listing its name and location. The “All” function from the “Select” pull-down menu then needs to be selected followed by the “Find” function found in the same pull-down menu. This command will present a “Find” pop-up window where a type “Program Element” and subtype “Variable” need to be selected from the appropriate scroll-selection boxes. The final step involves clicking on the “Options” button of this pop-up window, which displays another window with different find query options. From the “Scope” option, select “Global” from the list of choices and then click on the “OK” button followed by the “Find” button. In the Graph and List windows, only the global variables will be highlighted as a result of this operation. From the List window, the total number of global variables can be summed.

The following six screen shots show the results of this process for each of the three kernel projects and the three schedule modules. The three operating systems were also considered in this experiment, but the screen shots were not included due to the

problem of properly viewing such a large number of variables. For the schedule modules, the only change in procedure is to select the corresponding schedule module (i.e., “sched.c” for Linux, “kern_synch.c” for OpenBSD, and “schedule.c” for Talisker) from the “File Browser” while either in the “Browse” or “Explore” modes of Imagix 4D rather than restoring all the variables. The analysis section that follows will summarize the information gathered for the operating systems, the kernels, and the schedule modules.

a. Kernel Global Variables

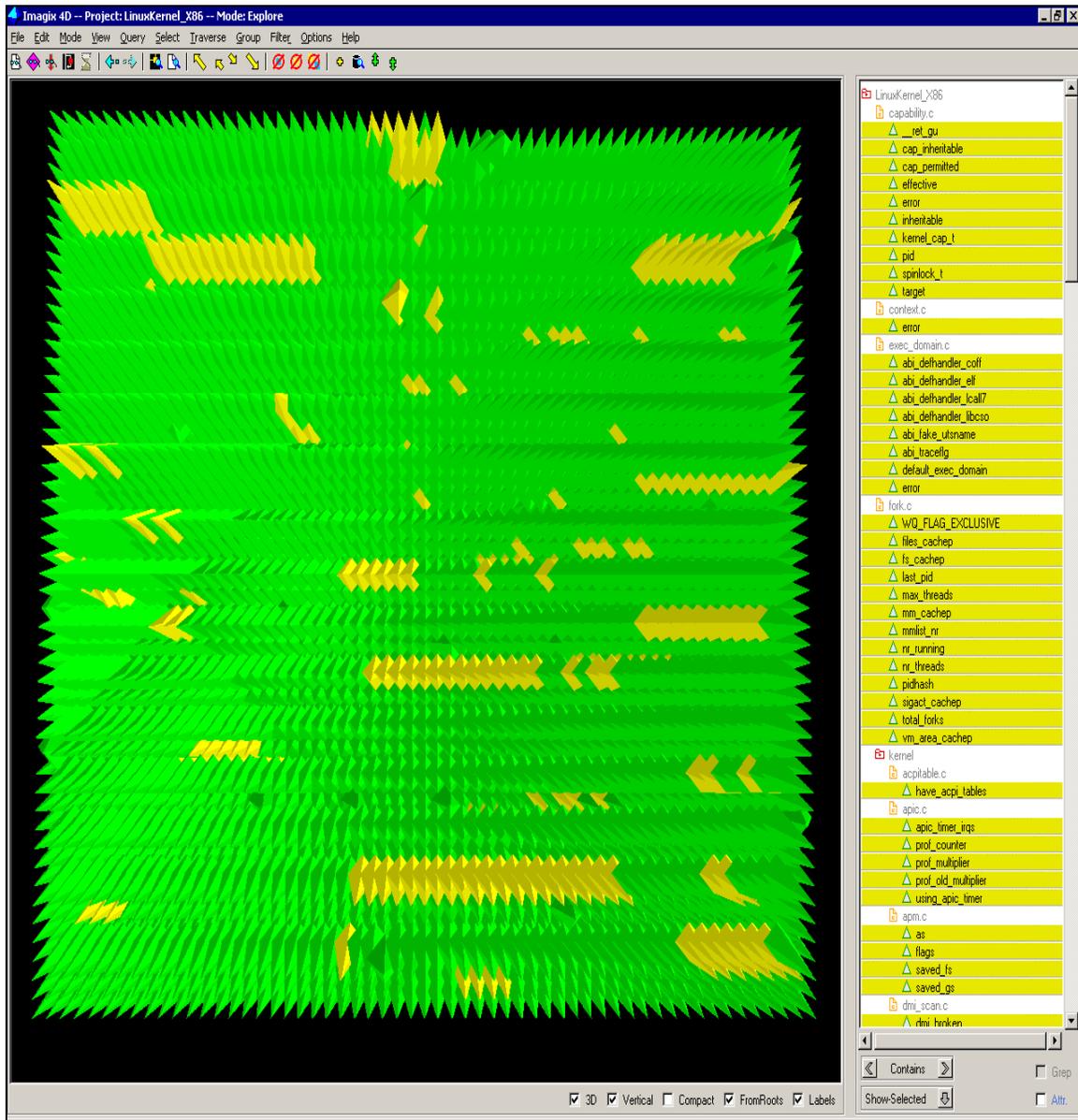


Figure 54. Linux Kernel – Global Variables.

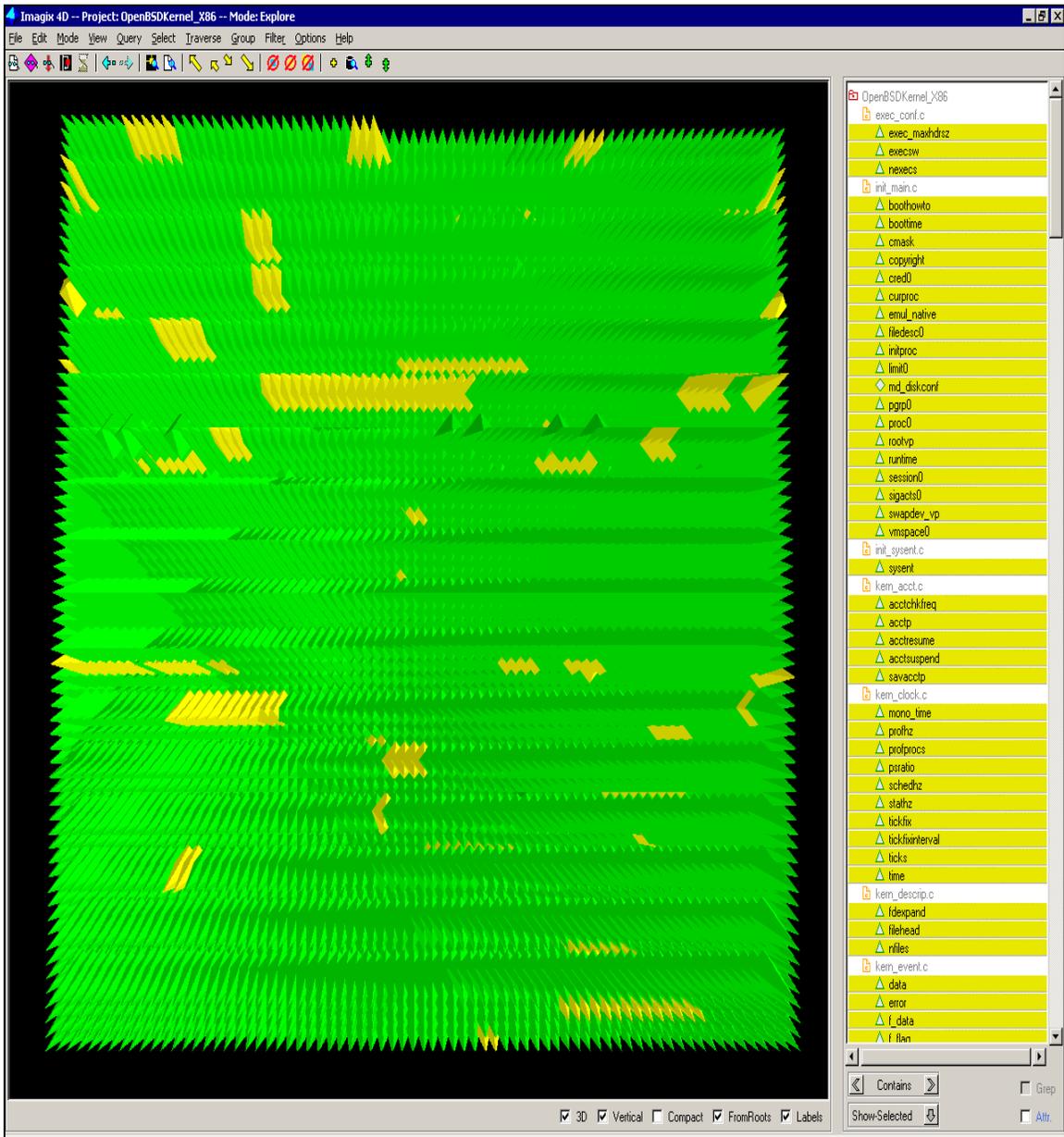


Figure 55. OpenBSD Kernel – Global Variables.

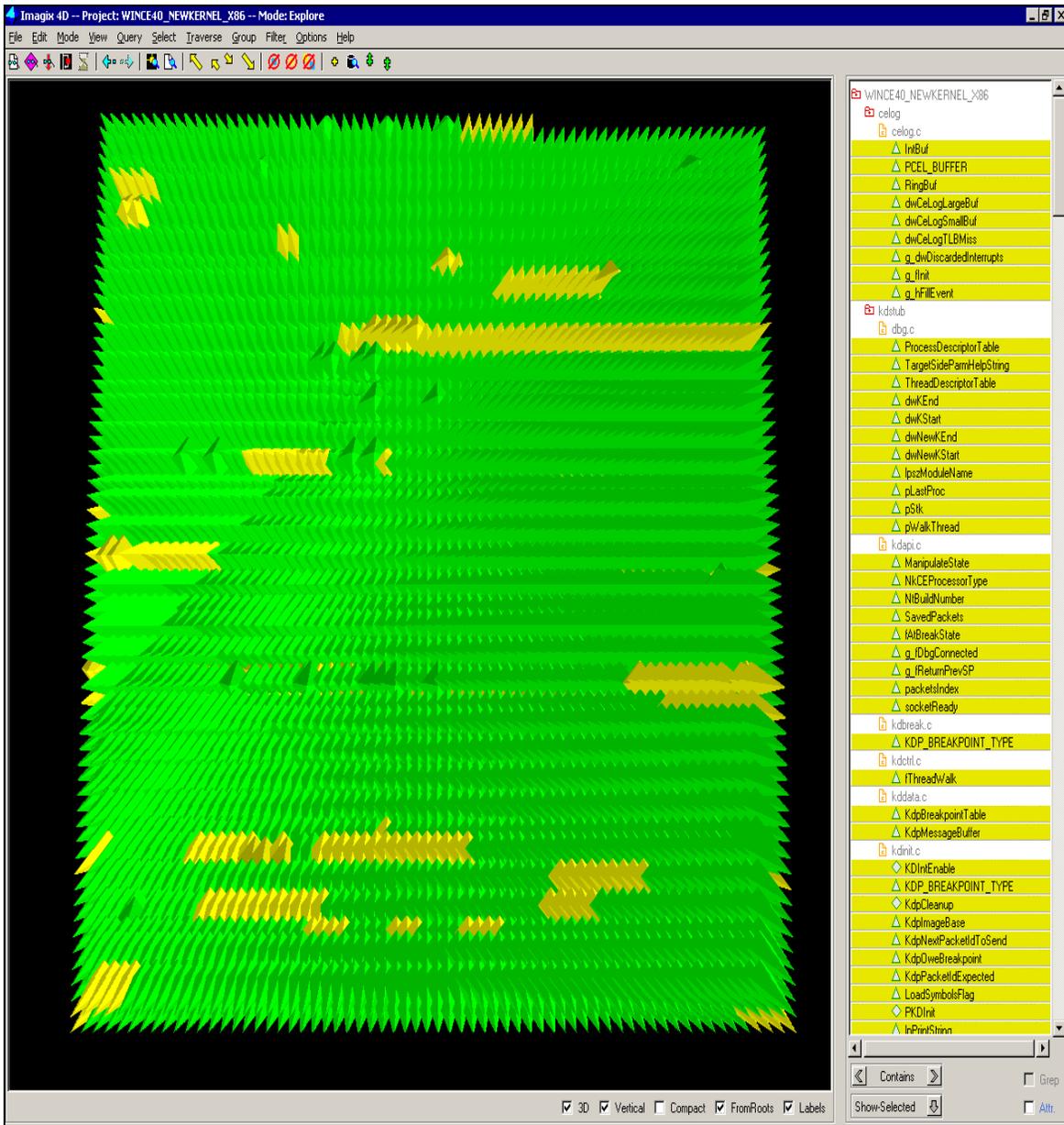


Figure 56. Talisker Kernel – Global Variables.

From the inspection of Figure 54, which shows the global variables of the Linux kernel highlighted in the List window and the Graph window, it looks visually apparent that Linux as well as the OpenBSD and Talisker systems shown in Figures 55 and 56 all have a considerable number of global variables.

b. Schedule Module Global Variables

The next three screen shots show all of the variables included in each of the three operating system's schedule modules. From these variables, the procedure discussed earlier was used to find all of the global variables. For instance, the Linux schedule module, shown in Figure 57, has seven global variables out of its total of 46 variables. In comparison, OpenBSD as shown in Figure 58 only has five global variables among its list of 67. Talisker, in Figure 59, has a total of 79 global variables out of 776.

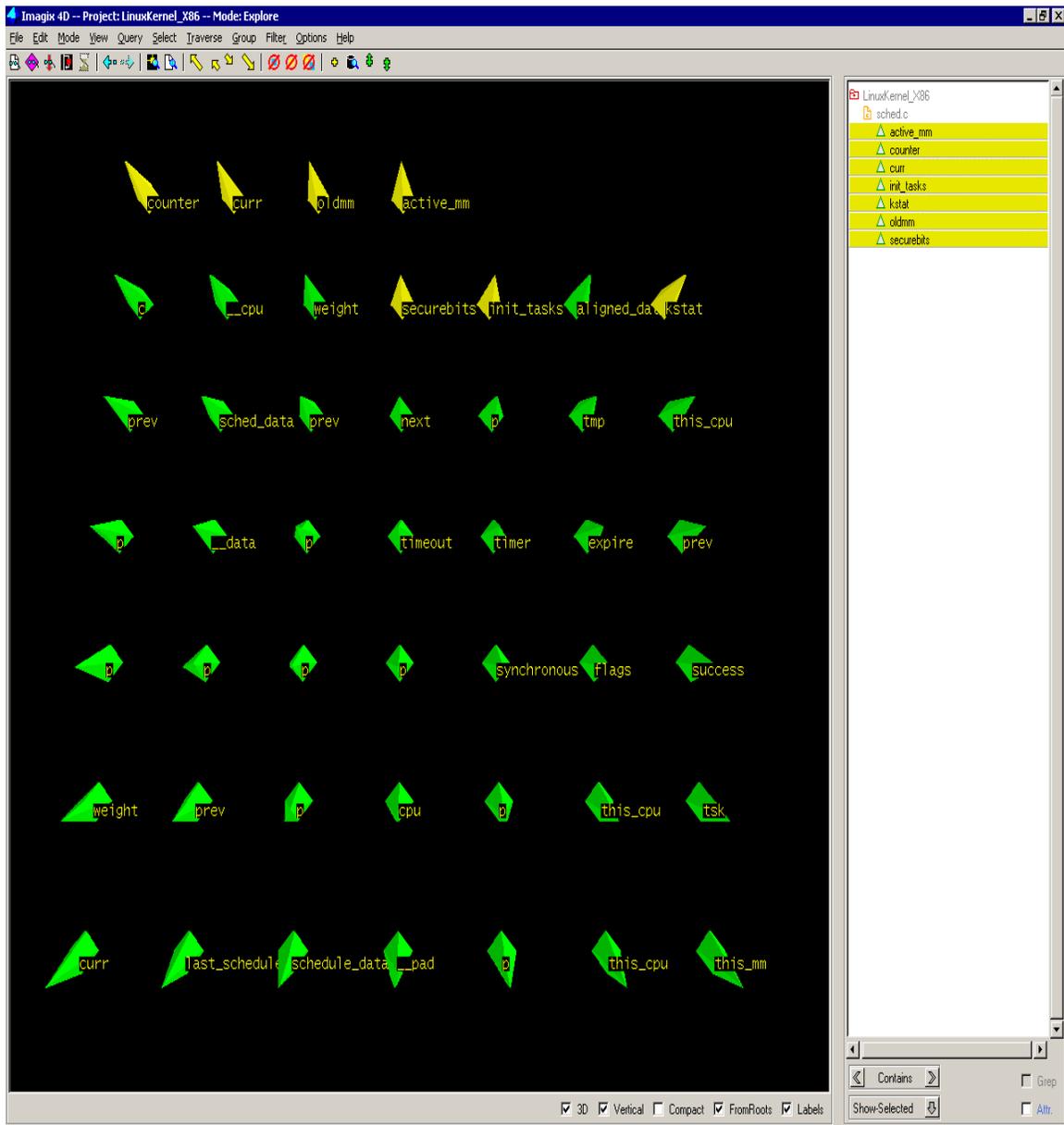


Figure 57. Linux Kernel Schedule Module – Global Variables.

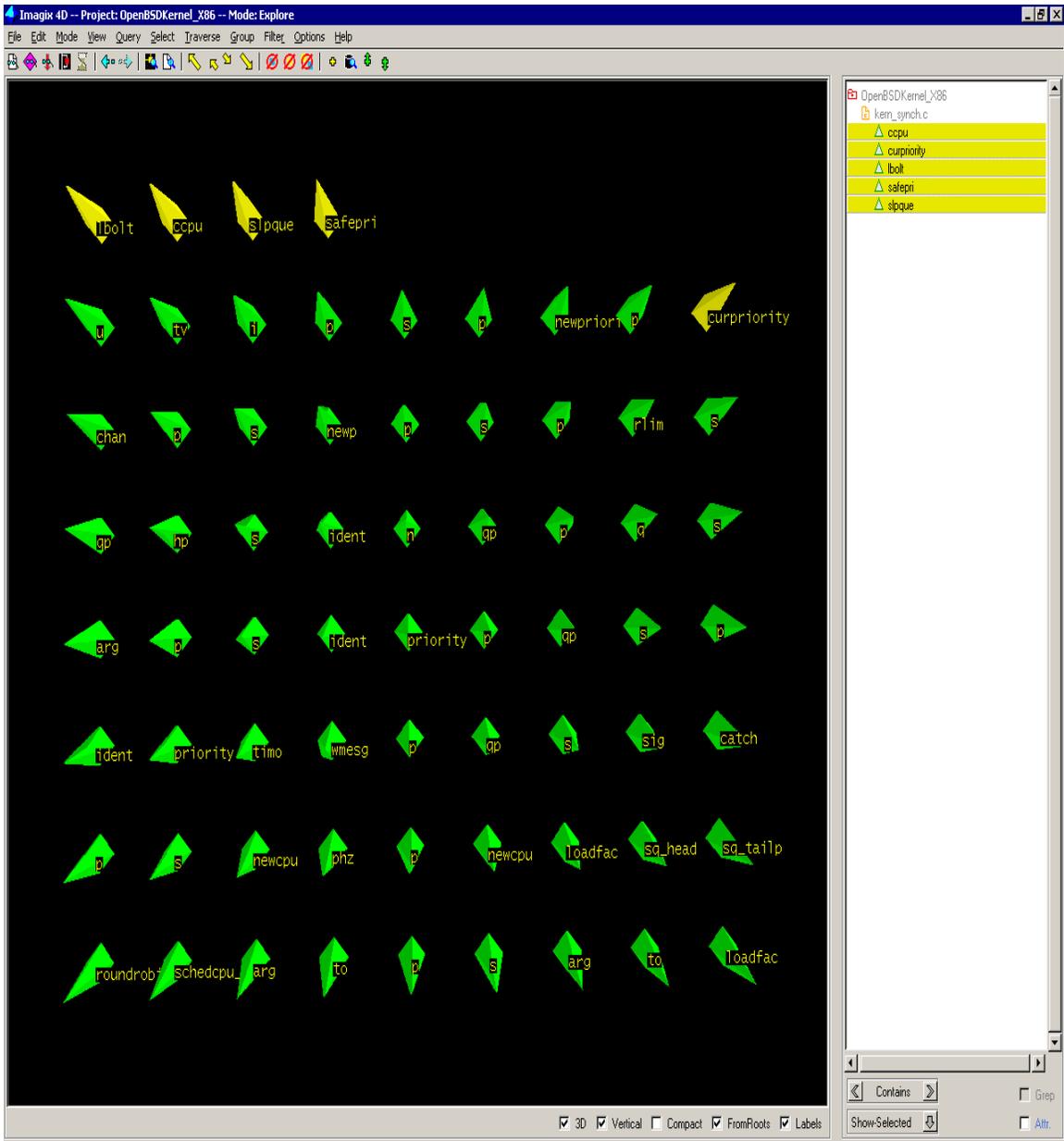


Figure 58. OpenBSD Kernel Schedule Module – Global Variables.

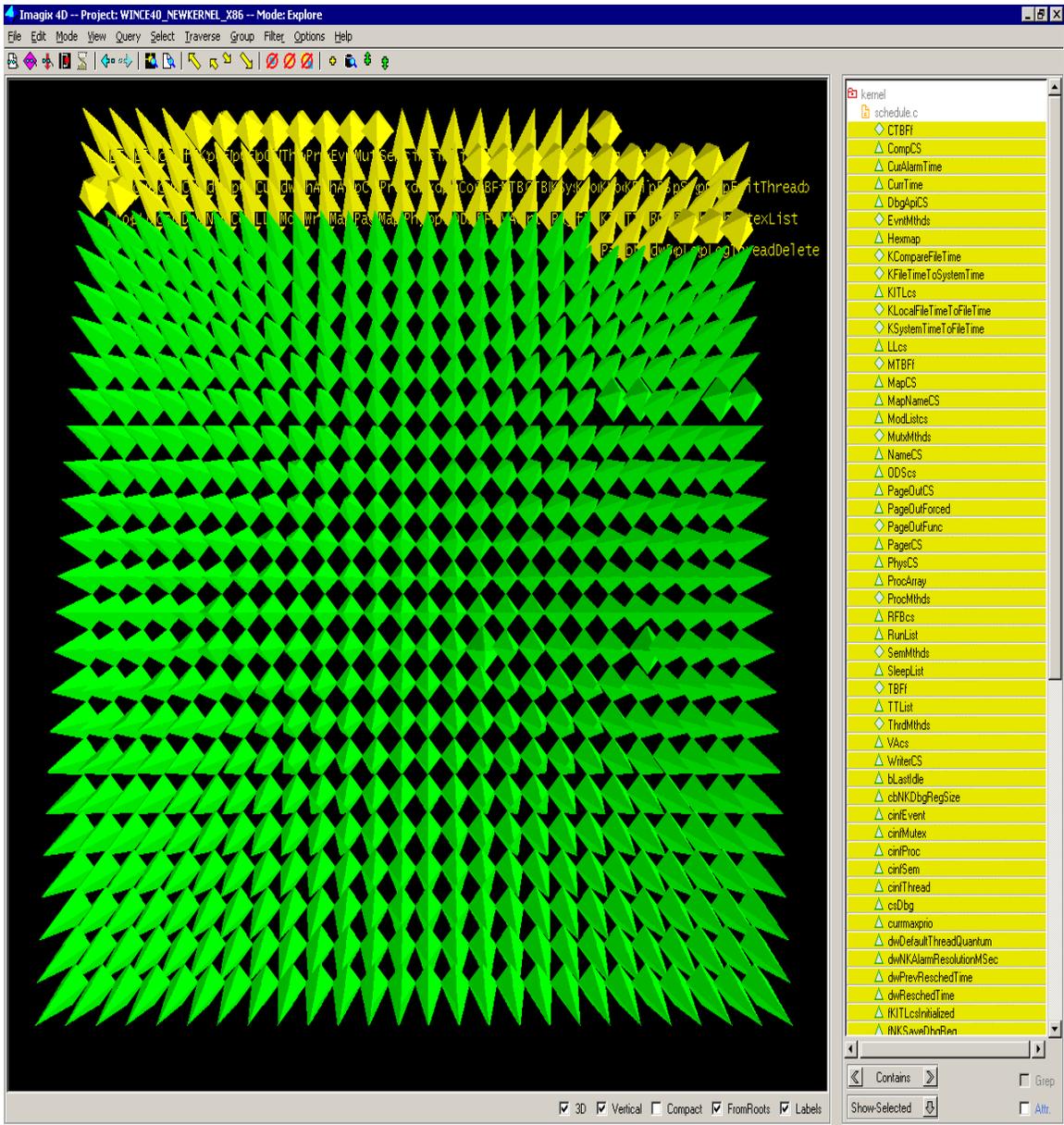


Figure 59. Talisker Kernel Schedule Module – Global Variables.

The analysis section that follows will summarize the information gathered for the schedule module. From these, we will determine which operating system has the scheduling module that best demonstrates information hiding.

F. ANALYSIS OF DATA GATHERED

1. Hierarchical Structuring

From the procedure we established to find the upward dependencies in the six operating system and kernel layering graphs shown in the data gathering section, we are

able to more closely study the grouping of dependencies that go either in the up or down direction in the hierarchical structures. If any dependencies are shown to exist in the upward direction, then we know that circular dependencies may exist between the layers with no partial ordering, thereby breaking down the hierarchical structure. From the screen shots observed in the previous section, we know that none of the structures we analyzed uses complete partial ordering in its layered design. Thus, we will use the numbers gathered from the previous graphs to determine which of the three operating systems and which of the three kernels best exhibits a properly layered system. To determine the number of calls and reads in both the up and down directions in each of the graphs, the arrowheads were counted and summed.

a. Operating System Layering

The operating system summary shown in Table 8 illustrates that the OpenBSD operating system with 11 calls and 3 reads in the upward direction best supports the principle of hierarchical structuring. With 20 calls and 5 reads in the up direction, Linux is the worst example of proper partial ordering. Talisker, in comparison, has 13 calls and 4 reads in the up direction, thereby falling in between the two other operating systems. We will now take a look at the kernel summary.

Operating System	Layers	Dependencies	
		Calls/Reads Down	Calls/Reads Up
Linux	8	6 / 6	20 / 5
OpenBSD	8	5 / 4	11 / 3
Talisker	9	6 / 4	13 / 4

Table 8. Operating System Layers and Dependencies.

b. Kernel Layering

Table 9 shows the kernel summary that was built from the data gathered in the previous section. Using the results found, the OpenBSD kernel with 4 calls and 2 reads in the upward direction again best supports the principle of hierarchical structuring. Talisker, however, with 10 calls and 4 reads in the up direction is the worst among the

three kernels of proper partial ordering. In comparison, Linux with 7 calls and 0 reads in the up direction falls in between the other two kernels.

Kernel	Layers	Dependencies	
		Calls/Reads Down	Calls/Reads Up
Linux	6	4 / 3	7 / 0
OpenBSD	5	3 / 1	4 / 2
Talisker	7	4 / 4	10 / 4

Table 9. Kernel Layers and Dependencies.

2. Modularity

From the data gathered in the previous section, some inferences can be made regarding modularity. For instance, if an operating system uses a large number of modules, it may be inferred that the design properly supports the understanding, analysis, and maintenance of the interfaces. However, if a given module handles a plethora of different functions and manages many databases, its modularity is questionable.

The summaries for the operating system and kernel projects were constructed using the file summary report data, where the total number of modules and functions can be obtained from the source file listings. Using these figures, an average number of functions per module were calculated to help compare the different operating systems and kernels. The summary for the schedule module analysis was built using the function information reports, where the number of “callers” and number of functions can be extracted. These together produced an average number of callers per function to help determine which module has the most well defined interface.

a. Operating System Code

From Table 10, which summarizes the operating system source code, it can be seen that the Talisker operating system exhibits 17 functions per module on average, while Linux has 10 functions per module and OpenBSD has 8 functions per module. Recall that if a given module handles several different functions and manages many databases, its modularity is questionable. This logic may be applied to our

statistical data. Among the three operating systems in our analysis, if a higher average number of functions per module can be found, then the corresponding operating system's modularity is debatable. The contra-positive of this inference would state that an operating system supports modularity better if a lower average number can be found. Since Talisker has the highest average, its modularity is indeed questionable. OpenBSD with the lowest average may be the best example of a modular design. In comparison, Linux exhibits a low average number, but not as well as OpenBSD.

Operating System	Modules	Functions	Avg No Func/Mod
Linux C Code	408	4015	10
OpenBSD C Code	467	3710	8
Talisker C/C++ Code	63	1075	17

Table 10. Summary of Operating System Modules and Functions.

b. Kernel Code

The kernel summary illustrated in Table 11 has the same conclusions as those found in the operating system's logical analysis regarding modularity.

Kernel	Modules	Functions	Avg No Func/Mod
Linux C Code	60	769	13
OpenBSD C Code	82	925	11
Talisker C/C++ Code	40	776	19

Table 11. Summary of Kernel Modules and Functions.

OpenBSD with 11 functions per module on average is the lowest average of the three kernels. Hence, OpenBSD best demonstrates modularity. The Talisker kernel is the furthest from demonstrating proper modularity with an average of 19

functions per module. In comparison, Linux falls in between these two but closer to OpenBSD with an average of 13 functions per module.

c. Schedule Module Code

In the analysis of the schedule module, the “callers” metric was used from the function information reports to reveal how well the module interfaces are defined and used. If the average number of callers per function is high, then other modules call upon the encapsulated functions of a given module through a well-defined interface. If this average is low, then there exists a wide number of functions that are called rarely, which may demonstrate an interface that is broad and not well-defined.

Schedule Module	Functions	Functions Called	Callers	Avg No Calls/Func
Linux sched.c	17	11	30	3
OpenBSD kern_synch.c	17	16	126	8
Talisker schedule.c	143	141	262	2

Table 12. Summary of Schedule Module Functions and Callers.

From the summary in Table 12 of the three schedule modules, the OpenBSD module, with an average of 8 calls per function (126 callers divided by 16 called functions out of 17), best meets this quality of a well-defined interface. Linux is not too far behind with an average of 3 calls per function (30 callers divided by 11 called functions out of 17). In comparison to these three, Talisker’s schedule module, with 2 calls per function (262 callers divided by 141 called functions out of 143), has the lowest. This result infers that Talisker’s schedule interface is too broad and its modularity is questionable since it includes several different functions and manages many databases that are rarely called. This makes writing the schedule module with little to no knowledge of the code in other modules difficult, and it does not allow the module to be replaced or reassembled without having any effect on the system as a whole.

3. Information Hiding

From our previous basis discussion, information hiding helps shift attention away from the code used to implement the module and concentrate more on the signature or interface of the module. Therefore, the member variables used by each module should be private, and a simple, high-level, well-defined interface should be the only mechanism that manipulates their values. Global variables do not exhibit any of these properties since two or more modules share them in the system, degrading the modularity of the design. The summaries that follow will help determine which of the operating systems, kernels, and schedule modules has the least percentage of global variables in each of their included sets of variables. This will aid in determining the better design regarding information hiding and modularity.

Variable data from the file summary reports along with the global variable search procedure explained in the last section were used to build the operating system and kernel summaries. For the number of variables in the summary tables, the *source* amount was extracted from the “source file” portion of the file summary report, while the *global headers* value was taken from the “header file” category. The *globals in the source* value was found using the search method using Imagix 4D’s graph and list windows. The total number of variables, used in finding the percentage of global variables overall, was taken from the “total file” portion of the summary report. The schedule module summaries only used the data directly found using the global search method.

a. Operating System Global Variables

The summary in Table 13 illustrates the results of the analysis performed on the three operating systems.

Operating System	Number of Variables		Globals in Source	Total Globals	% Total Variables that are Global
	Source	Global Hdrs			
Linux	21580	6980	1138	8118	8118/28560 (28%)
OpenBSD	21165	5129	1773	6902	6902/26294 (26%)
Talisker	5497	5907	450	6357	6357/11404 (56%)

Table 13. Operating System Global Variable Summary.

In Linux, 1138 global variables out of 21580 were found in the source. This number summed with the 6980 global variables included in the header files yielded a total global variable percentage of 28 (8118 global variables divided by 28560 total variables) with respect to all of the variables listed in the Imagix 4D database for the Linux operating system. With 1773 global variables out of 21165 found in the source summed with the 5129 global header variables, OpenBSD has the lowest percentage of 26 (6902 global variables divided by 26294 total variables) with respect to the variables loaded in the database. With respect to the total variable count in the Imagix 4D database, the Talisker operating system had a percentage of 56 (6357 global variables divided by 11404 total variables), which was found using summation of the 450 global variables out of 5497 found in the source plus the 5907 global header variables. From these results, the principle of information hiding is best demonstrated by the OpenBSD operating system. The kernel analysis will be presented next for comparison.

b. Kernel Global Variables

Of the global variables found in the source using the Imagix 4D search method defined earlier in the data gathering section, 214 global variables out of 2880 were found in the Linux kernel, 296 global variables out of 4923 were found in the OpenBSD kernel, and 369 global variables out of 4010 were found to exist in the Talisker kernel.

Kernel	Number of Variables		Globals in Source	Total Globals	% Total Variables that are Global
	Source	Global Hdrs			
Linux	2880	2112	214	2326	2326/4992 (47%)
OpenBSD	4923	2842	296	3138	3138/7765 (40%)
Talisker	4010	4765	369	5134	5134/8775 (59%)

Table 14. Kernel Global Variable Summary.

From the global variable summary presented in Table 14, it can be observed that the OpenBSD kernel has the fewest number of global variables with respect to the total number of variables loaded into the Imagix 4D database with a percentage of 40 (3138 global variables divided by 7765 total variables), while the Talisker kernel has

the highest number with a percentage of 59 (5134 global variables divided by 8775 total variables). Linux falls in between of these two with a percentage of 47 (2326 global variables divided by 4992 total variables). The ranking of the three kernels match that of the operating system analysis, with OpenBSD first, Linux second, and Talisker third.

c. Schedule Module Global Variables

In the schedule module analysis, the search method was the only approach available to count the total number of variables and global variables in the module source code. As illustrated in Table 15, which shows the schedule module summary, the “sched.c” module in Linux has 7 global variables out of 46 (15 percent). OpenBSD’s “kern_synch.c” file has the least amount with a percentage of seven (5 global variables divided by 67 total). The “schedule.c” module in Talisker falls in between the previous two with 79 global variables out of 776 (10 percent). Like the operating system and kernel analysis, OpenBSD’s schedule module best satisfies the information hiding principle, but Talisker is second with Linux being third.

Schedule Module	Number of Source Variables	Globals in Source	% Total Variables that are Global
Linux sched.c	46	7	15%
OpenBSD kern_synch.c	67	5	7%
Talisker schedule.c	776	79	10%

Table 15. Schedule Module Global Variable Summary.

4. McCabe Cyclomatic Complexity Distribution

From McCabe’s threshold points for the cyclomatic complexity as it pertains to software design, a value of less than 10 is ideal for good design, but a value greater than 10 requires that the designers know what behavior is desired, and that they know how to achieve this mapping of specifications to the code implementation properly.

In the operating system, kernel, and schedule module summaries that follow, function information reports were used to extract the cyclomatic complexity numbers for the functions included in each module. Of the total number of functions loaded into the

Imagix 4D database, most of these had complexity data calculated and made available by the automated tool. The others not considered did not fit McCabe’s profile of a function with entry and exit points and decision logic (i.e., macros or other types). Of the data gathered, the cyclomatic complexity distributions for the operating systems, kernels, and schedule modules can be compared against McCabe’s threshold points as well as against each other to see which is the most and least complex. Complexity values greater than 10 are discussed in the analysis, while the greater than 15 values are merely illustrated for continuity.

a. Operating System Complexity

From the summary in Table 16, which illustrates the complexity distributions for the three different operating systems, Talisker is the least complex system with only 13 percent (141 functions out of 1067) of its functions having complexity values greater than 10. The most complex is OpenBSD, with 720 of 3596 (20 percent) of its functions having complexity amounts exceeding 10. With 597 of 3973 (15 percent) of its functions having complexity values greater than 10, Linux is in between the previous two. The kernel summary will be reviewed next for comparison.

Operating System	Functions		Complexity	
	Total Number	Complexity Data Avail	Value > 10	Value > 15
Linux	4015	3973	597 (15%)	336 (8%)
OpenBSD	3710	3596	720 (20%)	415 (12%)
Talisker	1075	1067	141 (13%)	88 (8%)

Table 16. Operating System Complexity Distribution.

b. Kernel Complexity

The complexity distribution summary for the three different kernels is shown in Table 17. Like the operating system analysis, the OpenBSD kernel is the most complex of the three with 21 percent (195 functions out of 922) of its functions having complexity values greater than 10. In comparison to the previous analysis, the least complex is the Linux kernel with 82 of 753 (11 percent) of its functions having

complexity amounts exceeding 10. With 118 of 776 (15 percent) of its functions having complexity values greater than 10, the Talisker kernel falls in between the other two kernels.

Kernel	Functions		Complexity	
	Total Number	Complexity Data Avail	Value > 10	Value > 15
Linux	769	753	82 (11%)	37 (5%)
OpenBSD	925	922	195 (21%)	112 (12%)
Talisker	776	776	118 (15%)	75 (10%)

Table 17. Linux Kernel Complexity Distribution.

c. Schedule Module Complexity

As the schedule module summary shows in Table 18, this portion of the complexity analysis has Linux as the least complex followed by OpenBSD and then Talisker.

Schedule Module	Functions		Complexity	
	Total Number	Complexity Data Avail	Value > 10	Value > 15
Linux sched.c	17	14	1 (7%)	0 (0%)
OpenBSD kern-synch.c	17	17	3 (18%)	1 (6%)
Talisker schedule.c	143	143	33 (23%)	19 (13%)

Table 18. Schedule Module Dependencies and Complexity Summary.

The Linux schedule module with only one of its 17 functions having a complexity greater than 10 is the least complex, while OpenBSD with 3 of 17 functions has a complexity greater than 10. In comparison, Talisker has 33 of 143 module

functions with values greater than 10, leading to the highest complexity of all the schedule modules analyzed.

5. Number-of-Lines-of-Code

The file summary reports from the operating system and kernel projects provide data about the source files regarding the number of functions and the total number-of-lines-of-code. These measures were used to build the operating system and kernel summary reports that follow. For the schedule module analysis, the summary table constructed used data extracted from the function information reports since specific module details are not included in the file summaries. Using this data gathered in the previous section, an average number of lines per function were calculated for each case in the analysis. These results will help determine which operating system, kernel, and schedule module best matches the archaic thumb rule that a function should not exceed more than 60 lines of code. Although McCabe metric is a better measure of complexity, this analysis is included to compare against the cyclomatic complexity results.

a. *Operating System Code*

The McCabe complexity analysis for the three operating systems placed Talisker as the least complex followed by Linux, with OpenBSD next. In the complexity summary illustrated in Table 19, these results agree with the McCabe analysis. With an average of 52 lines of code per function (56K lines of code divided by 1075 functions), Talisker is the least complex. Linux follows with 61 lines per function (245K lines of code divided by 4015 functions). And OpenBSD is the most complex with 66 lines of code per function on average (245K lines of code divided by 3710 functions).

Operating System	Functions	Lines	Avg No. Lines/Func
Linux C Code	4015	245K	61
OpenBSD C Code	3710	245K	66
Talisker C/C++ Code	1075	56K	52

Table 19. Operating System - Lines of Code Complexity Summary.

b. Kernel Code

From the previous kernel analysis using McCabe’s complexity metric, Linux, Talisker and OpenBSD was the ranking found from least to most complex. Using the average number of lines per function, the results again agree with the earlier analysis as shown in Table 20. Linux, with 39K lines of code divided by 769 functions, has an average of 51 lines per function, making it the least complex kernel. The Talisker kernel comes in second with 57 lines per function (44K lines of code divided by 776 functions). And the most complex, with an average of 61 lines of code per function (56K lines divided by 925 functions), is the OpenBSD kernel.

Kernel	Functions	Lines	Avg No. Lines/Func
Linux C Code	769	39K	51
OpenBSD C Code	925	56K	61
Talisker C/C++ Code	776	44K	57

Table 20. Kernel - Lines of Code Complexity Summary.

c. Schedule Module Code

As the schedule module summary shows in Table 21, the “sched.c” module found in Linux is the least complex with 23 lines of code per function on average (394 lines divided by 17 functions).

Schedule Module	Functions	Lines	Avg No. Lines/Func
Linux sched.c	17	394	23
OpenBSD kern_synch.c	17	546	32
Talisker schedule.c	143	5413	38

Table 21. Schedule Module - Lines of Code Complexity Summary.

OpenBSD’s “kern_synch.c” source file is the second in line with 32 lines per function (546 lines divided by 17 functions). In comparison, the “schedule.c” module

found in Talisker has an average of 38 lines of code per function (5413 lines divided by 143 functions), making it the most complex. These results again match those found using the McCabe complexity metrics.

G. COMPLEXITY IN OPERATING SYSTEMS

Alan Shaw [SHA74] once stated that computer operating systems are among the most complex “systems” ever devised by humans, and it is only recently that we have been able to understand and coherently organize this complexity. From the complexity analysis that was conducted using the McCabe approach, the values calculated by Imagix 4D greatly surpassed the recommended threshold points previously presented. Some of the reasons for these large values may be due to the complex nature of operating systems as commented by Shaw. Memory addressing, process scheduling, interrupts and exceptions, system calls, I/O device management, virtual file system management, and program execution in a multithreaded environment, are only a few of the many capabilities required of modern operating systems today. The values found in our comparative analysis may not necessarily be bad, but rather characteristic of complexity at another level. Therefore, this higher complexity should not be a reason to deter improvements in the design and implementation of operating systems.

H. CONCLUSION

From our comparative analysis, several different aspects of the three operating systems and their respective kernels and schedule modules were studied, keeping the principles of hierarchical structuring, modular design, and information hiding serving as the basis of the analysis. The McCabe Cyclomatic Complexity and the number-of-lines-of-code metrics were also included in this basis. As a result, Table 22 shows a summary of the design and complexity issues considered. In this presentation, a number one signifies that a certain system, kernel, or module exhibited the best qualities of a certain design principle. For the two complexity measures illustrated, a number one signifies the least complex. As illustrated, OpenBSD is the best overall when considering hierarchical structuring, modularity, and information hiding. The Linux kernel and scheduler, on the other hand, are least complex when considering the McCabe complexity and the number-of-lines-of-code, whereas the Talisker scores best of the operating systems.

	Hierarchical Structuring	Modularity	Information Hiding	McCabe Complexity	Number-of-Lines-of-Code
Linux OS	3	2	2	2	2
OpenBSD OS	1	1	1	3	3
Talisker OS	2	3	3	1	1
Linux Kernel	2	2	2	1	1
OpenBSD Kernel	1	1	1	3	3
Talisker Kernel	3	3	3	2	2
Linux Sched	X	2	3	1	1
OpenBSD Synch	X	1	1	2	2
Talisker Schedule	X	3	2	3	3

Table 22. Summary of Comparative Analysis.

VII. CONCLUSION

A. OPERATING SYSTEM SECURITY REQUIREMENTS

It is important to note that system security requirements exist as a management tool. In essence, this tool assists in the identification and prioritization of project implementation decisions, which impact the evolution of a reasonably secure operating system. The effectiveness of this tool may be limited by the importance placed upon security in relation to the operating system and the data, which is generated and residing on the system.

When weighing performance requirements against system security requirements, a balance must be defined and preserved. It is of little value to a user of a high performance operating system if resource availability is sporadic due to control or manipulation of the operating system by an unauthorized source. It is also of little value to a system user if resource availability is significantly degraded due to over tasking in support of system security requirements. Nevertheless, security requirements within operating systems should not be left for discovery at a later date or briefly discussed and planned during a project wrap-up. They need to be thoroughly defined in parallel with other requirement analyses in order to reduce cost and produce a system that exhibits security behavior that is coherent with respect to its requirements and that possesses a proper balance between other functionality and security.

B. SECURE OPERATING SYSTEM DEVELOPMENT

As stated by Anderson [AND96], we should never forget that the great majority of actual security failures in operating systems result from simple blunders in design, construction, and operation.

From the secure operating system overview, we noted that the environment to be protected must be well understood, an assessment of the threats to security must be reviewed, and the operating system must then be designed to provide the desired protection. For higher levels of security, the protection scheme may include a security kernel that is responsible for enforcing the security policy. To encompass all of the parts of a trusted operating system on which we depend for the correct enforcement of a

security policy, the notion of a trusted computing base and its security perimeter can also be included in the design of a general-purpose or embedded operating system.

The security policy and model, trusted system design elements, and the security kernel concept are some of the key concepts to consider when building a secure operating system. Hierarchical structuring, modularity, and information hiding, which were presented in our study as secure software design principles, are also important elements of design and implementation. To ensure all of these principles are incorporated into the development of a secure operating system and maintained throughout its lifecycle, evaluation methods such as the Common Criteria (CC) and the Trusted Computer System Evaluation Criteria (TCSEC) can be utilized to establish a level of assurance. Thus, the operating system will be implemented in such a way that the user has confidence that it will enforce the security policy, and it will meet the expectations of the user with a certain degree of trust.

Since operating systems are complex by their nature, the designers of a secure operating system need to spend time thinking about the structure before writing the code. When implementation begins, the complexity of the source code should be measured to limit the size and complexity of the operating system, making it possible to conduct meaningful testing or certification as determined by evaluation methods such as TCSEC. The McCabe Cyclomatic Complexity metric, which measures the amount of decision logic in a software module, is one way to correlate a source module's complexity to its error frequency, thereby providing a strong indicator of its testability, as well as its understandability and receptiveness to modification. Thus, reducing complexity and size must be the goal in every step of the design, including system specification, design, and detailed programming. With these goals of constructing source code that is straightforward and understandable, an automated tool such as Imagix 4D can be a valuable aid to help each programmer of the project team have a thorough comprehension of the software, and help produce source code that is less complex, less error-prone, and easier to test.

C. COMPARATIVE ANALYSIS CONCLUSIONS AND COMMENTS

Hierarchical structuring, modularity, information hiding, McCabe complexity, and number-of-lines-of code composed the basis of the comparative analysis performed on the Linux, OpenBSD, and Talisker operating systems, kernels, and scheduling modules.

In the hierarchical structuring analysis, the OpenBSD operating system had the best design with the fewest number of circular dependencies between each of its software layers. Linux and Talisker, in this category, fell short with a few more upward dependencies. In the kernel analysis, however, Linux demonstrated more loops in its design as compared to Talisker. As Parnas [PAR96] stated, performance goals and hardware limitations often interfere with structure. As the results from our study illustrate, operating systems like Talisker and Linux may place functionality and performance as the primary design goals, rather than giving proper attention to the system architecture. If an operating system is designed with a layered hierarchy that exhibits partial ordering, then the upper layers can be removed without affecting the rest of the system and circular error states between the layers can be avoided.

Regarding modularity, OpenBSD best demonstrated the properties expected of a good design. In the operating system, kernel, and schedule module analysis, Linux followed behind OpenBSD. In comparison, Talisker with the fewest number of source files in its design demonstrated questionable modularity. Good modular programming helps guarantee that errors in one module will not affect another. With this kind of control in the design, it is much easier to gain confidence in the reliability of a large system, since the modular separation helps prevent the spread of trouble created by malicious activity or flaws in the operating system.

The ranking pattern of OpenBSD first, followed by Linux, and then Talisker was repeated in the information hiding analysis. Talisker did have a smaller percentage of global variables in its schedule module as compared to the Linux implementation, but the commercial operating system did exhibit the largest percentage in the operating system and kernel portions of the analysis. Regarding good modular programming, information hiding is an important element that ensures that the interface is well defined. The only

way a module's internal databases or variables should be manipulated is through a simple, high-level interface, thereby hiding the details of the implementation.

In the analysis of complexity using both the McCabe and lines of code metrics, OpenBSD does not fare well in comparison to the other two operating systems. Instead, Linux exhibits the least amount of complexity when reviewing the kernel and schedule module analysis results. In the operating system portion, however, Talisker was the least complex. The OpenBSD project team claims that no system installed with default settings has ever been remotely compromised in the past four years [GEO02]. But how can this be true if it is more complex in design? The entire source code tree for OpenBSD is audited for the most frequent of security problems, including buffer overflows. As commented by Vaughan-Nichols [VAU01], OpenBSD has had a team of auditors since 1996 working on finding potential problems and fixing them before they can develop in to security holes, making them more proactive rather than reactive to the computer security problem. From their extensive design, implementation, and code audit efforts, the source code is clean, consistent, and correct, providing a level of confidence that surpasses Linux and other commercial operating systems. This rigorous design process agrees with McCabe's argument that complexity measures greater than 10 should be reserved for projects that have several operational advantages over typical projects, including experienced staff, formal design, a modern programming language, structured programming, code walkthroughs, and a comprehensive test plan.

It is also worth mentioning that the first BSD operating system did not consider security as its first priority. As the Multics case study demonstrated, any attempt to retrofit security is a difficult task. Therefore, some of the complexity exhibited by OpenBSD may be remnants of the original BSD design.

Since it places security as its first priority, OpenBSD's design may be more complex due to the code it includes for security reasons. With the "daily security audit" and other features, OpenBSD includes many more auditing cryptographic functions and that are built into the design and turned on by default.

From Seifried's [SEI01] online discussions, he gives reasons why Linux will never be as secure as OpenBSD. He claims that in the Linux world no form of extensive

code auditing exists and likely never will. This is due to the vendors of the distributions having a small security team of less than a dozen people trying to review several hundred megabytes of source code. Hence, it is common for the original software to be insecure causing vendors to rely on the maintenance of their own patch sets rather than being proactive to security problems.

Security requirements may cause the software to exhibit more complexity and be somewhat less efficient in performance. Thus, the customer of an operating system needs to weigh the security risks against the performance costs, and try to find a medium that provides the proper amount of protection in the particular, security threat environment. Regardless of the complexity, the system development process should be to design the architecture before implementing it, document the design, review and analyze the documented design, and review the implementation for consistency with the design. Using these basic rules, the trusted system design elements, and the secure software design principles, a more secure operating system may be achieved.

D. FUTURE WORK

Future work may include normalizing and weighting of the data gathered in this analysis to provide meaningful overall metrics for operating system complexity, as well as finding a more appropriate set of McCabe threshold values that apply to operating system design. An inter-function complexity measure may also be useful to determine the gratuitous use of macros and sub-functions. Additionally, a more granular approach to the cyclomatic complexity would be useful to distinguish inherent functional complexity from pathological coding complexity. These improvements to the McCabe metric could help provide a more detailed result consistent with the expectations of operating system complexity.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [AND72] Anderson, J.P., "Computer Security Technology Planning Study." *U.S. Air Force Electronic Systems Division Technical Report 73-51*, October 1972.
- [AND96] Anderson, R., and Kuhn, "Tamper Resistance – A Cautionary Note." *The Second USENIX Workshop on Electronic Commerce Proceedings*, Oakland, California, November 18-21, 1996, pp. 1-11.
- [BBC01] "Microsoft Downplays Hack Attack." *BBC News*, October 30, 2000.
- [BEL73] Bell, D., LaPadula, L., "Secure Compute Systems: Mathematical Foundations and Model." *MITRE Report*, MTR 2547, Vol.2, November 1973.
- [BEN84] Benzel, T., "Analysis of a Kernel Verification." *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1984, pp. 125-131.
- [BOV01] Bovet, D. P., and Cesati, M., *Understanding the Linux Kernel*, O'Reilly and Associates, 2001.
- [BRI93] Brinkley, D., and Schell, R., "What Is There to Worry About? An Introduction to the Computer Security Problem." *Information Security: An Integrated Collection of Essays*, IEEE Computer Society Press, Essay 1, 1993.
- [CC99] *Common Criteria for Information Technology Security Evaluations*, version 2.1, August 1999.
- [COR91] Corbató, F., "On Building Systems That Will Fail." (1991 Turing Award Lecture), *Communications of the ACM*, Vol.34, No.9, September 1991, pp. 72-81.
- [DIJ68] Dijkstra, E.W., "The Structure of the "THE"-Multiprogramming System." *ACM Symposium on Operating Systems Principles*, Vol. 11, No. 5, May 1968, pp. 341-345.
- [DER97] Derrick, J., "Cyclomatic Complexity Metrics." *Pascal-central.com*, 1997.
- [DOD85] *Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, U.S. Department of Defense, December 1985.
- [FER92] Final Evaluation Report 92/003, "HFS Incorporated: XTS-200." National Computer Security Center, May 27, 1992.
- [FER95] Final Evaluation Report 94/34, "Gemini Computers, Incorporated: Gemini Trusted Network Processor, Version 1.01." National Computer Security Center, June 28, 1995.
- [GAU70] Gauthier, R., and Pont, S., *Designing Systems Programs*, Prentice-Hall, Englewood Cliffs, N.J., 1970.

- [GRA72] Graham, R., and Denning, P., "Protection – Principles and Practices." *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol.40, 1972, pp. 417-429.
- [GEO02] "Linux, OpenBSD, Windows NT/2000 Server Comparison," *GeodSoft.com*, Web Site Design and Development, 2002.
- [HAL01] Hall, M., and Maillet, S., "A Taste of Talisker." Microsoft Corporation, November 6, 2001.
- [HAR76] Harrison, M., Ruzzo, W., and Ullman, J., "Protection in Operating Systems." *Communications of the ACM*, Vol.19, No.8, August 1976, pp. 461-471.
- [KAR74] Karger, P., Zurko, M., Bonin D., Mason, A., and Kahn, C., "A VMM Security Kernel for the VAX Architecture." *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, IEEE Computer Society, Oakland, California, May 7-9, 1990, pp. 2-19.
- [LAM71] Lampson, B., "Protection." *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971, pp. 437-443.
- [LAM76] Lampson, B., and Sturgis, H., "Reflections on Operating System Design." *Communications of the ACM*, Vol.19, No.5, May 1976, pp. 251-266.
- [LEH99] Lehey, G., "Explaining BSD." *Enderunix.org*, April 1999.
- [LEM01] Lemos, R., "Microsoft Warns of Hijacked Certificates." *CNET News.com*, March 22, 2001.
- [MCC96] McCabe, T. J., "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric." *NIST Special Publication 500-235*, Computer Systems Laboratory, MD, September 1996.
- [MCW01] McWilliams, B., "Suspect Claims Al Qaeda Hacked Microsoft." *Newsbytes*, December 17, 2001.
- [MYE80] Myers, P.A., *Subversion: The Neglected Aspect of Computer Security*, Master of Science Thesis, Naval Postgraduate School, Monterey, California, June 1980.
- [NIS91] NIST (National Institute of Standards and Technology), "Glossary of Computer Security Terminology." *NIST Technical Report*, NISTIR 4659, September 1991.
- [PAR72] Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules." *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.
- [PAR96] Parnas, D. L., "Why Software Jewels are Rare." *IEEE Magazine*, Vol.29, No.2, February 1996, pp. 57-60.
- [PET00] Peters, J., and Pedrycz, W., *Software Engineering: An Engineering Approach*, John Wiley and Sons, 2000.

- [PFL97] Pleeger, C. P., *Security in Computing*, Second Edition, Prentice Hall, 1997.
- [SAL75] Saltzer, J.H., and Schroeder, M.D., "The Protection of Information in Computer Systems." *Proceedings of the IEEE*, Vol.63, No. 9, September 1975, pp. 1278-1308.
- [SCH75] Schroeder, M.D., "Engineering a Security Kernel for Multics." *Proceedings of The Fifth Symposium on Operating Systems Principles*, November 1975, pp. 25-31.
- [SCH77] Schroeder, M.D., Clark, D.D., and Saltzer, J.H., "The Multics Kernel Design Project." *Proceedings of Sixth ACM Symposium on Operating Systems Principles*, November 1977, pp. 43-56.
- [SEI90] Seiden, K., and Melanson, J., "The Auditing Facility for a VMM Security Kernel." *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1990, pp. 262-277.
- [SEI01] Siefried, K., "Why Linux Will Never Be As Secure As OpenBSD," *Siefried.org*, Information Security, August 11, 2001.
- [SHA74] Shaw, A. C., *The Logical Design of Operating Systems*, Prentice Hall, 1974.
- [STA98] Stallings, W., *Operating Systems Internals and Design Principles*, Third Edition, Prentice-Hall, 1998.
- [THO84] Thompson, K., "Reflections on Trusting Trust." (1983 Turing Award Lecture), *Communications of the ACM*, Vol.27, No.8, August 1984, pp. 761-763.
- [VAU01] Vaughan-Nichols, S., "OpenBSD: The Most Secure OS Around." *ZDNet.com*, ZDNet Tech Update, November 6, 2001.
- [WIR95] Wirth, N., "A Plea for Lean Software." *IEEE Magazine*, Vol.28, No.2, February 1995, pp. 64-68.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Commander, Naval Security Group Command
Naval Security Group Headquarters
Fort Meade, Maryland
San Diego, California
4. Commander Officer
Space and Naval Warfare Systems Center – San Diego
San Diego, California
5. Ms. Deborah M. Cooper
Deborah M. Cooper Company
Arlington, Virginia
6. Ms. Louise Davidson
N643
Arlington, Virginia
7. Ms. Elaine S. Cassara
Branch Head, Information Assurance
United States Marine Corps
Washington, DC
8. Mr. William Dawson
Community CIO Office
Washington, DC
9. Ms. Deborah Phillips
Community CIO Office
Washington, DC
10. Capt. Robert A. Zellman
CNO N6
Arlington, Virginia

11. Dr. Ralph Wachter
Office of Naval Research
Arlington, Virginia
12. Major Dan Morris
HQMC, C4IA Branch
Washington, DC
13. Mr. Richard Hale
Defense Information Systems Agency
Falls Church, Virginia
14. James P. Anderson
James P. Anderson Co.
Ambler, Pennsylvania
15. Mr. David Ladd
Microsoft Corporation
Redmond, Washington
16. Mr. Steve Lipner
Microsoft Corporation
Redmond, Washington
17. Mr. Rui Maximo
Microsoft Corporation
Redmond, Washington
18. Mr. John SanGiovanni
Microsoft Corporation
Redmond, Washington
19. Mr. Miguel Claudio
Microsoft Corporation
Redmond, Washington
20. Mr. Mike Thomson
Microsoft Corporation
Redmond, Washington
21. Mr. Sergio Cherskov
Microsoft Corporation
Redmond, Washington

22. Mr. John H. Townsend
Information Assurance and Engineering Division (D87)
SPAWAR Systems Center – San Diego
San Diego, California
23. Mr. John E. Sherwood
Code 723JES
SPAWAR Systems Center - Charleston
North Charleston, South Carolina
24. Mr. John Blattner
Imagix Corporation
San Luis Obispo, California
25. Dr. Annie I. Antón
North Carolina State University
Department of Computer Science
Raleigh, North Carolina
26. Dr. Cynthia E. Irvine
Computer Science Department, Code CS/IC
Naval Postgraduate School
Monterey, California
27. Mr. Timothy Levin
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, California