Combining Logical and Physical Access Control for Smart Environments

Kristine Frank
Ida C. Willemoes-Wissing

Lyngby, August 2004
M.Sc. Thesis

IMM

Computer Science and Technology
Informatics and Mathematical Modelling
Technical University of Denmark

Trykt af IMM, DTU

Abstract

Traditional access control models only protect logical entities within the computer (such as files and memory) and not information displayed on a computer monitor. Furthermore, it is processes that are granted or denied access to resources, not the persons who are physically present in front of the computer. Logical access control models are inadequate if the environment is physically unprotected and an intruder uses coercion to obtain access to otherwise classified information. The coercion could include weapons, leaving the user with no option but to grant access to the computer.

The theoretical contribution of this thesis is an access control model that not only takes files and process into consideration when making access control decisions, but also the persons physically present in the environment and the information displayed on a computer monitor. The model is a multilevel security model where files, processes, windows and unauthorized persons are associated with security levels. These levels are used as the basis for mandatory access control decisions. If a person in the environment is denied viewing access to a window, the window will disappear from the computer monitor so that it no longer is human-readable.

The technical contributions fall in three modules. Firstly, a stackable file system has been extended so that it can enforce mandatory access control. Secondly, a simple movement sensor based on two web-cameras can detect whether unauthorized persons enter or leave the environment. Finally, a module combines the logical and physical access control and ensures that windows on the computer monitor are made invisible when the data received from the sensor indicates that unauthorized persons are present. The system has been developed so that it can be integrated with a Unix operating system.

The security policy enforced by the system is set by parameters during startup. These parameters can, for instance, specify that the system should conform to the Bell-LaPadula model or the Biba model and thus address confidentiality or integrity, respectively.

Keywords: access control, multilevel security models, sensors, motion detection, operating systems, and stackable file systems.

Resumé

Adgangskontrolmodeller beskytter normalt kun logiske objekter i en computer (så som filer og hukommelse) og ikke information, der bliver vist på en computerskærm. Desuden er det processer, som kan få tildelt adgang til ressourcer, og ikke personerne der er fysisk tilstede foran computeren. Logiske adgangskontrolmodeller er utilstrækkelige, hvis miljøet er fysisk ubeskyttet og en uautoriseret person bruger tvang til at opnå adgang til klassificeret information. Tvangen kan inkludere våben, hvilket resulterer i at brugeren ikke har andre muligheder end at give adgang til computeren.

Det teoretiske bidrag i denne afhandling er en adgangskontrolmodel, som ikke kun betragter filer og processor, når beslutninger om adgangskontrol skal tages, men også de personer der er fysisk til stede i miljøet og den information der bliver vist på computerskærmen. Modellen er en fler-niveaus sikkerhedsmodel, hvor filer, processer, vinduer og uautoriserede personer er associeret med sikkerhedsniveauer. Disse niveauer udgør fundamentet for beslutninger om obligatorisk adgangskontrol. Hvis en person i miljøet ikke må se et vindue vil det forsvinde fra computerskærmen, således at det ikke længere er muligt at se det.

De tekniske bidrag kan inddeles i tre moduler: Det første modul er et stakbart filsystem, der er blevet udvidet således, at det kan håndhæve obligatorisk adgangskontrol. Det andet modul er en simpel bevægelsessensor, der er baseret på to web-kameraer, og som kan opdage om uautoriserede personer indtræder i eller forlader miljøet. Det tredje modul kombinerer den logiske og fysiske adgangskontrol samt sikrer, at vinduer på computerskærmen bliver usynlige, når data modtaget fra sensorerne indikerer, at uautoriserede personer er til stede. Systemet er udviklet således, at det kan integreres med et Unix operativsystem.

Den sikkerhed, der påtvinges af systemet, er sat vha. parametre som en del af systemopstarten. Disse parametre kan for eksempel specificere, at systemet skal rette sig efter Bell-LaPadula modellen eller Biba modellen og derved hhv. adressere fortrolighed og integritet.

Preface

This M.Sc. thesis is the result of our work carried out in the period from January 2004 to August 2004. The thesis was developed at the Computer Science and Engineering (CSE) division of the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU). The work was supervised by associate professor Christian Damsgaard Jensen.

We would like to thank Christian Damsgaard Jensen for his feedback and helpful suggestions. Furthermore, we would like to thank our family for their support, Susan Rabbe for proofreading parts of the thesis, and the PhD students in the Safe and Secure IT-Systems group at IMM for good company while writing this thesis.

Lyngby, August 2004.

Kristine Frank

Ida C. Willemoes-Wissing

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Sensor Enhanced Access Control Model 1

1.3 The Developed Prototype 3

1.4 Contributions 3

1.5 Thesis Organization 4

2 Security and Access Control **7**

2.1 Computer Security 7

2.2 Access Control 8

2.2.1 Physical Access Control 9

2.2.2 Logical Access Control 10

2.3 Summary 17

3 Sensors and Motion Detection **19**

3.1 Sensors 19

3.2 Motion Detection 20

3.2.1 Infrared Sensors 20

3.2.2 Radio Wave Frequency Sensors 20

3.2.3 Digital Images 21

3.3 Summary 22

4 Unix Background Information **25**

4.1 The X Window System 25

4.2 File Systems 27

4.2.1 The Virtual File System 27

4.2.2 Stackable File Systems 28

5 Sensor Enhanced Access Control Model 31
5.1 Logical Access Control 31
5.1.1 Files 32
5.1.2 Users 32
5.2 Physical Access Control 32
5.2.1 Windows 32
5.2.2 Persons 33
5.3 Combining Logical and Physical Access Control 34
5.3.1 Reference Monitors 35
5.3.2 Security Policy 38

6 Design 41
6.1 Software Architecture Overview 41
6.1.1 The Subsystems 41
6.1.2 Processes and Message Passing 42
6.2 The Security Policy Parameters 44
6.3 The Stackable File System 45
6.4 File Level Management 46
6.5 User Level Management 46
6.6 Window Management 47
6.7 Motion Detection using Sensors 48
6.7.1 Choice of Sensors 48
6.7.2 Motion Detection Programs 49
6.7.3 Design of Physical Premises 50
6.8 CSP Specification 50
6.8.1 Processes, Channels, and Users 51
6.8.2 Data Types 53
6.8.3 Functions 53
6.8.4 The Communication Protocol 55

7 Implementation 65
7.1 The Stackable File System 65
7.1.1 Storage of Levels 66
7.1.2 FiST Input File 66
7.2 Window Management 70
7.2.1 The Visibility Manager 70

7.2.2 Intercepting Window Creation and Destruction 71
7.2.3 Intercepting File Open 72
7.2.4 Handling of Backup Files 72
7.2.5 The Sensor Server 74
7.2.6 Printing Subject and Window Status Information 74
7.3 Sensors 75
7.3.1 Web-cameras 75
7.3.2 Motion, a Motion Detection Program 75
7.3.3 Motion Detection Programs 78
7.3.4 Known Limitations in the Camera System 83
7.4 The Security Manager GUI 84
7.4.1 The GUI Functionality 84
7.4.2 Interfacing between Java and C Programs 86
7.4.3 The GUI Classes 86

8 Evaluation **87**

9 Further Developments **89**
9.1 Porting the System to Other Unix Versions 89
9.2 Exportation of Classified Data 89
9.3 Using other Access Control Models 90
9.4 Extending or Replacing the Sensor Subsystem 90

10 Conclusion **93**
10.1 Summary of Contributions 93
10.2 Future Work 94

Bibliography **99**

A CSP and VDM-SL Notation **101**
A.1 CSP Process Expressions 101
A.2 VDM-SL Symbols 101

B Installation Guide **105**
B.1 Installation of the Access Control Part 106
B.1.1 Changes made to the Stackable File System Templates 107
B.2 Installation of the Web-cameras and Motion Detection Programs 107

C User's Guide **109**
C.1 System Startup and Shutdown . . . 109
C.1.1 Guide to the Super User . . . 109
C.1.2 Guide to a Non-Privileged User . . . 111
C.2 The Web-Camera Sensor . . . 112
C.3 The Command Line Programs . . . 113
C.3.1 File Level Management . . . 113
C.3.2 User Level Management . . . 114
C.3.3 Window Management . . . 114
C.3.4 Sensors . . . 115
C.4 The Security Manager GUI . . . 115

D Testing **117**
D.1 Stackable File System Test . . . 117
D.2 File Level Management Test . . . 119
D.3 User Level Management Test . . . 119
D.4 Mandatory Access Control Test . . . 119
D.5 Window Management Test . . . 121
D.6 Editor Test . . . 121
D.7 Web-camera Sensor Test . . . 121

E GUI Screen-shots **125**

F Source Code **135**
F.1 Common Files . . . 135
F.1.1 mount_point.h . . . 135
F.1.2 seac_ipc.h . . . 135
F.1.3 seac_ipc.c . . . 137
F.1.4 security_policy_parameters.h . . . 139
F.1.5 sensor.h . . . 139
F.2 The Stackable File System Files . . . 139
F.2.1 macfs.fist . . . 139
F.2.2 security_policy.c . . . 148
F.2.3 file_levels.c . . . 150
F.2.4 user_levels.c . . . 154
F.2.5 seac_init.c . . . 156

F.2.6 initcl.c . . . 156
F.2.7 seac_destroy.c . . . 157
F.3 File Level Management Files . . . 158
F.3.1 getfl.c . . . 158
F.3.2 setfl.c . . . 159
F.3.3 listfl.c . . . 160
F.4 User Level Management Files . . . 162
F.4.1 getul.c . . . 162
F.4.2 setul.c . . . 163
F.4.3 listul.c . . . 164
F.5 Window Management Files . . . 166
F.5.1 visibility_manager.c . . . 166
F.5.2 file_open_monitor.c . . . 180
F.5.3 sensor_server.c . . . 181
F.5.4 listwl.c . . . 183
F.5.5 listsl.c . . . 184
F.5.6 getcl.c . . . 185
F.6 Editor Files . . . 186
F.6.1 x_create_window_interceptor.c . . . 186
F.6.2 backup_interceptor.c . . . 188
F.7 Sensor Files . . . 189
F.7.1 swsensor.c . . . 189
F.7.2 motion_handler.c . . . 190
F.7.3 camera_client.c . . . 195
F.7.4 event1.c . . . 199
F.7.5 event2.c . . . 200
F.7.6 pipe2.h . . . 201
F.7.7 start_motion.c . . . 201
F.8 GUI Files . . . 202
F.8.1 Exec.java . . . 202
F.8.2 SecurityManagerGUI.java . . . 206
F.8.3 BasicPanel.java . . . 209
F.8.4 InitPanel.java . . . 210
F.8.5 MessagePanel.java . . . 210
F.8.6 FileLevelPanel.java . . . 211

F.8.7 FileLevelTableModel.java 214
F.8.8 SimpleTableModel.java 215
F.8.9 UserLevelPanel.java 215
F.8.10 UserLevelTableModel.java 217
F.8.11 WindowPanel.java 217
F.8.12 WindowTableModel.java 220
F.8.13 SubjectLevelPanel.java 220
F.9 System Administration Scripts 221
F.9.1 reset.sh 221
F.9.2 startup.sh 222
F.9.3 shutdown.sh 222

List of Figures

1.1 Sensor deployment 2

2.1 The Onion Model 9
2.2 The Garlic Clove Model 9
2.3 Fundamental Access Control Model 10
2.4 An Example Security Lattice 12
2.5 The Bell-LaPadula Model 13

4.1 The X Window System 26
4.2 The virtual file system 28
4.3 Stackable file systems 29
4.4 Stackable file system development using FiST 29

5.1 Mediation of access to a file 36
5.2 Mediation of access to a window 37

6.1 Layered Architecture with subsystems. 42
6.2 User space process interaction. 43
6.3 CSP processes and channels in the system. 52
6.4 CSP specification of the visibility manager process. 60
6.5 CSP specification of the visibility manager process, continued. 61
6.6 CSP specification of the MAP_WINDOWS process. 62
6.7 CSP specification of the UNMAP_WINDOWS process 62
6.8 CSP specification of the client processes 63

7.1 FiST grammar outline 66
7.2 Sample window status information 76
7.3 The Camera Software Architecture 79
7.4 Pseudo-code for the main `motion_handler` loop 81

E.1 GUI Screen-shot: File Level Management 126
E.2 GUI Screen-shot: File Level Management, including browse dialog window . 127
E.3 GUI Screen-shot: User Level Management, non-privileged user view 128
E.4 GUI Screen-shot: User Level Management, super user view 129
E.5 GUI Screen-shot: Unmapped Windows 130
E.6 GUI Screen-shot: Unmapped Windows, names of open files shown 131
E.7 GUI Screen-shot: Mapped Windows 132
E.8 GUI Screen-shot: Current Subject Levels 133

List of Tables

5.1 Sensor enhanced access control model terminology 34

A.1 CSP notation for processes and process expressions. 102
A.2 VDM-SL data types. 102
A.3 VDM-SL operators. 103

D.1 Default Users and Files 117
D.2 Policies used in testing 117
D.3 Test of the Stackable File System by the Super User 118
D.4 Test of the Stackable File System by a User 118
D.5 Test of the File Level Management 119
D.6 Test of User Level Management 119
D.7 Test of the Mandatory Access Control 120
D.8 Test of Window Management Tools 121
D.9 Test of the Window Management 122
D.10 Editor and Viewer Test Cases 122
D.11 Test of Editors and Viewers 122
D.12 Test of the Camera Sensor 123

Chapter 1

Introduction

1.1 Motivation

Access control is traditionally restricted to logical access control where access only is granted to authorized users in authorized locations. The authorization of a user is usually determined after the user is identified and authenticated by the system, using for example a user name and corresponding password. After such a login procedure, a user ID will be associated with every process the user starts. The user ID will be used to determine which resources the process is permitted to access. This form of access control is logical because it is a process and not a person in front of the computer that accesses a given resource in the system. After a user has logged in, an unauthorized person may also be able to obtain physical access to the computer. This could, for instance, occur if the user temporarily leaves his computer unattended. Another more severe scenario could be that an intruder uses a weapon to threaten the authorized user to provide access to the resource; the access would then be obtained despite the user's physical presence. In any case, the processes that are started on the computer cannot determine who are currently present in the environment. They only know the user ID of the person who is currently logged in. In a military or commercial setting, it can have severe implications if an unauthorized person obtains access to classified information by circumventing the logical access control.

1.2 Sensor Enhanced Access Control Model

The logical access control limitations of traditional computers can be evaded in an environment where a sensor is used to detect the presence of unauthorized persons. A sensor can detect when an unauthorized person enters or leaves the surveyed environment. Depending on the capabilities of the sensor, it may detect an ID that uniquely identifies the unauthorized person, or it may be very basic and only detect whether someone is present or not. To provide a more fine-grained detection method, more than one type of sensor can be deployed in an environment. Regardless of the sophistication of a sensor, it must as a minimum be able to detect whether a person enters or leaves the environment so that a list of all the currently present persons can be maintained. A possible sensor deployment is shown in Figure 1.1.

In the logical access control part of our system, the term user will be used to denote the authorized person who is identified by the system using a login procedure. A user obtains

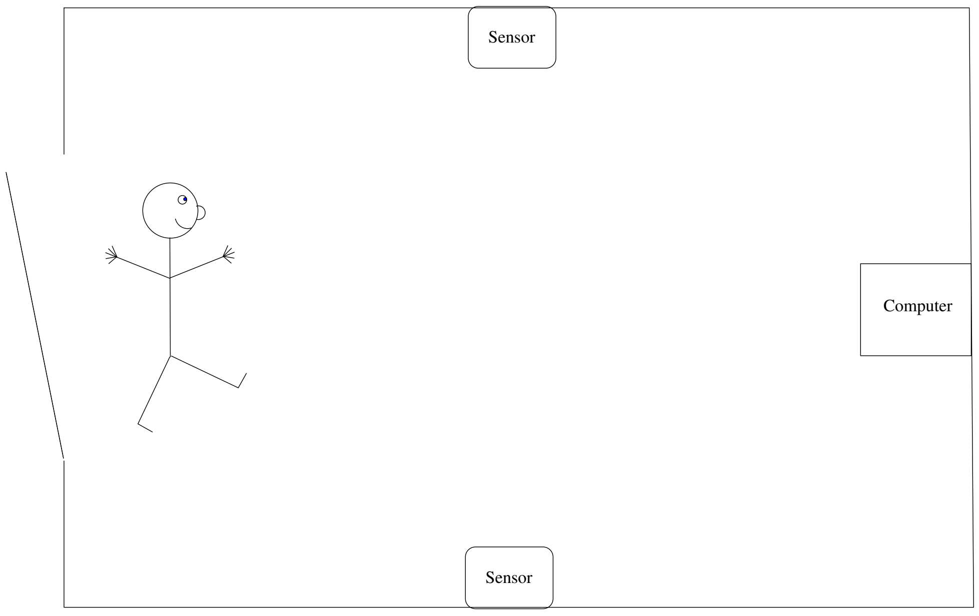


Figure 1.1: A possible deployment of two sensors in an environment. The sensors will detect whether the person enters or leaves the environment.

logical access to a file via a process. There are many possible access operations available for processes, but our model only encompasses read and write access to files. If the logical access control denies a process read or write access to a file, the operating system will generate an appropriate error message.

In the physical access control part of our system, the term person will be used to denote anyone who is detected by the sensor. If the user has an editor open and it contains classified data, the physical access control may deny the person access to view this data. In case the access is denied, the editor will disappear from the display so that it no longer is possible for humans to physically see it.

The rules that determine whether an access operation should be successful or not are defined by the access control model. There exist many possible access control models that meet different security requirements. We have chosen to use multilevel security models at the core of our Sensor Enhanced Access Control Model since these are frequently used in environments where classified data are processed. In our model, each subject and object should be assigned a level by a central authority.

The combination of logical and physical access control is modeled by the clearance level. The clearance level denotes the combined subject level for all the unauthorized persons and the user in a given environment, and it will be equal to the minimum of all the detected subject levels. A security policy must be stated by a trusted authority and it is used to determine whether an editor that contains classified data should be visible or not. Our model does not enforce a particular security policy since we have developed a flexible model where parameters are used to specify the security policy. This is in line with the general Unix design philosophy of providing mechanism, not policy.

1.3 The Developed Prototype

A prototype that conforms to the Sensor Enhanced Access Control Model has been developed. The system is designed so that it can be integrated with an existing Unix system. The files and users can therefore be uniquely identified by an inode number and user ID, respectively. The core of the system provides logical access control using file and user levels associated with inode numbers and user IDs.

The part of the system that stores and retrieves file and user levels and mediates the access to files by users is a stackable file system. A stackable file system is a layer that resides in the kernel above a native file system and below the Virtual File System. It is very important from a security perspective that the access mediation part of the system (a.k.a. the reference monitor) is implemented in the kernel since it then is protected from non-privileged users via the operating system user/kernel modes. Furthermore, including the access control mechanism in the kernel gives better performance because fewer context switches have to be made.

The sensor part of the system has been implemented using two web-cameras and a motion detection program. The program cannot differentiate the detected persons and a common level must therefore be assigned to all persons. This level should depend on what other physical access control measures are deployed in the environment. In addition to the web-camera sensor, a very simple software sensor simulator has been implemented so that it is possible to test the system without the deployment of web-cameras.

The part of the system that manages the visibility of the windows is based on the standardized protocol, which is used in the X Window System. The X Window System is a network-based graphics windowing system that is commonly used on Unix systems. It assigns a window ID to each created window and provides functions for making a window visible or invisible. Our system associates a window level with each window ID, and by using this level and the environment level received from the sensor, the system can make access control decisions about whether a window should be visible or not.

Finally, many command-line programs have been implemented for managing the system. Most users will, however, probably prefer a GUI for managing the system and providing an overview of the stored data. For this purpose, a GUI has been developed which constitute a presentation layer for the system.

1.4 Contributions

The conceptual contribution of our work is a Sensor Enhanced Access Control Model that combines logical and physical access control. The model uses information from physical sensors to determine a combined security level for all the persons currently present near the computer. By monitoring whether or not unauthorized persons are near the computer, a system based on this model can determine what information should be visible on the computer display. The consequence of this is that computers with sensitive information can be placed in unprotected environments where unauthorized persons have physical access to the premise of the computer.

The technical contributions can be divided into three modules:

A kernel module that conforms to a multilevel security model where levels are assigned to users and files. The access control decisions made by the module are determined by security policy parameters. These parameters can, for instance, be set so that the system enforces the Bell-LaPadula model or the Biba model and thus addresses confidentiality or integrity, respectively. The design decision regarding the parameters has been made so that the system, to the extend possible, provides mechanism and not policy. Furthermore, the module can be used independently of the other modules to form a system that only provides logical and not physical access control.

A simple movement sensor based on two web-cameras. The web-cameras are only used to determine whether someone enters or leaves the environment and a common level will therefore be assigned to all the detected persons.

A window management module that combines the logical access control implemented by the kernel module with the physical access control information received from the sensor module. The main responsibility of this module is to ensure that unauthorized persons cannot view classified data on a computer display.

1.5 Thesis Organization

The problem that we are set to solve in this thesis is that of combining logical and physical access control using input from sensors. The organization of the main chapters of the thesis can briefly be summarized as follows: Chapter 2 to Chapter 4 will lead the reader into our problem area by reviewing the state of the art and presenting some background information. Chapter 5 to Chapter 7 describe how we solved the problem, and Chapter 8 to Chapter 10 evaluates our solution and provides guidelines for future work.

A more detailed overview of the thesis is presented in the following. In Chapter 2, a brief description of some basic security concepts is provided. The main purpose of this chapter is to provide a survey of the state of the art within logical and physical access control. The logical access control models are for instance the Bell and LaPadula model and the Biba model, and the physical access control models are the Onion model, and the Garlic Clove Model.

Chapter 3 describes different sensors and motion detection technologies. This includes a review of the main services provided by sensors and some examples of current devices that are used as sensors. Furthermore, the mathematical foundation of motion detection, based on image analysis, is described briefly.

Chapter 4 provides a brief overview of some Linux technologies which the reader should be familiar with in order to understand the description of the implementation in Chapter 7. The focus will be on technologies such as the X Window System, shared libraries, system calls, kernel modules and stackable file systems.

In Chapter 5, the Sensor Enhanced Access Control Model is described in detail. In particular, the different types of subjects and objects are described along with the available access operations. A special type of user, the super user, is also introduced; this user is special because it is fully trusted and its access to objects is therefore not restricted by any of the access control mechanisms in our model. The chapter ends with a description of the security parameters that can be used to create a security policy for a given deployment of the system.

In Chapter 6, the system design is described in terms of a number of subsystems. The `file level management` and `user level management` sub-systems provide programs for setting and retrieving file and user levels, respectively. The `stackable file system` provides storage for the file and user levels and enforces the access control policy set by the system administrator. The `window management` subsystem ensures that windows are made visible or invisible, depending on the input received from the `sensor` subsystem. The subsystems consist of many processes that must communicate in order to provide the required functionality. The chapter ends with a CSP-specification of this inter-process communication.

In Chapter 7, it is described how the different subsystems are implemented. In particular, it is described and motivated which technologies have been used. Some issues that were not foreseeable during the design phase are also described; this is, for instance, the issue of handling backup files created by an editor. The GUI subsystem is also introduced. It is only a presentation layer and does not add any new essential functionality to the system.

In Chapter 8, we evaluate the system. More precisely, we describe how the system was tested by first testing the individual sub-systems and finally testing the entire system.

In Chapter 9, the further development possibilities are described. This will in particular revolve around the sensor part of the system, which can provide a more usable system if the sensors are capable of detecting the level of the persons.

Finally, in Chapter 10 we review the contributions of this thesis and give directions for future work.

We follow with several appendices. In Appendix A, the CSP and VDM-SL notation used in the specification in Section 6.8 is described briefly.

In Appendix B, directions for installing the system is provided. Most of this require super user privileges.

In Appendix C, a user's guide describes how the system can be used. This will first include a description of how the super user and a non-privileged user can initialize and shutdown the system. It is followed by a reference guide for all the created command line programs. Some of these programs will require super user privileges.

In Appendix D, all the test cases and expected results used in our test are listed, followed by the test results. In the tests where program output where available, this is listed too.

In Appendix E, some screenshots from the Security Manager GUI can be seen. The GUI, and the screen-shots of it, gives an overview of the system since one can easily see how file and user levels are managed, the information stored about visible and invisible windows, and the subject levels of detected persons and the user.

In Appendix F, all the source code is listed. To the extend possible, the code is listed according to which subsystem it belongs to.

Chapter 2

Security and Access Control

Computer security is a widely investigated topic, as computers in any organization often contain or are valuable resources. A general background of important topics in computer security is given in Section 2.1. Access control is one way to protect the assets in a computer system and is described in Section 2.2. Models and mechanisms grant or deny access to resources. Access control can be both physical and logical. Physical access control protects physical valuables using physical means, for example a locked door restricting access to a printer. Logical access control protects logical resources using logical means, such as a password restricting access to a file.

2.1 Computer Security

Computer Security deals with securing assets in a computer system. This implies that the computer system contains something valuable that requires protection. This could be information, processing power and the like. Usually, computer security deals with the following three aspects [21]:

Confidentiality: Unauthorized disclosure of data should be prevented. Confidentiality is also known as secrecy, and this is what normally comes to mind when people think about security.

Integrity: Unauthorized modification of data should be prevented. Accidental modification of data should also be prevented.

Availability: Denial of authorized access should be prevented.

When dealing with computer security it is important to consider what should be protected and from whom. It is also necessary to conduct a cost-benefit analysis to determine the level of security to implement. When doing this, it is important to keep in mind that cost is not only the monetary cost of implementing a security system, but also the cost in inconvenience and ineffectiveness to the users of the system.

To achieve the goals of confidentiality, integrity and availability, several methods have been developed, as well as attacks to circumvent those methods. Computer security topics cover a wide area of different models, technologies and methodologies. Methods exist to keep communication secret, e.g. cryptography, and to prevent or detect unauthorized access to resources using technologies like firewalls and intrusion detection systems. Availability might also determine how to set up a mail-server so that it will not be used to relay foreign mails, showing the breadth of the computer security subject. Methods dealing with restricting access to information are called access control. These will be the main focus of this security discussion, and will be described in depth in the following.

While access control governs immediate access to objects, information flow models take implicit information flow into account. The concept of information flow control was developed by Denning [19]. An information flow model seeks to consider every kind of information flow, including flow through so-called covert channels. This is implicit information, such as the information you can get from being denied access to a resource. The advantage of an information flow model is that it takes every kind of information flow into account, while access control models might only consider the explicit information flow. It is on the other hand more difficult to design a secure system based on information flow models than on access control models [21].

At the most basic level of computer security, we have identification and authentication. Identification is concerned with stating who a user is, usually done by the user himself. Authentication means proving who you are to the system, and the system's verification of your claimed identity. This is most frequently done with a password, only the user knows the password to her account. It could also be done with more advanced methods, for example fingerprint scanning. Knowing the identity of users is important when making access control decisions. It is also important when the system needs to keep track of its users, for instance to log behavior. It should be noted that there exists a number of threats to password authentications, where social engineering is not the least effective.

2.2 Access Control

Access control revolves around granting or denying access to resources, and it deals with what information may be accessed by which users. Normally, access control cannot take place unless the user is properly identified and authorized, as the system usually exercises access rights based on user identity.

Access control consists of two components, physical and logical access control. Physical access control protects physical assets such as hardware, and printed files. It deals with access that is denied or granted in the physical world, and it takes place outside the system via physical means. This could be guards restricting access to a building or magnetic cards restricting access to a room. Logical access control protects logical assets such as information, (computer) resources and etc. It concerns how a computer system internally grants or denies access to the logical assets and takes place within the system. Logical access control is usually based on constraints placed on users of the system and the information they seek to obtain.

There is an important overlap of logical and physical access controls when logical assets get embodied in physical assets. This is for example when a file is printed or displayed on a computer monitor. In this case a logical asset become physical one, and requires more than logical access control to protect it.

2.2.1 Physical Access Control

Physical access control is normally used to grant or deny access to physical assets using physical means. It deals with restricting access to physical premises, such as persons entering a building. The physical access control can for instance be guards allowing or disallowing access or cameras recognizing a person to determine whether to open a specific door or not.

Physical access control creates levels of protection, where each level protects some sensitive assets, and persons trying to access these must undergo access control. One model for physical security is the onion model, illustrated in Figure 2.1. It is a layered security model that describes the existence of transitions from outer layer to the innermost one. The inner layer contains the most sensitive assets, and to get there you must pass through the outer layers. The transitions could be guarded by receptionists, guards, badge readers etc. Once a person is allowed into one layer, he is allowed to go anywhere within it. A person cannot go between layers without being subjected to control [40].

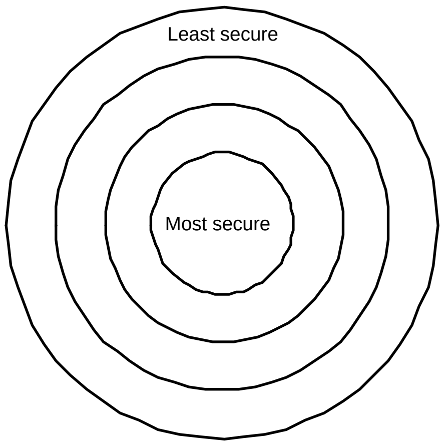


Figure 2.1: The Onion Model

The onion model is very simple. A more realistic model is the garlic clove model shown in Figure 2.2. It takes into account that not all layers are consecutively more and more sensitive, and that different classes of people can be restricted from some assets, while having access to others [40].

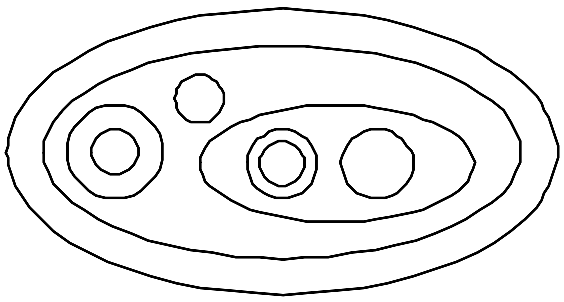


Figure 2.2: The Garlic Clove Model

When designing physical facilities one needs to make sure that a person can only move between layers in the ways explicit permitted. This includes limiting the risk of unauthorized access through various 'back doors'. The 'back doors' includes behavior such as entering a building through a window, or closely following a trusted person and entering a door which was opened by that person. While it might not be feasible to prevent all unauthorized access, it is still worth considering how problematic it is for an intruder to gain access. It might not be difficult to break in a door or smash a thin wall, but it is obvious if it is done as it is noisy. If the door is left unlocked, the intrusion will be noiseless [40].

2.2.2 Logical Access Control

In most cases, logical access control governs admission to logical assets such as information and an important part of it is the policies or mechanisms the system uses to grant or deny the access.

The basic entities of the access control model are subjects and objects. Subjects are active, operating on objects. Objects are passive, being operated upon. Subjects try to access objects, and can e.g. be users or processes. Objects are resources that can be accessed or used, and are usually files, printers etc. When a subject tries to operate on an object, a reference monitor decides if access will be granted or denied. This is shown in Figure 2.3, and this is a basic model of access control [21].

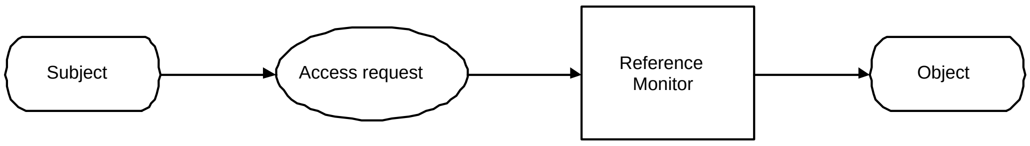


Figure 2.3: Fundamental Access Control Model

Access control models usually define a number of possible, or allowed access operations on files. At the most basic level we can define two modes; observe and alter. Many systems uses 3-4 modes. The well-known Bell-LaPadula model has the operations read, write, append and execute. To read is to observe, write is to observe and alter, while append is to alter without observing. Execute is hard to describe in terms of observe and alter, as it can be neither, but in reality it is often difficult to implement execution without observation [21].

Access control systems can either be mandatory or discretionary. Mandatory access control (MAC) is a system wide policy that decides which users should have access to what files. Discretionary access control (DAC) is where the owner, or a trusted individual, decides access control over a file. This means that the owner of the file has discretion over who should have what accesses to the file. Thus it is up to the user to decide 'how secret' he considers the file [21], [24].

In some systems, mandatory and discretionary access control are used simultaneously. For a user to access information he needs to be allowed to access it in relation to the global, mandatory rules, and the owner of the file needs to have made it possible for him to do so. If either of the two mechanisms fails to grant access, it will be denied [36].

When a system has any kind of access control, it needs to have some way to maintain and determine which access rights have been given. This is generally modeled by an access matrix.

An access matrix consists of the set of access operations a given subject can perform on a given object. The matrix is usually a $m \times n$ matrix, where m is the number of subjects and n is the number of objects. The element at (i, j) then lists the access operations subject i can preform on object j.

Usually, an access matrix is very sparse, making it ineffective to store. Therefore the information is stored in capability and/or access control lists. A capability list is a list for each user, showing what that particular user has access to. This corresponds to storing the rows of the access matrix. An access control list is a list for each file, stating which users can do what access operations on the file. An access control list is equivalent to storing the columns of the access matrix. The concept of an access matrix, as well as the idea to store it as lists are presented in Lampson's *Protection* [23].

In military access control the system usually operates with security levels or labels. A piece of information is assigned a security label, defining the level of that information. A user or process is assigned a clearance, which is also a security label. The labels are then arranged hierarchically, and a user may only access files with level up to and including his clearance. An access control model that makes use of levels belongs to the class of multilevel security models. A security level is said to dominate another if the level is above the other. The level that dominates all other levels is normally called system high, while the level dominated by all other levels is known as system low [21]. Security levels are partially ordered, meaning that the ordering relation is reflective, transitive and antisymmetric, and that two levels do not have to be comparable.

An important concept in multilevel security models is security lattices. A security lattice emerges when you not only have levels, but also departments subjects and objects can belong to. This means that even though a user is cleared to see very sensitive material in one department, he might not be cleared to view any information from the other departments. The department and level pair gives the security label of the file. This gives a lattice of paired levels and departments with a partial ordering. The partial ordering of security labels is important here, as two files of information can belong to different departments, and their security label will then be incomparable.

As an example of a multilevel security lattice we can have the security levels *high* ad *low* and the departments MARINE and NAVY. The security labels would consist of a security level and one or two departments. The following relations would hold in the lattice:

$(\mathit{low}, \{\text{MARINE}\}) \leq (\mathit{high}, \{\text{MARINE}\})$

$(\mathit{low}, \{\text{MARINE}\}) \leq (\mathit{low}, \{\text{MARINE, NAVY}\})$

$(\mathit{low}, \{\text{MARINE}\}) \not\leq (\mathit{low}, \{\text{NAVY}\})$

The lattice this forms can be seen in Figure 2.4. This shows that (*high*, {MARINE, NAVY}) is system high and dominates all other levels, while (*low*, $\emptyset$) is the system low. This system will ensure that while a user might be cleared for *high* level information in MARINE, he will not have access to information in NAVY. It gives a more flexible way of restricting and giving access to information.

The Bell and LaPadula Model

One of the best known multilevel security models is the one proposed by Bell and LaPadula in *Secure Computer Systems: Mathematical Foundations* [15] and *Secure Computer Sys-*

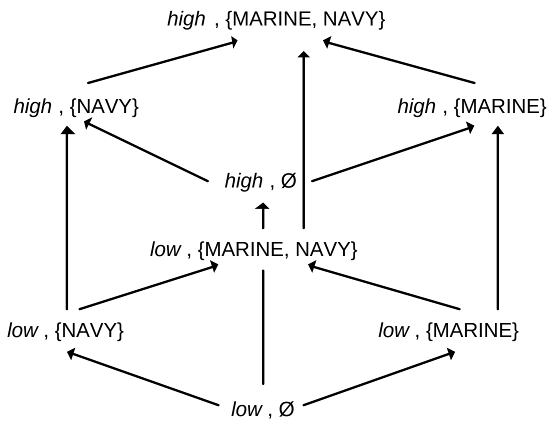


Figure 2.4: An Example Security Lattice

tems: A Mathematical Model [25]. In this text we have used the slightly more modern and informal way of describing it given in Dieter Gollmann's *Computer Security* [21].

The security model can be described by a state machine model, where the system is in a state which is either secure or insecure. A change (i.e. transition) in the system will then lead to another state. The main idea is to avoid transitions out of a secure state and into an insecure state.

The system consists of a set of subjects, objects, access operations and partially ordered security levels. The states of the systems are rather complicated. The state set is defined as $B \times M \times F$ where B is the set of current accesses, M is the access permission matrix and F is the set of security level assignments. The last part has three parts, the classification of objects, as well as both the maximal and current security level of the subjects.

The Bell-LaPadula model uses four different access operations: Read, write, append and execute. In the original interpretation, to write is to alter the text while viewing it, so it implies both reading and writing. On the other hand, append is writing without reading. In some descriptions of the Bell-LaPadula model the append operation is not used. Usually, the write operation then means to write without reading. In the following write will be used in this meaning, as a operation with no observation.

The model defines two security properties which should be satisfied for a state to be secure, the simple security property (ss-property) and the star property (*-property).

ss-property: This property defines that there can be no read up. This means that for read access operations on all subject/object combinations in b, the security level of the object will not be higher than the (maximal) level of the subject accessing it for read.

The ss-property is not necessarily sufficient to prevent a low level subject from reading a high level object. This is because a high level subject (which could be a program) could write the content of the high level object in a low level object, which the low level subject then can access. The low level subject could be the one to create the high level subject, thus compromising the security of the system. To get around this, the *-property is defined.

***-property:** This property defines a no write down policy. It states that for each element in b, where the access operation is write, the (current) security level of the subject should be lower than or equal to the object.

The following example will show the importance of the *-property. A user **A** who is cleared to *high* level information creates a file ***a*** with sensitive information. It will have the security level *high*. Another user, **B**, only cleared to *low* level information, creates *low* level file, ***b***. He also creates a program, `smart_program` and convinces user **A** to use it. `smart_program` will now be run by user **A**, thus the program's executing process will have the level *high*. `smart_program` can then read the secret file ***a***, as it has the correct security level. `smart_program` is written so it will take the information stored in ***a*** and write it to the *low* level file ***b***. Thus user **B** has gained access to the information in the secret file ***a***, violating the security of the system by creating a so-called Trojan program. The *-property ensures that this cannot happen, as `smart_program` would not be allowed to write in a file with a lower security level [38].

The information flow in the Bell-LaPadula system can be seen in Figure 2.5 [39]. When a subject reads an object the information flows from the object to the subject. When a subject writes an object, the information flows from the subject to the object. As seen in the figure, the rules of the Bell-LaPadula model ensures that information can only flow from a lower level to a higher, not the other way around.

This means that a high level subject can read a low level object, but only write objects on its own level and above (using an appending write operation). It cannot alter those on a higher level, due to the ss-property. To make a higher level subject access a lower level object for write or append, you must either temporarily downgrade the subject, or you must define a set of trusted subjects that can violate the *-property.

If the subjects are processes, the first way is feasible, as they would be able to 'forget' what they know at a higher level, as their security level indicates which files they can read and that is what they know. This approach does not hold if the subjects are humans. Then the system would need to have a number of trusted users that may violate the security policy.

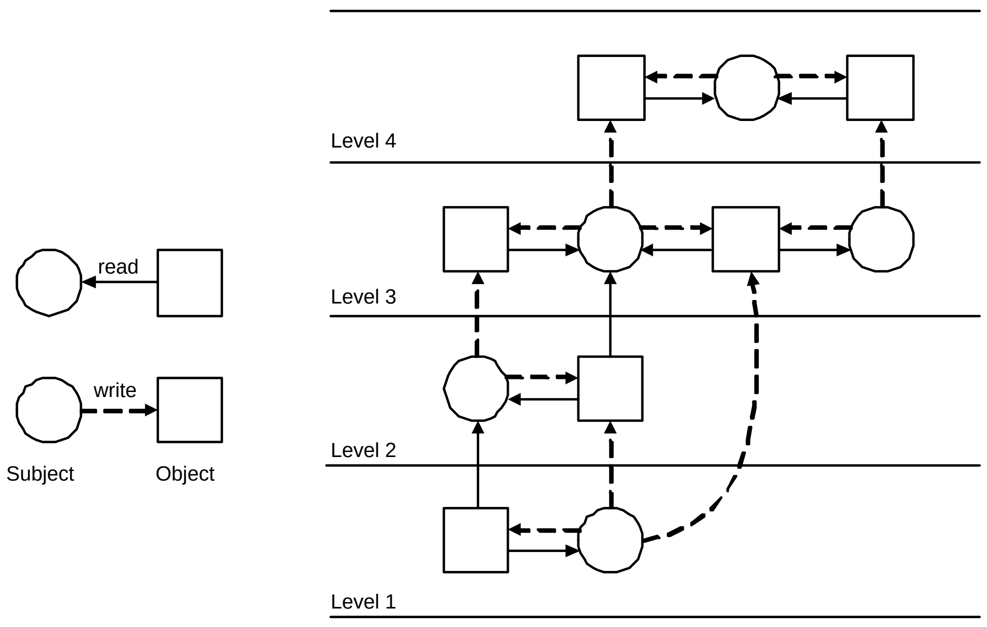


Figure 2.5: The Bell-LaPadula Model

The Bell-LaPadula model has a number of assumptions, one of them being that the classifications do not change in normal operation. This is an aspect of the model that the developers decided on from their systems analysis [25].

In their papers, the authors define and prove a lot of properties of the system that they have modeled. This is important because a system that is built using their model will be known to satisfy certain security properties.

The Biba Integrity Model

Biba [21][1] developed a Bell-LaPadula-like model to contain integrity, as the original Bell-LaPadula model did not include this aspect. The model is interesting because of this focus on integrity, it shows that an access control model also can be used to obtain this security goal.

The need for integrity in a system can be seen when considering the needs of the organization the system should support. In a military system focused on confidentiality the important part is that the secret code to the rocket launcher can only be read by the General and not by the Captain. In a business cooperation the opposite might be true - while the overall business strategies that the CEO has written should be known by every employee in the company, a secretary should not overwrite them with his own ideas. Thus, the important thing is not who can read the information, but who has access to write it. The system should be able to guarantee a certain level of integrity.

The main idea in the Biba model is that low integrity information should not be allowed to flow to high integrity objects, but the opposite is permitted [35]. Thus, the information flow is from high to low, the opposite of a confidentiality system like the Bell-LaPadula model. The Biba model also has rules corresponding to the Bell-LaPadula rules:

simple-integrity property: This defines that there can be no write up. Subject s can write object o only if the integrity level of s is higher than or equal to the integrity level of o.

integrity *-property: This defines that there can be no read down. A subject s can read object o only if the integrity level of s is less than or equal to the integrity level of o.

If the Bell-LaPadula and the Biba models are combined, they can be enforced using the same mechanism. However, if the integrity level and the security level have the same label, this results in conflicting restrictions. This will simply mean that a subject can only read and write information at their own security level, and this will result in a trivial system [35].

If, instead, different labels can be used for confidentiality and integrity, we will obtain a useful system. The following rules can be defined in terms of subject s and object o:

1. s can read o only if the confidentiality level of s is greater than or equal to the confidentiality of o, and if the integrity level of s is less than or equal to the integrity level of o.

2. s can only write o if the confidentiality level of s is less than or equal to the confidentiality level of o, and if the integrity level of s is greater than or equal to the integrity level of o.

[1]First described in *Integrity Considerations for Secure Computer Systems* by K.J. Biba, Technical Report ESD-TR76 -372, MITRE Corp., 1977. The document was unavailable.

This model is described in Ravi S. Sandhu's *Lattice-Based Access Control Models* [35] and called the composite model.

The Chinese Wall Model

The Chinese Wall security model was first described by Brewer and Nash in *The Chinese Wall Security Policy* [17]. They showed that in the financial world the Bell-LaPadula model could not fulfill the sector's specific needs, and other security models were needed.

In a consultant business or similar, a specific consultant will probably possess insider knowledge about his clients to fulfill his job. The company itself might have several clients in the same market, and thus if consultants worked for multiple clients in the same field, it would lead to conflicts of interests. A consultant should not have insider knowledge of two clients with conflicts of interest, but he may have it on two or more clients with no conflicts. The basic rule of the Chinese Wall model is that there must be no information flow that causes a conflict of interests [21].

The wall metaphor is that no subject can access an object on the wrong side of the wall. Information that the consultant has access to is inside the wall, while information belonging to clients with conflicts of interests, with respect to the information the consultant already has access to, is outside the wall [17].

The model is described in *Computer Security* [21] by:

- A set of companies C and a set of objects O, with each object belonging to a company in C.
- A set of subjects S that consists of the analysts/consultants that the company has employed.
- All objects relating to one company are collected in a company dataset. The function $y : O \to C$ gives the company dataset of each object.
- Conflict of interest classes are defined and each conflict of interest class covers companies that are in competition. The function $x : O \to P(C)$ gives the conflict class for each object, i.e. the set of all companies that should not learn about the content of the object.
- The security label of an object o is $(x(o), y(o))$.
- Sanitized information has been cleaned for sensitive information, and access to it need not be restricted. For a sanitized object o, we set $x(o) = \emptyset$.

The main idea of this model is that we need to keep track of history. A subject must not previously have had access to a company with an interest conflict with the one he tries to access. The rule to ensure this is the following:

ss-property A subject may only have access to an object o, if he previously not has had access to an object in o's conflict of interest class or if the objects belong to the same company as o.

Note that this gives the consultant a freedom of choice, when he has not yet accessed anything in a conflict class. He can freely choose an object from any company.

There will still be a problem if two different companies with conflicts use the same external resource, i.e. a bank. If an analyst dealing with one of the companies writes some sensitive information about that company in an object belonging to the bank, another analyst might read it when accessing the bank's files. Thus, an insecure indirect information flow has occurred. To prevent this, a *-property is defined to govern write access:

***-property** Subject s can only write object o if s has no read access to an object $o^{'}$ where o and $o^{'}$ belong to different companies and $o^{'}$ is not sanitized.

An important aspect of the Chinese wall model is that each actual user (each person) only can have one user account on the system, as a person will store knowledge in his brain [35]. Another important aspect of the Chinese wall model is that the access rights are dynamic and change over time.

Other Access Control Models

There are other access control models, some of which will be briefly mentioned here. They are different in their approach than the ones presented above.

User Groups A different way of keeping track of permissions is to have user groups on the system. A user is a member of one or more groups, and a group has permissions to some files. Single users can also have negative permissions, excluding them from accessing a file a group they are a member of can access. Thus the group system can be very complicated in theory. When it is used on Unix systems it is simple, with users only belonging to one group and no negative permissions. It is important to notice that this system does not use security levels. It is not designed for a hierarchical military system, but for a multi-user environment where users naturally belong to information-sharing groups.

Protection Rings When referring to subjects as processes rather than users, protection rings can be used as access control. The protection rings are a very simple mechanism that has processes running at different levels. Usually, the most important processes are the kernel processes and the least important are the user processes. Objects are numbered in the same way as the subjects, and access is granted or denied by comparing levels. The protection ring model is much closer to the system than the other approaches discussed

Role Based Access Control Another form of access control is role based access control. It gives access based on the role of the users of the system, meaning the job a certain user is supposed to preform. Each user can have more than one role, and each role can be assigned to more than one user. Role based access control models are a different approach, where the user's actual needs while he is using the systems are the basis for what he can access.

Orange Book Security

The United States Department of Defense (DoD) have spent resources and research effort on secure systems. In 1985 they published the DoD standard DoD 5200.28, called The Orange Book [13]. It divides operating systems into categories depending on their security properties. The standard has been replaced by a more complex one, but it is still a guide to security properties [39].

The Orange Book has 4 major divisions for operating systems, from A to D. C is split up into C1 and C2, and B into B1 to B3. The Orange Book describes in detail what requirements a system must meet to be assigned to a certain division. The requirements are very elaborate covering everything from policies and mechanisms to testing, verification and user guides. The following is a brief overview of the divisions and their main requirements [13, 39]:

Division D: Minimal protection, no requirements at all. Windows 95/98/Me falls into this category.

Division C: Discretionary protection for systems with cooperation users. C1 requires discretionary security protection and a separation of users and data. C2 requires some more finely specified user controls. The Unix rwx scheme meets C1, but not C2.

Division B: Mandatory protection. The system must be capable of enforcing the Bell-LaPadula model. B1 requires labeled security protection, B2 structured protection and B3 security domains. All of this is in addition to the requirements of C2.

Division A: Verified protection. A1 systems must meet B3 requirements and have a more formal design specification and verification. Covert channels must also be analyzed. Beyond A1 is for systems with an even more rigorous analysis, design and verification, as well as room for the inclusion of future properties.

2.3 Summary

Computer security seeks to protect assets on a computer. The goals are confidentiality, integrity and availability, and many different methods and technologies exist to obtain these goals. One of these is access control. This revolves around granting or denying access to resources. This can involve both physical access control, where physical assets are protected for instance with guards or locked doors, and logical access control, where logical assets like information on a computer is protected.

The basic entities of logical access control are subjects and objects. Subjects can use different access operations to gain access to objects, like read or write. It is important to know exactly what the different operations encompass, for instance if to write also includes reading the modified data.

One basic distinguishing feature of access control models is the difference between mandatory and discretionary access control. In discretionary systems the access control is at the discretion of the users. In mandatory access control a system-wide policy exists. This means that for every resource there will be a policy on who can access it or not and this control is beyond the decisions of the single user.

Mandatory military access control systems levels are often multilevel security models. Files and users have levels, and rules are made to govern how subjects can access objects depending on their levels.

An important model is the Bell-LaPadula model. The main rules and center of the model is that there can be *no read up*, i.e. a user cannot read a file with a higher level than himself, and there can be *no write down*, i.e. a user cannot write to a process or file that has a lower security level than himself. Information can flow from a lower to a higher, but not the other way around.

An interesting modification of the Bell-LaPadula model is the Biba integrity model. It uses the same type of rules, but governs integrity instead of confidentiality. The rules specify that there can be *no read down* and *no write up*. Thus information can flow from a higher level to lower.

There are also many other access control models to cover different needs. A very well known model is the concept of groups, as this is used in the Unix operating system. The Chinese wall is a security model for professional companies. It does not have levels, but rather classes of conflict of interests. Information in the same conflict of interests class should be obtained by the same subject. Due to this, it also keeps track of history so the information a subject previously has accessed is known.

This chapter has described some general subjects in computer security, in particular access control. Both physical and logical access control were discussed, and some different models were described. When designing a system with access control, it is important to chose the right model, as they have different purposes, for instance they might seek to ensure integrity or confidentiality.

Chapter 3

Sensors and Motion Detection

Sensors are used to produce data for a system, an overview of which is given in Section 3.1. The main use of sensors is to transform physical properties into data a system can use. One of the things that sensors can be used for is to provide data for motion detection. This can be done in different ways depending on the sensor used and the sophistication of the algorithm used. Some motion detection methods are presented in Section 3.2.

3.1 Sensors

Sensors are devices that gather data and pass it on to a system. A sensor might be a physical device, or it might be a logical entity that produces data for the system. Physical sensors transform information about the physical world into data understandable by a computer [42]. Examples of physical sensors are cameras, infrared sensors, thermometers, barometers, RFID tags (Radio Frequency Identification tags) and pressure sensitive floors. Logical sensors can be things like event timers or load indicators.

In this context, sensors are devices that are used to detect things about their environment and pass it on to the system. The system will then process the information gathered by the sensors to present an impression of the environment that is needed on the system or to the user.

Sensor outputs are rarely useful unless the system has a direct connection to the sensed input, so there is a simple mapping between the system and the sensed input. If this is not present, the data the sensor produces must be processed using more or less advanced algorithms [42].

Sensor systems can be either active or passive. Active sensor systems are systems that interact with their environment and sense how their actions affect it. The sensors actively probe into the environment to sense a change. An example of an active sensor is a touch sensor. Passive systems sense ambient radiation or signals, passively receiving information. An example of a passive sensor system is a Global Positioning System [42].

When deploying several sensors to gather information, it will be necessary to fuse their views to form a unified image of the world. Each sensor can return errors, and the fusion engine should take this into account to develop a unified view with the least possible errors. There are several advanced methods for dealing with this problem [42].

Sensors can be used for a wide variety of purposes. Simple tasks such as to learn the state of a system, like the temperature in a chemical process, or complicated tasks like asserting the location of an object in an office environment. When trying to gain information about persons in an environment, the data the sensors deliver should often have the ability to reveal the presence or absence of motion. Sensors such as cameras, light detectors or infrared sensors are often used detect motion.

3.2 Motion Detection

Motion can be detected with a variety of sensors. An example of simple motion detection is a light sensor in a doorway. When someone or something passes through, the motion is detected and a signal can be emitted. A more sophisticated example would be a security camera that not only detects the motion, but attempts to track it as well. Machine vision also uses motion detection when a robot tries to navigate in real time.

Motion can be detected with many different sensors, ranging from a simple light detector to a sophisticated pressure sensitive device installed in a floor. Some of the simple sensors are infrared sensors and cameras. Alternatively, motion can be detected with radio frequency sensors. They are often active sensors, so it is not pure motion detection, but rather the detection of presence of an emitter in the area thus leading to indirect motion detection. When motion detection discovers human beings it can be used to aid in access control decisions.

3.2.1 Infrared Sensors

Infrared sensors are sensors that detect infrared radiation, i.e., electromagnetic waves with a wavelength longer than visible light. All objects that generate heat will also generate infrared radiation. Special materials such as germanium and silicon can be used to detect infrared radiation[1].

The human body has a skin temperature around 33°C, which emits infrared radiation at a specific wavelength (9 and 10 micrometers). Passive infrared sensors are typically designed to sense radiation in this spectrum or a little wider. The data from the sensors can then be used to detect motion, as the amount of infrared energy changes rapidly when a human body enters the field it is measuring. These types of sensors have widespread use as burglar alarms and similar.

3.2.2 Radio Wave Frequency Sensors

Radio waves are, like light and infrared radiation, electromagnetic waves. They have a wavelength longer than infrared and microwaves. Radio waves have different frequencies, and a radio receiver can be tuned to listen on a specific frequency. Radio waves are used for many different things, not only common radios, but everything from baby alarms and garage door openers to mobile phones, satellite communications and electronic warfare.

When using radio wave frequency sensors to detect motion they are sensing it indirectly, so to speak. The sensor would be a receiver waiting for a signal to be emitted. If it is known that the signal comes from a person that has moved into the area, the sensor will then have detected motion.

An interesting sensor in this respect is the Radio Frequency Identification (RFID) system. It consists of tags and a reader, which is an active sensor. The reader will send a request of identification to the tag, and the tag will send back data. RFID tags are often used for warehousing, where they replace bar-codes or other systems for keeping inventory. In a motion detection context, they would of course be placed on a person, so that when person enters a room the system would not only notice his presence, but also gain additional data. This could for instance be data to base access control decisions on.

Another interesting radio wave frequency technology is Bluetooth. Bluetooth is a short range data transmission standard. It is developed to feature wireless plug and play connections, for instance between a computer and peripherals such as a PDA or printer. It could be used to detect motion much in the same way as the RFID tags. The data communication between the client and the sensor could be more extensive, but even with motion detection for access control purposes data in a RFID tag might be sufficient. The interesting part is that the emitter would not have to be made for a specific purpose, but could be a general device such as a mobile phone or a PDA.

3.2.3 Digital Images

Motion detection with cameras uses digital images to determine motion, as this what the cameras will output. Cameras are very useful sensors, as the digital images can be used for many different kinds of analysis. Some background about digital images will briefly be described here.

A digital image is a representation of visual information by digital numbers. Often the picture is represented as a two dimensional matrix, and each element is called a pixel (picture element). The value of a pixel represents a measurement that is connected to the position of the pixel. In a black and white picture the pixel values will be measurements of the light intensity. In a colour picture more than one measurement per pixel is needed, usually 3, corresponding to the colour scheme chosen. Pictures have two kinds of resolutions. The spatial resolution is the number of pixels per picture,i.e. how finely grained the picture is. The gray resolution is the number of gray levels in the picture [18].

A monochrome picture of the dimension $N \times M$ can be described as:

$$f = \{f(i,j) \mid 0 \leq i \leq M{-}1, 0 \leq j \leq N{-}1\}$$

Thus, a single pixel in picture f will be referred to by its position as $f(i,j)$.

An important concept in digital images is noise. Noise is the term for imperfections of image sensors, i.e. cameras. They are incorrect measurements and will appear on the image as discrepancies from the real world scenery.

When analyzing images to detect motion there will usually be more pictures to be analyzed than in regular image analysis. Knowledge about the problem, like whether the camera is moving or not, time between images taken, etc., will help to reduce the data analysis. When detecting motion there is no foolproof technique, no general algorithm for all purposes. It depends on the circumstances and goal [30].

Motion detection objectives can be split into three major groups [30]:

1. Motion detection of any motion. This is usually for security purposes, and is done using a single static camera.

2. Moving object and location detection. This is done with a static camera and a moving object, or with a moving camera and a static object. The objective is to not only detect motion, but to detect the movement and location of an object. This can include the detection of the trajectory of the object's motion, and a prediction of the future trajectory. This is a more difficult task than the problem in the first group.

3. The determination of 3D object properties from 2D projections obtained at different time instants of object movement.

Motion analysis deals with consecutive static images, where image analysis is taking place on each individual image. The motion is usually analyzed by looking for corresponding pairs of interest in sequential pictures.

When looking at simple motion detection, one method is to use the difference between snapshots taken at different points in time. This is called differential motion analysis. It is a simple subtraction of two pictures, f_1 and f_2, to obtain the difference image d [30]:

$$d(i,j) = \begin{cases} 0 & \text{if } \mid f_1(i,j) - f_2(i,j) \mid \leq \varepsilon \\ 1 & \text{otherwise} \end{cases}$$

where ε is a small positive number.

An element $d(i,j)$ may have value 1 due to the following reasons [30]:

1. $f_1(i,j)$ is a pixel on a moving object and $f_2(i,j)$ is a pixel on static background (or the other way around).
2. $f_1(i,j)$ is a pixel a moving object and $f_2(i,j)$ is a pixel on another moving object.
3. $f_1(i,j)$ is a pixel on a moving object and $f_2(i,j)$ is a pixel on a different part of the same moving object.
4. noise and other inaccuracies.

The first three reasons will correctly identify that movement have occurred. The last will be a false positive, as no motion has occurred. To eliminate this, one solution is to ignore any region smaller than a certain threshold, although this may prevent slow motion and small objects from being detected. The success of differential motion analysis depends on background to object contrast -if there is almost none, the movement will be much harder to detect [30]. Differential motion detection does not reveal direction, but for pure detection of movement by objects of a reasonable size and speed it is adequate.

3.3 Summary

In this chapter an overview of sensors has been provided, and motion detection with more specific sensors have been discussed. Sensors are in the most general term data producers as they give input to a system. They can be split into two groups, active sensors that probe the environment and passive sensors that observe the environment.

Motion detection can be used to detect the presence of persons in an environment. There are many sensors that can detect motion Here we have discussed infrared sensors,

sensors using radio wave frequencies and cameras. Infrared sensors use the heat of a human body to detect the presence of a person. Radio wave frequencies are used for many purposes, but in a motion sensing capability technologies such as RFID tags and Bluetooth are of interest. They are both examples of active sensors, where the receiver polls an emitter for data, and the fact that an emitter provides data signifies that motion has taken place in the area of the receiver.

Cameras produce images which in digital form can be processed by a computer. Image analysis is a very wide topic, and many different forms of motion detection can be done. The simplest form is differential motion detection, where the pixel-wise difference between two images taken at different points in time is used to determine if motion has occurred.

Chapter 4

Unix Background Information

Unix is a widely used operating system that was originally developed by Ken Thompson of Bell Laboratories in 1969. Many Unix variations have since been developed, such as Linux that was initiated by Linus Torvalds in 1991. Unix has a modular construction that allows it to be easily modified. Two important modular components are the X Window system and file systems. In this chapter, we will review some theory within these areas, which we later in this thesis will assume that the reader is familiar with. The references in this chapter can be used as a starting point for further reading. Most of the references are for resources that are freely available on the Internet.

Before we start our review of X Window System and file system theory, we will point out an important difference between the two technologies, at least from a security perspective. Although they both are part of the operating system, a file system will (usually) run in kernel space whereas the X Window System runs in user space. This separation is enforced because system software must be protected from unauthorized access by applications[33]. This access control is enforced by the CPU, which has different privilege levels corresponding to different roles. Unix uses two such levels: the kernel executes in the highest level and is not restricted by any access control, whereas applications execute in the lowest level where the access is regulated. Whenever an application invokes a system call, the execution is transferred from user space to kernel space. When the kernel executes the system call, it works in the context of the process and has access to the data in the process's address space. The process's context can be used to enforce additional access control based on, for example, the user ID associated with the process.

4.1 The X Window System

The X Window System (or simply X) is a graphical windowing environment for UNIX. It consists of a collection of programs, protocols and routines for organizing and maintaining a graphical user interface. Originally, it came into existence at MIT as part of project Athena in 1984. Its main purpose was to provide a platform independent graphics system that could link together the heterogeneous systems that were deployed at MIT. Version 10 (X10) was the first version of the X Window System that achieved widespread deployment, and it was shortly thereafter replaced by version 11 (X11) in 1987[2]. Since then there have been many further releases, which have added extra functionality while attempting to remain largely backwards compatible. The current release is the sixth one and is known as X11R6. The X.Org Foundation is a consortium that handles the development of the X Window

System technology and standards[3]. XFree86 is an open source implementation of the X Window System, which is included in all modern Linux distributions[4].

The X Window System is a network based windowing system. This means that a network terminal, denoted the *X terminal*, is used to connect a user to a remote computer over a network. The network can either be a local area network or a wide area network. The *X protocol* specifies the message types that can be transferred over the network. The network architecture is illustrated in Figure 4.1.

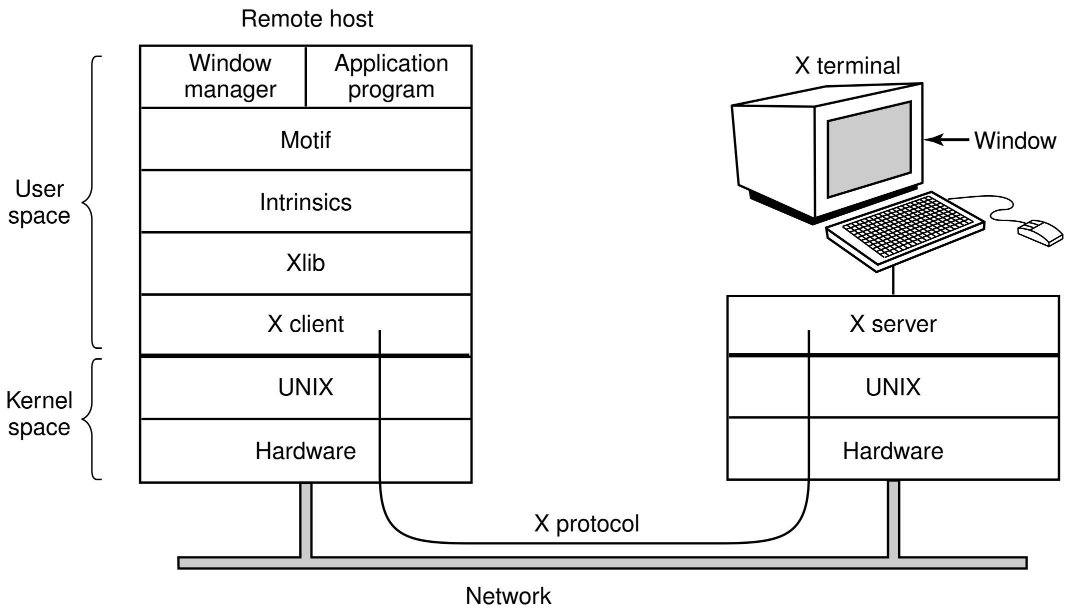


Figure 4.1: Clients and servers in the X Window System[39]. The X Window System consists of the X protocol, X server, X clients, Xlib, and Intrinsics.

The X Window System makes it possible to run programs on remote computers and redirect their output to the display on a local computer. The *X display server* (or *X server* for short) runs on the local computer and listens for network connections on a specified port. The *X clients* are applications that sends commands to the X server. They may run on remote computers, but can also run on the same computer as the X server.

The X server offers graphics display services to the X clients. An X client can, for example, be an editor that sends drawing requests to the X server, specifying that some text should be displayed. It is the responsibility of the X server to display the appropriate bits. Besides rendering graphics on the display, the X server is also in charge of sending input events from the keyboard and mouse to the X clients.

Everything that can be drawn on the screen by the X clients appears in *windows*, and each window is associated with a specific X client. When a user for example presses and releases a mouse button, the X server will send an input event to the X client that created the window containing the cursor.

X clients are usually programmed in C using the library Xlib[5]. Xlib is a low level interface to the X protocol, and it contains, for instance, functions for creating, mapping and unmapping windows. A higher level interface is provided by Intrinsic. It is a toolkit that can be used to build GUI components, such as buttons and scrollbars. To create a GUI interface with a uniform look and feel, an even higher level toolkit is required. The Motif toolkit was widely used in the 1980's and early 1990's. Today the most popular toolkits are Gtk and Qt, which are used in the GNOME and KDE projects, respectively[28].

The X Window System is designed to provide mechanism, not policy. In particular, the X protocol only specifies the basic tasks that the X server performs, whereas the user interface policy is determined by the X clients. One important X client that manages the layout of windows on the screen is the *window manager*. It controls the creation, deletion, and movement of windows on the screen and sends commands to the X server, informing it about what it should display. Furthermore, it will generally provide standard components for the other X clients, such as title bars, menus and frames. Many window managers have been developed (for example Metacity, Enlightenment, and Sawfish) and they provide a range of different appearances and behaviors[6].

A *desktop environment* provides a uniform looking desktop interface which uses the services of a window manager. It offers a more complete interface to the operating system and provides its own range of integrated utilities and applications. Currently, the most popular desktop environments are GNOME and KDE.

X clients should adhere to the Inter-Client Communication Conventions Manual Manual (ICCCM) which specifies a protocol for communication between X clients[32]. Because the X Window System is designed to provide mechanism, not policy, the X clients does not have to adhere to ICCCM. However, they should do so in order to coexists properly with other X clients, especially the window manager. The ICCCM specifies the X client interactions at a low level. The KDE and GNOME projects originally developed their own extensions to the ICCCM in order to support special features in their desktop environment. The Extended Window Manager Hints (EWMH) is an extension to ICCCM that has been developed to replace these custom extensions[14]. It is developed by freedesktop.org which is a is a free software project that currently is not a formal standards organization.

4.2 File Systems

A *file* is an abstraction for a data container that supports sequential and random access. A *file system* is software that permits organizing, manipulating, and accessing files. In a Unix file system, an *inode* is a data structure that holds information about a file, such as the type of file, the number of links to the file, the owner's user and group ids, and the number of bytes in the file. A file is uniquely identified by the file system on which it resides and its inode number on that system. A *directory* is a file that stores (inode number, file name) pairs.

4.2.1 The Virtual File System

Most Unix systems can be modeled using a layered architecture; in Figure 4.2, some layers related to file systems are shown. The top layer is the system call layer. It handles system calls such as `open`, `read`, `write`, and `close`. After a system call has been parsed and the arguments have been checked, it invokes the *Virtual File System* (VFS) layer.

The VFS resides in the kernel and implements the most abstract part of the kernel's file handling infrastructure. It provides a set of standard internal file-system interfaces for file handling functionality, which are independent of the actual implementation of the file. The VFS can therefore provide support for many different types of lower file systems.

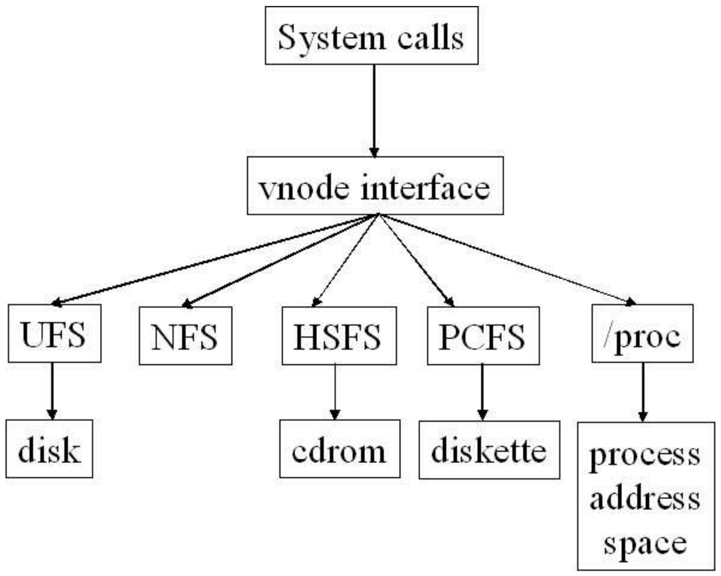


Figure 4.2: The vnode interface in the VFS permits that many different types of file systems can be supported when an application, via a system call, uses a file.

Because of the high abstraction level, the VFS can also work on entities are not true files, but have pathnames, such as character devices, pipes and sockets.

Like the inode is used to store data about a file on a given device, the *vnode* (virtual inode) is a data structure in the VFS layer that is associated with an open file. A system call has a user space and a kernel space part, and when the kernel space part is invoked a vnode operation is called, which in turn calls a function in a lower file system. The lower file system can, for example, be a Unix File System (UFS), Network File System (NFS), High-Sierra File System (HSFS - found on CDROMs), MSDOS File System (PCFS), or the /proc file system (resides only in memory). For further reading on file systems, see for example [16, 29, 39]

4.2.2 Stackable File Systems

The development of a new file system is a difficult, long, and non-portable process that requires skilled programmers who understand kernel internals. A *stackable file system*, however, is easier to develop because it uses existing file systems and interfaces. The interfaces used are vnode interfaces. A stackable file system does neither change the system call interface nor the vnode interface. There are many ways in which vnode interfaces can be stacked; one (simple) possibility is shown in Figure 4.3.

The FiST (File System Translator) system enables programmers to write stackable file system for many different Unix systems [44, 45]. It consists of three parts as shown in Figure 4.4. Firstly, a set of stackable file system templates for several versions of Linux, Solaris, and FreeBSD makes it possible to write file systems for many platforms. Secondly, the FiST input file is written in the high-level FiST language, which can describe stackable file systems in a portable manner. Thirdly, the code generation program `fistgen` can compile a FiST input file into loadable kernel modules for several Unix systems, depending on the templates used. The performance overhead when using FiST generated stackable file system is only 1-2%.

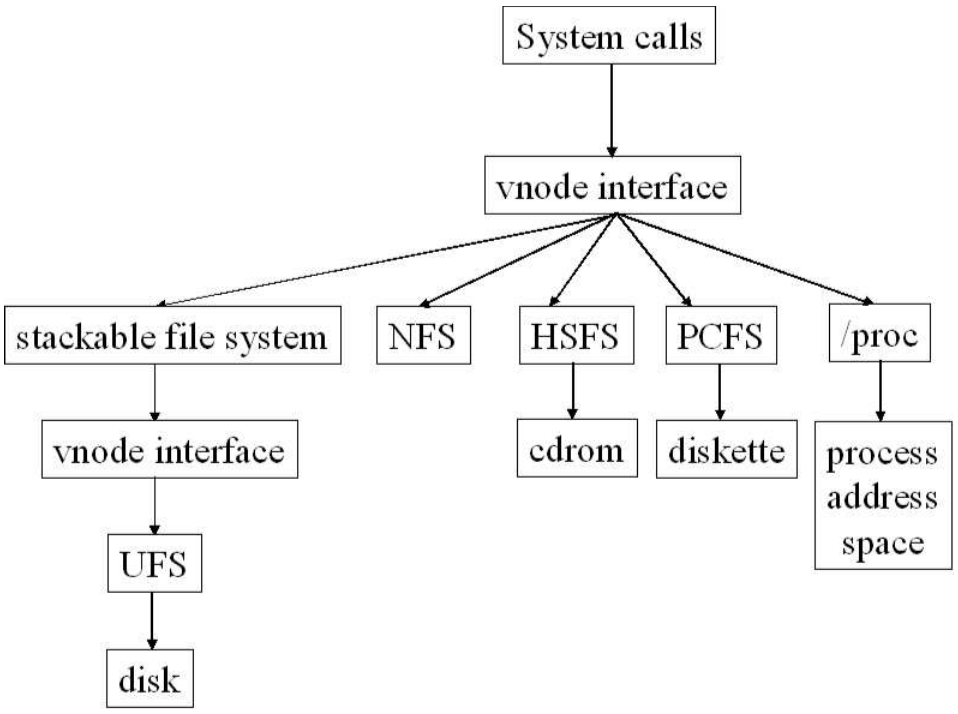


Figure 4.3: A stackable file system is a layer that resides in the kernel below the vnode interface layer. It mediates access to one or more lower file systems.

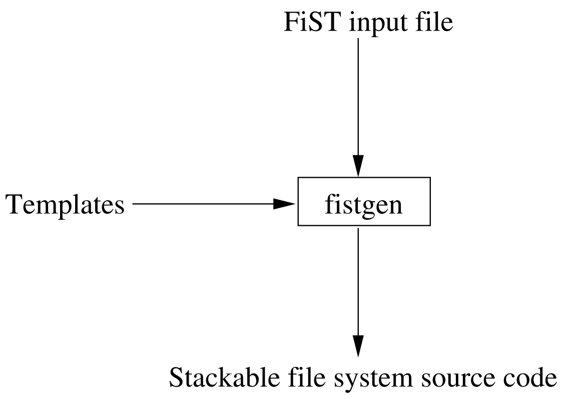


Figure 4.4: Stackable file system development using FiST

The code generated by `fistgen` constitute a kernel module. A *kernel module* is a piece of kernel code which usually implements a file system or device driver. Its main advantage compared to static kernel code is that it can be loaded and unloaded from memory separately from the main body of the kernel. It is therefore not necessary to rebuild and reboot the kernel every time new functionality should be added[34, 33].

Chapter 5

Sensor Enhanced Access Control Model

In organizations with high security requirements, mandatory access control (MAC) is often used to ensure that information does not leak to unauthorized persons. These organizations can, for instance, be the military, hospitals, and corporate patent departments. An authority within the organization states rules about who can see what, and these rules cannot be changed by the individual users as it is possible with discretionary access control (DAC).

The most widely used models are multilevel security models where each subject and object is associated with a security level. However, these models are inadequate when logical resources obtain physical form. In particular, they do not encompass information on a computer screen that is visible to all the persons who are present in the environment. In this chapter, we describe a model that extends traditional multilevel security models with subjects and objects that are physical entities. The model consists of a logical access control model (see Section 5.1) where only users and files are considered to be subjects and objects, respectively. Furthermore, the model consists of a physical access control model (see Section 5.2) where persons in the environment and windows on computer displays are considered to be subjects and objects, respectively. The core of our model is the combination of these two access control models (see Section 5.3).

The system we are set to develop will be integrated with an existing Unix system. The terminology used in our model will reflect this: we will use the terms user ID, process ID, inode number, and window ID to denote a uniquely identified user, process, file, and window, respectively. A consequence of using a Unix system is that the Unix provided DAC (in the form of rwx mode bits) can be used by the individual users to add an extra layer of access control. The MAC will, however, take precedence over the DAC.

5.1 Logical Access Control

Our logical access control model is concerned with mediating access to files by users.

5.1.1 Files

A *file* is a passive entity that stores information on a computer. It is uniquely identified by an *inode number.* There exists many types of files, such as regular files, directories, or symbolic links.

A *file system* is a collection of files along with the operations that can be performed on these files. In a file system where MAC is enforced, the only operations that are subject to access control restrictions are *read* and *write.* A *file level* is associated with each inode number and is used by the reference monitor to determine whether access to the file with the given inode number should be granted or denied.

5.1.2 Users

A *user* is an active entity that can access files. In most deployments of our system, a user account will exists for each user, and a user must identify himself and be authenticated as part of a login procedure. When a file is accessed by a user, it is actually not the user who directly accesses the file, but a process that runs on behalf of the user. The logical subject is therefore a user-started *process*, whereas the user is the physically present person who has logged in to the system. A user must be uniquely identified by a *user ID*, and this user ID must be associated with each process that is started by the user. A *user level* will be associated with each user ID, and it will constitute the basis for deciding what files a given user has access to.

The system has a special user, the *super user* (root in Unix), which is completely trusted. The actions of the super user are not restricted by any access control. Furthermore, only the super user will be permitted access to modifying file levels, assigning user levels to users, and starting and shutting down the system. A special user ID is associated with the super user so that it can be distinguished from the non-privileged users. A level can be associated with the super user, but it will not be used in any access control operations. The person who is able to log in as the super user should use some form of physical access control, such as locking the door, before using the system. Otherwise, an unauthorized person might enter the environment and force the person to misuse the super user privileges, bypassing all logical access control mechanisms.

5.2 Physical Access Control

Our physical access control model is concerned with mediating access to data displayed in computer windows by persons.

5.2.1 Windows

A *window* is an area of screen space that is used to represent a computing function graphically. Windows are created by running a GUI application, such as a text editor or an Internet browser. Each window must be uniquely identified by a *window ID.* Most applications will open one *top-level window* and possibly some sub-windows inside this special window. A sub-window can, for instance, be a button, menu, or scrollbar. Our model will only use the first top-level window created by an application and ignore any further created top-level windows as well as all the sub-windows of top-level windows. This ensures that an application can be uniquely identified by a window ID. When a window is closed, usually when the corresponding application is killed, the system should notice this.

A window will be considered to be a physical entity since it constitutes human-readable output that is physically present on a computer monitor. The access operation for a window is *viewing*. If access to viewing a window is denied, the window will be *unmapped* so that it no longer is visible on a computer monitor. Otherwise, if access to a window is granted and it currently is unmapped, the window will be *mapped*.

The rules that determine whether access to a window should be granted or denied depends on the window level. A *window level* is the maximum file level of all the files that are or have been open by the application associated with the window. The model does not encompass closing of files by applications, so a window level can never decrease. Initially, before any files are opened, the window level will have the lowest possible value, and it will remain unchanged or increase as new files are opened by the application. This implies that if the window level should be decreased because the application no longer has files with high file levels open, the application must be restarted.

Close operations are omitted in our model because it is very difficult to determine when a file no longer is in use by the editor: when an editor has opened a file, it will store its content in memory and close the file immediately after. Furthermore, it would be a vulnerability in the system, if the window level could decrease: when a file no longer is used by an editor, the memory where it had been stored will most likely not be zeroed out. The window level should therefore rather be associated with the address space of the editor process than the window which appears on the screen. The system will be more secure if the window level cannot decrease, and this is more important than the inconvenience experienced by the users.

5.2.2 Persons

Our model will not only deal with users who are directly identified by the computer during some login procedure, but also with any principal who is physically present in the environment. These unauthorized principals will be denoted *persons*. Furthermore, the term *environment* will denote the area around the computer where the persons may be detected.

The physical access control is based on sensors that detect persons in a given environment. To provide a more fine-grained detection method, several sensors can be used and they can even be constructed using different technologies. The only requirement is that they communicate using the same protocol so that they present information about the persons using the same format. A sensor must as a minimum be able to detect whether a person enters or leaves the environment since a list of all the currently present persons must be maintained. The term *direction* will denote the walking direction of the person. A direction can either be 'i' or 'o' which models that the person enter or leaves the environment, respectively.

If a sensor can identify and authenticate persons, a level can be associated with the persons. This could, for example, be done using a smart card and corresponding smart card reader which should be used before physical access is granted. If the sensor is not capable of distinguishing between the detected persons, the same level must be associated with all persons. This level must depend on other security measures taken in the environment, such as physical access control using locks or guards. The term *environment level* will denote the level of a person as it is detected by a sensor.

Whether persons should be recognized as individuals may not only depend on the sophistication of the sensor(s), but also on privacy concerns. In some situations, it may be preferred that some sensors can identify persons, but only when explicit consent has been given. A possible scenario is a company where the employees are recognized with magnetic cards, whereas guests are detected by infrared sensors and thus remain anonymous.

5.3 Combining Logical and Physical Access Control

To summarize our logical and physical access control models, subjects are either processes or persons, and objects are either files or windows. These logical and physical subjects and objects have to be combined to form a coherent access control model. In this model, a *subject level* will denote either a user level or an environment level, and an *object level* will denote either a file level or a window level. The minimum of all the subject levels that are currently registered by the system is denoted the *clearance level.* It models the combined subject level and it will be used in all access control operations by the system. The clearance level will change as persons enter or leave the environment. A summary of the introduced concepts related to physical and logical access control is listed in Table 5.1.

Access Control Concept	**Physical Access Control**	**Logical Access Control**
Subject	Person	Process
Subject level	Environment level	User level
Subject detection method	Sensor	Login procedure
Object	Window	File
Object level	Window level	File level
Access operation	View data	Read or write data

Table 5.1: Sensor enhanced access control model terminology

To ensure that the model is consist, a user is also regarded as an person after he has logged in. If no one else is present during log in and the user subsequently leaves the environment, the sensors will detect this and it will be registered that a person with the level of the user has left the environment. No one is then present, but when the user reenters the environment, the clearance level will again be set to the original value. If it is not the user, but an unauthorized person who enters the environment before the user, this will be detected, and the clearance level will set accordingly.

When the sensors cannot detect the level of persons, the system will not be fully functional if the user leaves the environment. The environment level will most likely be lower than the user level, and when the user leaves the sensors will report that a person has left the environment. If no one is in the environment other than the user, the system will believe that an error has occurred since it had not detected the entrance of the person. The clearance level will therefore be unchanged, but when the user reenters the clearance level will be reduced to the environment level and the user cannot continue to work on his classified files. If persons are present when the user leaves, the system will believe that it is one of them who leaves. When the user later enters the environment, the sensors will report that a person has entered, and even when all real persons have left, the clearance level will still be equal to the environment level.

5.3.1 Reference Monitors

A reference monitor mediates accesses to objects by subjects. Our system uses two reference monitors, corresponding to the logical and physical access control mediation. Firstly, a stackable file system acts as a reference monitor when it mediates access to files by processes. The stackable file system will be denoted `macfs` (mandatory access control file system). Secondly, the `visibility_manager` process will act as a reference monitor when it mediates access to windows by persons.

The `macfs` and `visibility_manager` are not reference monitors in the sense described in the orange book[13] since they do not meet its requirements regarding complete mediation, isolation, and verifiability. In our model, a reference monitor will only denote an entity that mediates access to resources and is protected with the security mechanisms available in Unix systems.

Figure 5.1 illustrates how the `macfs` mediates access to files by processes. The figure also includes a person and a window to illustrate how the physical and logical entities in our model interact. The process displays some output to the user in a window on the screen. It could for example run an editor that reads in a file. If the access operation is write, no output is displayed, except possibly an error message.

The steps when data in the file system is accessed are as follows: Initially, a person starts a process and instructs it to read or write the data (Step 1). In an editor, the person may for instance open a file or save a file, respectively. The process then makes an access request to the reference monitor `macfs` (Step 2). This is done when a `read` or `write` system call is invoked. The `macfs` must then determine whether the access should be granted or denied. This is done using the clearance level and the file level associated with the file. If the access is granted, the file is accessed on the hard disk (Step 3). If the access operation is `read`, data will be read from the hard disk and returned (Step 4). If the access operation is `write`, only some status information is returned. When data is read, the `macfs` will next copy the data into memory in user space where it can be accessed by the process (Step 5). The process is now ready to display some output to the person. The X Server is in charge of displaying graphics, so the process (which is an X Client) will relay the data to this process (Step 6). The X Server uses its knowledge about the computer hardware to display the appropriate bits on the monitor (Step 7). Finally, the output of the access operation is visible to the person (Step 8).

If the `macfs` prohibits the access request, a “permission denied” error message is sent back to the process, which in turn will instruct the X Server to display an error message. In this case, Step 3 and 4 are skipped.

Figure 5.2 illustrates how the `visibility_manager` mediates access to windows by persons. Initially, the deployed sensor will detect that an unauthorized person enters the monitored environment (Step 1). It will notify the `visibility_manager` about this event by sending it the walking direction and environment level of the person (Step 2). If the environment level is less than the current clearance level, the clearance level must be set to the newly detected environment level. This decrease in clearance level implies that access should be denied to some classified data that could be accessed before the person’s entrance in the environment. Since the system has two reference monitors that both uses the clearance level when making access control decisions, the `macfs` must also be notified about the new clearance level (Step 3). Besides ensuring that the logical access control part of the system still mediates access correctly, the `visibility_manager` has the responsibility of mediating access to windows. The `window management` knows which

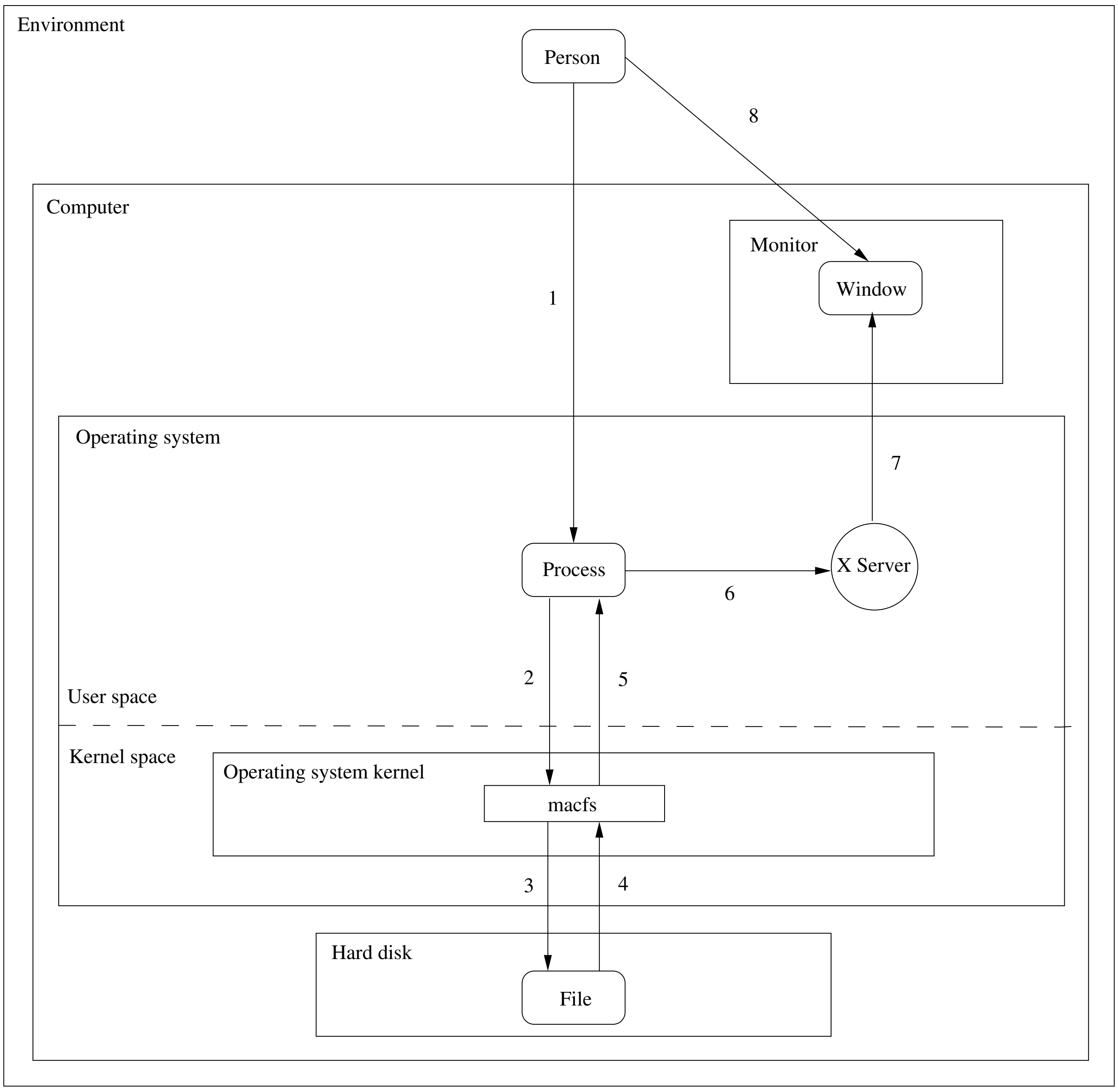


Figure 5.1: The stackable file system `macfs` acts as a reference monitor when it mediates access to a file by a person. The person can either be the authorized user or an unauthorized person.

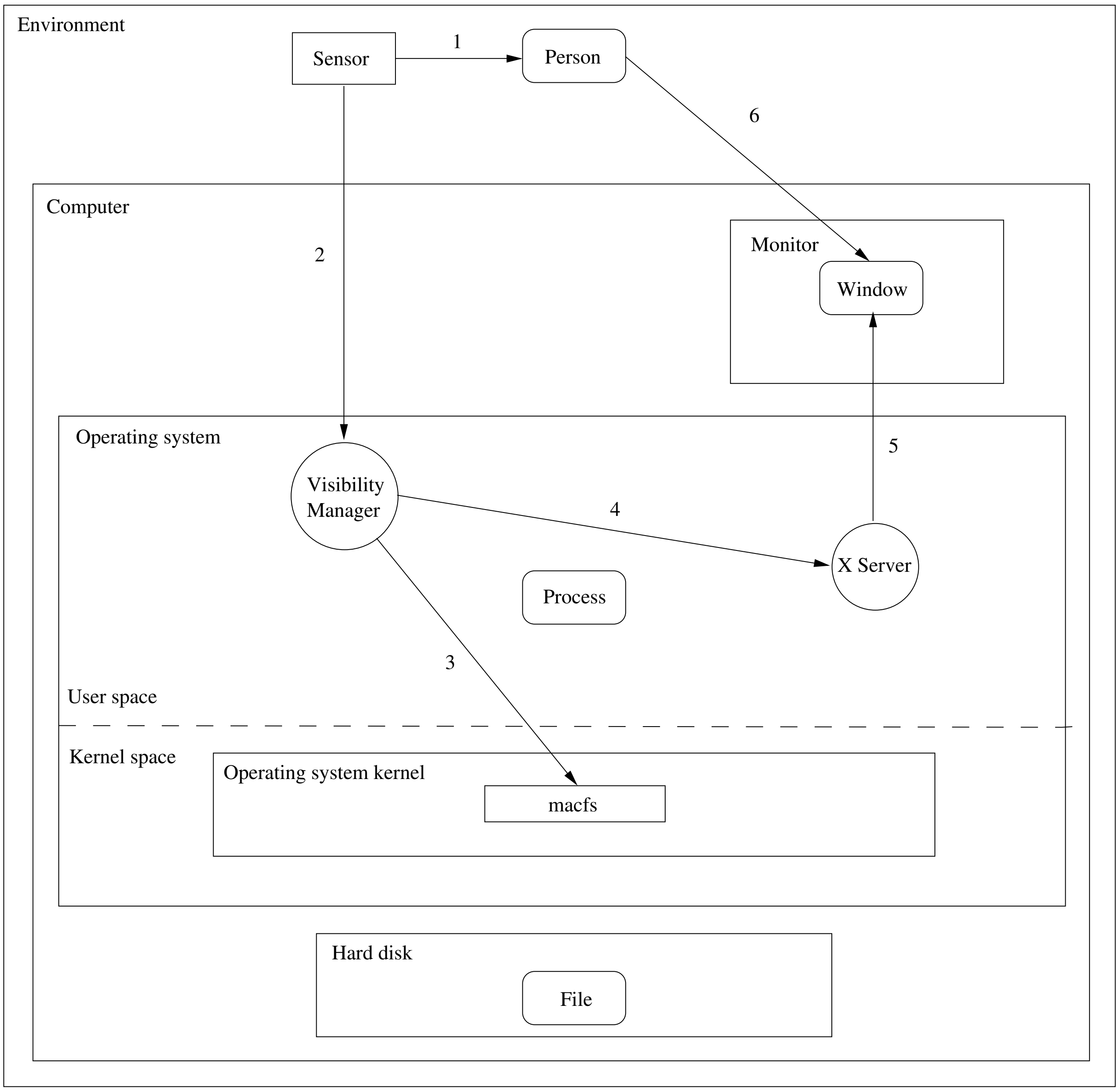


Figure 5.2: The Visibility Manager process acts as a reference monitor when it mediates access to a window by a person.

windows are created by the running processes and the corresponding window levels. It uses the window levels and the clearance level to determine which windows should be made invisible. The **`visibility_manager`** will send its decisions to the X Server so that they can be enforced (Step 4). The X Server will obey the orders and remove all the windows in question from the display (Step 5). The windows (if any) that must not be seen by the newly entered person will therefore be invisible (Step 6).

When a person leaves the environment, the sensor will also detect this and notify the **`visibility_manager`** about it so that the windows can be mapped again. Special cases arise when a person enters the environment and no one was present beforehand. These cases will be described in the next chapter, along with a more detailed description of the **`visibility_manager`** and its interaction with the other parts of the system

5.3.2 Security Policy

Adhering to the general Unix philosophy, the system provides mechanism and not policy to the extend possible. Therefore, as much as possible can be specified using parameters, and together they will specify a security policy. The parameters should be set by a trusted authority. Since there must also be a trusted super user, these two roles can be managed by the same person.

There are six parameters that can be set when defining the security policy. The most important parameters are the *no_ read_ up*, *no_ read_ down*, *no_ write_ up*, *no_ read_ down* parameters. These four parameters specify the rules that are used by access operations to determine whether access to a file or window should be granted or denied. They can, for example, be used to specify a security policy that enforces the Bell-LaPadula model or the Biba model and thus addresses confidentiality or integrity, respectively. The fifth parameter, *hide_ non-readable_ files*, should be set if the files that are non-readable, according to the *no_ read_ up* and *no_ read_ down* parameters, should be hidden by the stackable file system. The purpose of this parameter is to avoid information flow via file names. The final parameter, *permit_ lower_ level_ login*, can be set if the users should be permitted to log in at a level below their user level.

All types of levels in our model will be modeled as non-negative integers, so we do not impose an upper limit for a level and the lowest possible level is zero. This decision has mainly been made in order to provide a model that is as policy free as possible. An organization will probably have an upper limit on the number of levels. For instance, the military might have four levels, corresponding to unclassified, confidential, secret and top secret, but another organization might need more or fewer levels. Therefore, the only constraint that we enforce on the levels is that they cannot be negative; the upper limit will be machine dependent. The super user in an organization must ensure that the levels does not exceed the limit set by the organization.

Although we have attempted to make a system that is as policy free as possible, the design does contain some policy decisions as described in the next chapter. These decisions are especially related to what information users are permitted to retrieve about the system state. When in doubt about whether an access restriction should be imposed on the usage of a program, we omitted the restriction. This was the case for the programs that are used to retrieve subject levels, window levels and the clearance level. We have therefore disobeyed the least privilege principle. The main reason for this is that the Unix DAC can be used by the super user to impose access restrictions on the usage of certain programs.

This points out that our model is not designed for a stand-alone system, but for a system that is tightly integrated with a Unix operating system which also provides (policy free) access control mechanisms on its own.

Chapter 6

Design

The system that we are set to develop must encompass many different types of functionality, ranging from storing file and user levels to detecting when persons enters or leaves a given environment. In order to make the system more comprehensible, it has been divided into a number of subsystems that provide services to each other. These subsystems will be described in this chapter along with how the different parts interact with each other. In Section 6.1, an overview of the system architecture is provided. Section 6.2 describes the security parameters that can be used to specify which security policy should be enforced in a given deployment of the system. Each of the subsystems that the system has been divided into are then described in Section 6.3 to Section 3.1. One of the developed subsystems, the `security management gui`, will not be described until Chapter 7 since it is only an insignificant part of the system as a whole. The system will be denoted SEAC (Sensor Enhanced Access Control).

Each subsystem contains between one and six programs. The services provided by each of these programs will be described in the section corresponding to the subsystem where the program resides. Many of the programs send messages to each other when they run. Some of these processes only provide services to other processes and therefore act as servers, some of them only use services provided by others and therefore act as clients, and finally some act both as clients and servers. To provide an overview of all these programs and their interaction, a CSP specification is listed in Section 6.8. It describes the developed protocol at a high abstraction level.

6.1 Software Architecture Overview

6.1.1 The Subsystems

The system has been divided into a number of subsystems, where each subsystem focuses on a different aspect of the functionality of the system as a whole, see Figure 6.1. These subsystems can be organized in a layered architecture where each layer represents different levels of abstraction. Some of the layers have been partitioned further into subsystems because the subsystems have different focus of functionality.

The lowest layer is a native file system where all the classified files will be stored. Our design is not dependent on the type of used file system so this layer will not be described further. For each file in the system that should be protected by access control, the

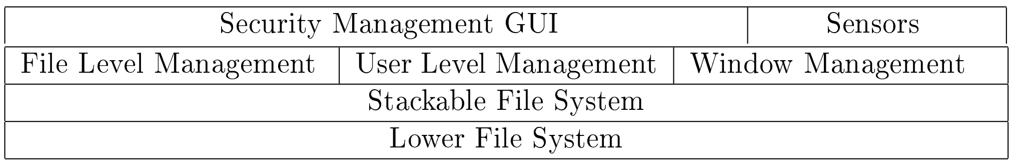


Figure 6.1: Layered architecture for the system.

`stackable file system` subsystem stores a file level corresponding to the classification of the file. Furthermore, the `stackable file system` stores a user level corresponding to the clearance level of each user in the system. The `file level management` and `user level management` subsystems provide programs for storing and retrieving file and user levels, respectively. The `visibility_manager` subsystem manages the Window visibility; its main purpose is to provide a service to the `sensor` subsystem so that windows are (in)visible dependent on who are present in the environment. Finally, a `security management gui` subsystem is included in order to make the system more user-friendly. It does not add any new functionality to the system and is primarily developed because many users prefer a GUI over command line tools.

The layered architecture is closed in the sense that each layer only uses the services provided by the layer(s) immediately below it. For instance, the `file level management` uses the services provided by the `stackable file system`, and the `security management gui` uses the services provided by the `file level management`, `user level management`, and `window management`.

The two lower layers reside in the kernel of the operating system since non-privileged users must not have access to these parts. The two upper layers reside in user space and use the system call interface of the operating system to interact with the stackable file system.

Altogether, the `stackable file system`, `file level management`, and `user level management` provides a MAC that is dependent on the security policy chosen when the system is initialized. These three subsystems can be used independently of the other subsystems, which may for example be useful in a setting where only logical and not physical access control is required. The GUI part of the system will still be available in this setting, although with restricted functionality. The `window management` and `sensor` subsystems are not usable on their own.

6.1.2 Processes and Message Passing

In the user space part of the system, a number of processes must run all the time as daemon processes so that they can provide services to other processes that are only invoked occasionally by the user of the system. Many of the processes communicate using message passing as indicated in Figure 6.2.

The system has a trusted super user who can use all the programs without any restrictions imposed by the access control mechanism. This user is named `root` in Figure 6.2. All the other users in the system are non-privileged, and they are denoted `user` in the figure. The root user can of course also act as a non-privileged user.

The `editor` and `window manager` processes shown in the figure have not been developed as part of this project and they are therefore not included in one of the subsystems. The

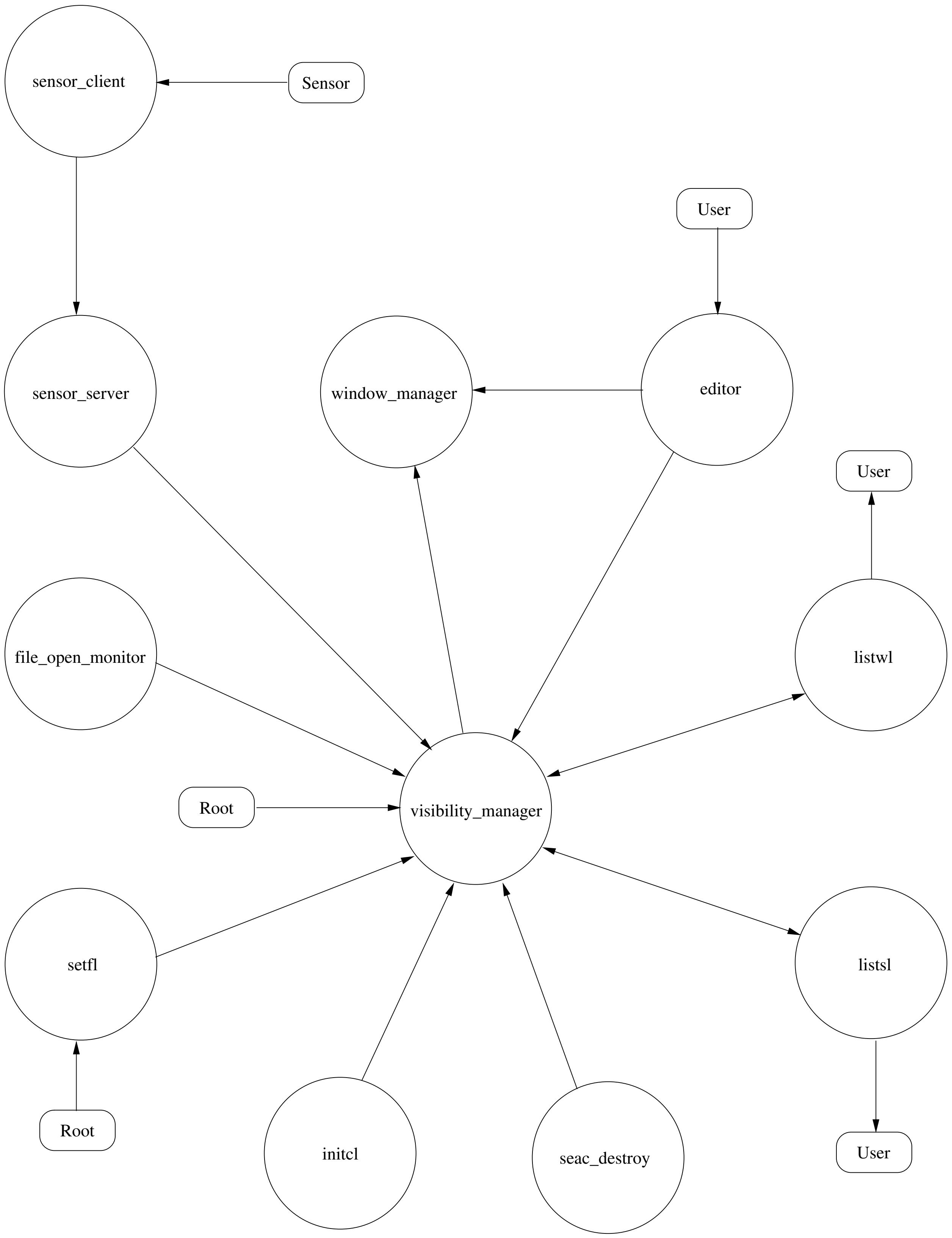


Figure 6.2: Message passing between processes in user space. The ovals illustrate processes, the arrows illustrate the message flow, and the boxes denote an entity that interacts with the system.

assumptions that we have made about the behavior of these processes will be described briefly in the reminder of this subsection. The remaining processes will be described in the following sections along with the subsystems where they reside. The interactions between the processes are described in the final section, and the description of parts of their functionality will therefore be postponed to this section. A reference guide for the programs are provided in C.

The term *editor* will be used to denote any application that can be used to view and/or modify a file. It can, for example, be a text editor, a web browser or the Unix output redirection operator '>' which is used to write to a file. In most cases, a user who is editing a file will use a text editor that both reads and writes the file contents. Only one such editor has been shown in Figure 6.2, and it is denoted `editor`. After having experimented with the `emacs` and `gedit` editors that were included in the Fedora Core 1 Linux distribution, we have decided to make a few assumptions about how the editors to be used in the system works in general. First of all, when a file is opened in an editor it will be closed immediately afterwards and then stored in a buffer for internal usage by the editor. Because the file is stored in a buffer, it will be very hard to discover when the user "closes" the file by killing its buffer. We have therefore chosen to simply refrain from considering when a file is closed in the editor. This implies that the user must restart the editor if the fact that a given file has been opened should no longer be stored by the system. Furthermore, this approach is also preferable from a security perspective, as mentioned in Section 5.2.1.

The `window manager` process shown in Figure 6.2 must maintain the state of each created window. This should be done so that it is consistent with what is actually displayed on the computer display by the X server.

6.2 The Security Policy Parameters

Six parameters are available for specifying a security policy. Two of the parameters restrict read access for both files and windows. We will consider a window to be readable if it is visible on the computer monitor, and the window will be unmapped if read access to the window is denied. The security parameters are as follows:

no_read_up Read access to a file is denied, if the file level is greater than the clearance level. Likewise, view access to a window is denied, if the window level is greater than the clearance level. This parameter enforces the Bell-LaPadula simple security property.

no_read_down Read access to a file is denied, if the file level is less than the clearance level. Likewise, view access to a window is denied, if the window level is less than the clearance level. This parameter enforces the Biba integrity * property.

no_write_down Write access to a file is denied, if the file level is less than the clearance level. This parameter enforces the Bell-LaPadula * property.

no_write_up Write access to a file is denied, if the file level is greater than the clearance level. This parameter enforces the Biba simple integrity principle.

hide_non-readable_files The file names of non-readable files will be hidden. A file is considered to be non-readable if read access is denied according to the *no_read_up* or *no_ read_ down* parameters. The file names of non-readable files will be omitted when entries in a directory are read, so they will for instance be omitted in a directory listing.

The purpose of this parameter is to prevent information flow via file names, since this constitutes a covert channel. For example, if the *no_ read_ up* policy is chosen, a low level subject should be denied access to both the content of a high level file and the name of the file.

permit_ lower_ level_ login The clearance level will normally be initialized to the level of the user who logged in. In some situations, however, it may be desirable if the clearance level is initialized to a level below the user's user level. For example, if the Bell-LaPadula model is enforced a high-level user will be denied access to send a message to a low-level user via a file. If it should be possible for users to escape from this restriction, the *permit_ lower_ level_ login* policy should be chosen. A user is then allowed to specify the level that the clearance level should be initialized to, as long as it is not greater than the user's user level.

If only the *no_ read_ up* and *no_ write_ down* policies are chosen, the security policy will enforce the Bell-LaPadula model. If only the *no_ read_ down* and *no_ write_ up* policies are chosen, the security policy will enforce the Biba model. For both models, the *hide_ non-readable_files* policy should also be chosen to avoid information flow via file names.

6.3 The Stackable File System

The users of the system must not have direct access to the files containing classified information. Instead, some mechanism must be established that can mediate all accesses to files by users. One candidate for providing this type of functionality is a stackable file system that stores a level corresponding to each file in the underlying native file system. In order to provide MAC, the stackable file system must also store a level corresponding to each user who has access to the system. Using these levels, the stackable file system can mediate all accesses to files by users. If sensors are deployed in the environment, the levels of persons will also be used in access control decisions. The main reason for including the MAC in a stackable file system is that it will then reside in kernel space and therefore be protected from non-privileged users via the operating system user/kernel modes.

Two programs are provided for initializing the system, and one for shutting it down:

`seac_init` initializes the stackable file system. First of all, the security policy should be specified, i.e., the *no_ read_ up*, *no_ read_ down*, *no_ write_ up*, *no_ write_ down*, and *hide_ non-readable_files* parameters should be set. Secondly, the system parameters that specify where the user levels and file levels should be stored persistently are set. If the system has been used previously, it retrieves the previously stored user and file levels. Finally, the part of the stackable file system that must discover when a file is opened is initialized; this part is used by the `file_open_monitor` program. Only the super user is permitted to use this program.

`initcl` The user level of the user invoking this program will be used to initialize the clearance level. Before the `initcl` is invoked, the clearance level will have the lowest possible level. The user will therefore not have access to any files in the stackable file system until the level is increased. The program will thus function as a login program. It will not involve any identification and authentication since we assume that this has already been done by the operating system, since the user ID of the user will otherwise not be associated with the process that executes the `initcl` program.

If the *permit_ lower_ level_ login* policy option is chosen, the user can specify a level below his user level which the clearance level should be initialized to. If the specified level is above the user's user level, the clearance level will be initialized to the user level. A logout operation will not be supported, so the system must be restarted if the user subsequently wants to log in with another level.

The clearance level can only be initialized once, so after the `initcl` program has been invoked, any further invocations will be ignored by the stackable file system. If the `visibility_manager` is running, it will be notified about the clearance level and use it to determine whether window should be visible or not.

`seac_destroy` destroys the stackable file system and terminates the `visibility_manager` and `file_open_monitor` process. Furthermore, it ensures that all file levels and user levels are stored persistently. Only the super user is permitted to use this program.

6.4 File Level Management

A file level is associated with each inode number in the stackable file system. The `file level management` subsystem provide three programs that can be used to set and retrieve these file levels:

`setfl` sets the level for a file. The level can be any positive integer. If the `visibility_manager` is running and a user has the file open in an editor, the `visibility_manager` will be notified about the change of level and if the new level implies that the visibility of the window should change it will map or unmap the window accordingly. Only the super user is allowed to change the file level.

`getfl` retrieves the file level for a given file. If the *hide_ non-readable_files* security policy has been chosen, it will not be possible to retrieve this level if the file is non-readable according to the *no_ read_ up* or *no_ read_ down* security policies.

`listfl` lists the file name and corresponding file level for every file in a given directory. If no directory is specified, the file level for the current working directory will be printed. If the *hide_ non-readable_files* security policy has been chosen, all the names of non-readable files will be skipped in the listing. Whether a file is readable depends on the current clearance level and the *no_ read_ up* or *no_ read_ down* parameters. The stackable file system can therefore also provide filtering by hiding file names; the purpose is that information flow via file names must be prevented.

6.5 User Level Management

The `user level management` subsystem provides three programs that can be used to set and retrieve user levels. A user level is associated with each user ID in the stackable file system. The default user level will be the lowest possible level, which is zero.

`setul` sets the level for a user. The level can be any positive integer and it will be associated with the user ID of the user. If no user level is explicitly associated with a user ID, the default value zero will be used. Only the super user is allowed to change the user level.

`getul` retrieves the user level for a given user. A user can retrieve his own level, but not the level of another user. The super user can retrieve the level of any user.

`listul` lists the user name and corresponding user level for every user who has a valid login account in the system. An account is considered to be valid if it belongs to a normal user and not a system process such as an FTP demon.

Only the super user is allowed to list the user levels. This is because of the principle of least privilege. A malicious user who has a low level could abuse this information to determine which account he should try to break into in order to obtain a higher security level.

6.6 Window Management

The main responsibility of the `window management` subsystem is to ensure that a person is prohibited from viewing sensitive information in windows on the screen. This task is accomplished by means of a sensor client along with the following three programs:

`visibility_manager` The `visibility_manager` acts as a server for all the programs in the `window management` subsystem, except `getcl`. Its main task is to manage the visibility of windows and ensure that they are mapped or unmapped according to the window levels and the current clearance level.

It is also the responsibility of the `visibility_manager` to ensure that the clearance level in the stackable file system is updated. The clearance level should be set to the environment level when a person enters the environment and the detected environment level is less than the current clearance level. The clearance level should also be updated when an subject who currently has the lowest subject level leaves the environment. If all subjects leaves the environment, including the user, the clearance level should be set to the lowest possible clearance level.

`file_open_monitor` Whenever a file is opened in the stackable file system, the `file_open_monitor` will notify the `visibility_manager` about this by sending a message to the `visibility_manager`. This message will contain information about the process ID of the editor that opened the file, the inode number, file level, and file name. The `visibility_manager` can use the file level to determine the window level since it equals the maximum file level among all the open files in the editor.

`file_open_monitor` will only report/discover when regular files or links are opened, not directories.

`sensor_server` The `sensor_server` receives information about a person form a `sensor_client` and relays it to the `visibility_manager`. The information includes the level of the person and the direction. The direction can be either ”in” or ”out”, depending on whether the person is entering or leaving the environment.

These programs must either be started by the super user or be started automatically as part of the operating system startup. This will prohibit that a non-privileged user simply terminates a process by sending it a kill signal since a user will not have access to do so in a Unix system.

The system also provides a few programs that can be used by any user to retrieve status information about the system:

`listwl` list the window levels for all the current windows along with other status information.

The file names of secret files will be replaced by ”unavailable”.

`listsl` list the subject levels for every subject in the environment. If no person is in the environment, the list will only contain the level of the user who has currently logged in.

`getcl` retrieves the current clearance level from the stackable file system. This level equals the minimum level in the list returned by `listsl`.

6.7 Motion Detection using Sensors

The sensor subsystem should provide information about persons on the physical premises to the logical part of the system. To achieve this we need to collect data with some sensors and use some software to interpret the data. When designing the sensor subsystem it should be possible to use different kinds of sensors and also several at once. Thus the sensor subsystem will be disjoint form the rest of system and have a simple interface to it.

The important thing in this sensor subsystem is that we can show the integration of the physical world with the logical access control. Since it is only a prototype it is not important that complete coverage of the physical environment is achieved or that every sensor type is tested. We want to detect when someone enters the office through a designated way. We do not want to attempt to gather more information about the person, but the system should be able to handle possible additional information, such as a different level based on some form of recognition.

The sensors will provide simple motion detection. We will be able to tell if motion is present or absent, and whether the motion was caused by persons. Furthermore, we need to determine in what direction the person is moving, if he is entering or leaving the physical environment. We want to find an easy way to achieve the desired information about the motion without the use of complicated motion analysis. This means that we are not interested in tracking the motion, recognizing people or similarly complicated tasks. The system will use a simple analysis to reduce the complexity of calculations in the system, and furthermore, the analysis is not the main focus of the project as this is not our area of expertise. It should, however, be possible to use more complicated analysis or sensors if so desired.

6.7.1 Choice of Sensors

We chose to use web-cameras as example sensors. They have the advantage that they are cheap and widely available. Web-cameras also have an interface to computer systems that is straight forward, drivers already exists and they connect to the system via USB cables.

The pictures taken by web-cameras can be used to preform simple motion detection analysis on, such as differential motion analysis. This does not reveal anything about the direction of the movement, but we do need to know which way people move, if they entering or exiting the office. To solve this problem while still using differential analysis, we can use two cameras, and note the time difference between the motion in front of them. Additional issues we need to know to obtain enough information about the motion will be done by indirect analysis, e.g. using the number of pictures taken between two points in time. Using two cameras and differential motion detection will result in a cheap and simple setup, where the analysis is straight forward.

A web-camera is a simple passive sensor that only gives information about the presence or absence of motion. Other passive sensors that could have been used includes infrared sensors. They are also very cheap, and would simply detect someone passing through a doorway. The problem with infrared sensors would be the interface to the computer, a driver and special cables might be needed. More advanced passive sensors would be noise sensors or touch sensitive floor. These are more expensive, not to mention more advanced to install or interpret data from.

The system could also use active sensors such as RFID tags. Employees could wear RFID tags or similar that are read when entering a room, and on the basis of this information determine more accurately what information can be accessed. This is much more elaborate and expensive than web-cameras, but the system should be prepared to handle this sort of information from the physical world.

6.7.2 Motion Detection Programs

The sensor subsystem is designed to communicate with the rest of the system through a client-server architecture. This should be replaceable if there was a need for using e.g. an event bus architecture. A sensor acts as a client that sends information about the environment to the `sensor_server`. A sensor can be either a virtual entity or an actual sensor system.

A sensor client should provide the following functionality:

sensor client whenever a person enters or leaves the office/environment being monitored, an event should be sent to the `sensor_server`. The event should include the subject level and the direction of the person. The direction must either indicate that the person enters the room or leaves the room. We have provided the `swsensor` program with this functionality.

As mentioned above, we have chosen to implement a sensor client that uses web-cameras as sensors. To accomplish the motion detection we want, we need some software system. It consists of two parts, one part that detects motion based on raw input from the camera by differential motion detection. The second part analyzes the results of this to determine if the motion was caused by a person and the direction of the person. For the first part, the differential motion analysis, we use an existing piece of software called Motion[7].

The second part consists of a number of programs providing the following functionality:

`event1/event2` are auxiliary programs that are needed due to the way Motion is functioning.

`motion_handler` does further analysis to determine whether or not the motion detected was a person passing the camera. One instance is needed for each camera.

`camera_client` will collect the data from the two motion handlers and determine if the person moved in or out, and pass this information on the `sensor_server`. The communication variables as well as the level for a default person is passed to the program on start.

6.7.3 Design of Physical Premises

When designing the sensor subsystem an important part of it is the physical premises. We try to gather information about the physical world, and for this information to be as accurate as possible we need to lay out the physical environment carefully.

The cameras need to placed so they will detect any persons entering the room. They should be placed so there will be a distinct time difference between the passing in front of one camera to passing in front of another, so we can determine the direction of movement. It is also important to consider what the surface the camera is placed on. It must be placed so a random push or jump on the floor does not jar the camera too much, as this will of course be considered as motion, since it would generate a large pixel wise difference. A possible way to prevent this is to screw the cameras into the ceiling or wall.

The computer placement is also important. Firstly, it should be placed so that you cannot look at the screen without passing in front of the cameras, for example by standing in a doorway. It has to be impossible to look at the computer screen without being detected by the sensors. Secondly, the computer should be far enough from the sensors to allow the software to process the information, so that sensitive information will already be removed from the screen when a person arrives in front of it.

Furthermore the general design of the room and building need to be considered. It should not be possible to circumvent the system by, for example, crawling through a window or just looking through it. Our system has not directly taken such things into account, but it can be expanded by adding sensors that detect persons in these locations.

6.8 CSP Specification

In this section, the interactions between the user level processes will be described at a high level using a formal specification. The protocols will be described using the notation in [37]. This implies that Communicating Sequential Processes (CSP) will be used to describe the message passing between the communicating processes, and the Vienna Development Method Specification Language (VDM-SL) will be used to describe the data types.

CSP can be used to model a system which encompasses multiple concurrent activities using as a sequence of sequential processes. Each process can be constructed from simpler processes which describes a subset of the entire process's behavior. Furthermore, processes can synchronize their activity by sending and receiving messages via channels. A *channel* provides a one-way path from a sending process to a receiving process. When CSP is used to model message passing between processes, the messages transmitted are denoted *communication events*. A channel is in this case a FIFO queue of pending communication events. To initiate a communication, a process can output an expression e to a channel c using the output expression $c!e \rightarrow P$, which behaves like P when another process has acquired e by receiving from c. The other process can assign e to a variable x of type M by using the input expression $c?x : M \to P$. e and x must have the same type M, and the process expressions are then said to *match*. Execution of matching input and output expressions can be viewed as a distributed assignment that transfers a value from one process to a variable in another.

VDM is a collection of techniques for the formal specification and development of computing systems. One of its components is the model-oriented specification language VDM-SL. A specification in VDM-SL consists of a mathematical model built from simple data types like sets, lists and mappings, along with operations which change the state of the model.

The syntax that we will use differs from the one used in [37] in two respects. Firstly, a let expressions has been included for introducing new variables since this makes it easier to read the specification. Secondly, the map construct from VDM-SL has been included so that the map type $k \xrightarrow{m} v$ defines a mapping from a key of type k to an information value of type v. The subset of CSP and VDM-SL that we have used is described briefly in Appendix A.

The focus in our specification will be the message passing part of the system and not the details about the functionality of individual processes, since this was described informally in the previous sections. Furthermore, the specification will only describe the important parts of the system. We will for instance use CSP to specify how windows are mapped and unmapped, but CSP will not be used to specify how window status information is presented to the user and how backup files are handled. The description of these details is deferred to the next chapter.

6.8.1 Processes, Channels, and Users

All the user space processes that interact using message passing are shown in Figure 6.3. Only one instance of each type of process is shown, although multiple instances of some of the processes may run simultaneously without affecting the system behavior. These processes are `editor`, `sensor_client`, `setfl`, `listwl`, `listsl`. The `visibility_manager`, `file_open_monitor`, and `sensor_server` must run as daemon processes, and `initcl` and `seac_destroy` are used to initialize and shutdown the system. Only one instance of these processes must run at a time, and any subsequent activations should be rejected since the system behavior otherwise will be unpredictable. How this is handled is an implementation issue and will therefore be described in the next chapter.

The channels that connect processes are named left or right. The `visibility_manager` acts as a server that communicates directly with all of the other processes, except the `sensor_client`. An index has therefore also been included in the channel names for these channels so that messages that pass through *left*$[i]$ reach *right*$[i]$ and vice-versa.

Two types of users are shown in Figure 6.3: `root` denotes the privileged super user, and `user` denotes any non-privileged user or the super user. A `user` or `root` interacts with the system via a Service Access Point (SAP) which may, for instance, be bound to standard input or output in a terminal. Each SAP will be modelled as a channel in CSP. A channel that connect a `user` with a process is named SAPx, where x is an integer. The two channels that can only be used by `root` are denoted SAPR1 and SAPR2.

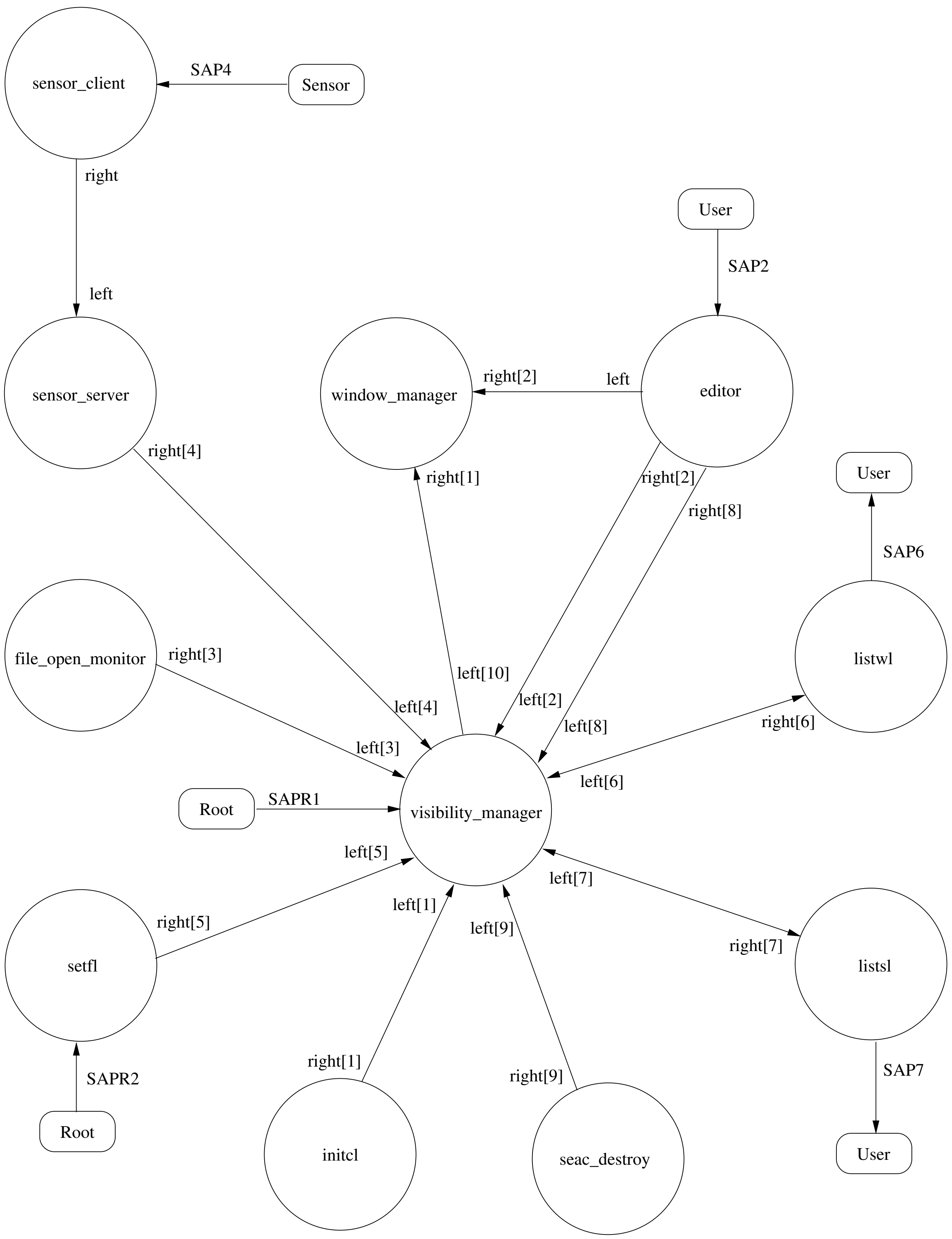


Figure 6.3: CSP processes and channels in the system.

6.8.2 Data Types

The specification uses a few new types, which are defined as follows:

$$
\begin{aligned}
&\mathsf{subject\_level} = \mathbb{N}_0;\\
&\mathsf{object\_level} = \mathbb{N}_0;\\
&\mathsf{PID} = \mathbb{N}_0;\\
&\mathsf{WID} = \mathbb{N}_0;\\
&\mathsf{inode} = \mathbb{N}_0;\\
&\mathsf{string} = \mathsf{char}^*\\
&\mathit{FileInfo} = \mathsf{string} \times \mathsf{object\_level} \times \mathsf{inode};\\
&\mathit{WindowTable} = \mathsf{PID} \xrightarrow{m} (\mathsf{WID} \times \mathsf{string} \times \mathit{FileInfo}^* \times \mathbb{B});\\
&\mathit{Error} = \{\mathit{INVALID\_MESSAGE}\};
\end{aligned}
$$

A level is is associated with each subject (i.e. a user, the super user or a person) and each object (i.e. a file or a window). An upper limit is not impose on these levels, so they are just modeled as non-negative integers.

A process, a window, or an inode number is uniquely identified by a non-negative integer; the PID, WID, and inode types represent process IDs, window IDs, and inode numbers, respectively.

Each value of type *FileInfo* represents a file which is open in an editor, and the elements in the tuple denote a file name, a file level, and an inode number, respectively. Only an inode, and not a file name, can be used to uniquely identfy a file. The file name is only used when status information is presented to a user, whereas the inode will be used to determine whether two files are identical.

A value of type *WindowTable* contains mappings from process IDs of editors to tuples. Each entry in the table represents an editor that has created a window and possibly opened one or more files. The elements in the tuple denotes a window ID, an application name, a sequence containing elements of type *FileInfo*, and a boolean value that is true if and only if the window is mapped. The *FileInfo* sequence is sorted in decreasing order, according to the file level in the *FileInfo* tuple. The window level for the editor is the maximum file level of all the files opened by the editor. Since the sequence is sorted, the window level will be equal to the file level in the first *FileInfo* element of the sequence.

To keep the specification simple and maintain the focus on the core functionality of the system, only three types of errors have been included. These are used to model when a the system is used before a user has logged in, a sensor client sends an invalid direction to the sensor server, and a file level could not be set, respectively. Other erroneous situations may of course arise, but the handling of these is postponed to the implementation phase of the system development.

6.8.3 Functions

Since the main purpose of the specification is to describe the message passing between processes, some details regarding the functionality of individual processes will not be described formally. Furthermore, for some parts of the functionality it will not even be possible to express the desired behavoiur using the available notation. In these cases, a function will called in the specification and the behaviour of the function will only be described informally. The type of the return value, function name, and argument(s) of these functions are as follows:

string $get\_application\_name(window : WID)$ returns the name of the application that has created the *window*.

$\mathbb{B}$ $contains\_inode(file\_list : FileInfo^*, inode : \mathbb{N}_1)$ returns true if and only if the *file_list* contains the *inode*.

$FileInfo^*$ $insert\_sorted\_decr(file\_list : FileInfo^*, file\_info : FileInfo)$ inserts the *file_info* element into the *file_list* sequence so that the resulting sequence is sorted in decreasing order. The new sequence is returned.

subject_level* $insert\_sorted\_incr(subject\_list : \text{subject\_level}^*, env\_level : \text{subject\_level})$ inserts the *env_level* element into the *subject_list* sequence so that the resulting sequence is sorted in increasing order. The new sequence is returned.

subject_level* $remove\_env\_level(subject\_list : \text{subject\_level}^*, env\_level : \text{subject\_level})$ removes *env_level* from *subject_list* and returns the resulting sequence.

$WindowTable$ $remove\_window(table : WindowTable, window : \text{WID})$ removes the mapping from *table* where the information value contains *window*. The resulting table is returned.

S^* $set\_to\_seq(s : S\text{-set})$ returns a sequence containing the elements from the set s in an arbitrary order.

$WindowTable$ $update\_file\_levels(table : WindowTable, inode : \mathbb{N}_1, level : \text{object\_level})$ updates the table entries in *table*. Firstly, all the *FileInfo* tuples which contains *inode* are updated so that the level in the tuple is replaced by *level*. Secondly, the *FileInfo* sequences are sorted in decreasing order.

subject_level $get\_clearance\_level()$ returns the current clearance level.

$\mathbb{B}$ $set\_file\_level(inode : \mathbb{N}_1, level : \text{object\_level})$ sets the file level for *inode* to *level*.

string $table\_to\_string(table : WindowTable)$ returns a string representation of the table. It uses the *hide_non-readable_files* parameter to determine whether file names of non-readable files should be hidden.

string $list\_to\_string(subject\_list : \text{subject\_level}^*)$ returns a string representation of the sequence *subject_list*.

WID $x\_create\_window()$ creates a new unmapped window and returns its window ID.

PID $getpid()$ returns the process ID of the process that called this functions.

$(\text{PID}, \text{object\_level}, \mathbb{N}_1, \text{string})$ $block\_until\_file\_opened()$ blocks until a file is opened and then returns the process ID of the program that opened the file, the file level associated with the file, the inode number of the file, and the file name, respectively.

6.8.4 The Communication Protocol

The system consists of many processes that must interact according to a specified protocol so that the desired system behaviour is obtained. The most essiential behaviour arises when a `sensor_client` process discovers a person in the environment, sends a message to the `sensor_server` process, which relayes the message to the `visibility_manager` process. The `visibility_manager` process knows the file levels of all the open files because the `file_open_monitor` process has informed it about these. It can then use these file levels to determine which windows should be visible, when the message about the person is received from the `sensor_server` process.

The `visibility_manager`, `file_open_monitor`, `sensor_server`, and `sensor_client` are among the most important processes, but the system also consists of other important processes, and some less important utility processes that are only used to retrieve information about the system state. The interactions between all the processes are specified formally, as shown in Figure 6.4 to Figure 6.8. In the reminder of this section, an informal description of this specification is provided. This is done by using the process expressions as a starting point and then describing which messages are send to or received from the channels.

visibility_manager

The `visibility_manager` process is the central server in the system that controls the visibility of windows. In order to perform this task, it must know the security parameters *no_read_up* and *no_read_down* which, along with the clearance level and window level, determines whether a window should be mapped or not. In Figure 6.4, the `visibility_manager` process will initially receive the values from `root` which are used to set the *no_read_up* and *no_read_down* parameters.

The `visibility_manager` process must also control the visibility of file names of open files. This is done using the *hide_non-readable_files* security parameter which must also be set initially. The parameter will, however, only be used when the `listwl` is invoked in order to present status information to the user. Since the specification will focus on the communication protocol, and not on how status information is presented, we will not specify how *hide_non-readable_files* is used in `listwl`. The parameter will therefore be omitted from the specification since it would not be used and only reduce the readability of the specification.

The `visibility_manager` acts as a server for many different types of clients and it can therefore receive many different types of messages. To separate the messages from the different clients, it uses many choice expressions of the form

$$c1?x1 : M1 \ \rightarrow \ P1 \ [] \ c2?x2 : M2 \ \rightarrow \ P2$$

which describes the messages $x1$ and $x2$ of type $M1$ and $M2$ that it is willing to accept from the clients via channels $c1$ and $c2$, respectively. The messages received by the `visibility_manager` will always contain at least one input value which indicate the type of the message. This will always be the first input value, and it will be bound to the the variable *op*. When an input value is irrelevant, the underscore character is used as a wildcard to indicate this.

The `visibility_manager` process uses three choice operators to distinguish between possible client messages. Messages from *left* channels with index 2, 3, 4, 5 or 8 will be ignored. Messages from *left*[6] or *left*[7] will result in a reply containing an error message since the clients that use these channels (i.e. `listsl` and `listwl`) expect a reply. Finally, messages from *left*[9] will terminate the process.

initcl

The `visibility_manager` receives an *INIT_CLEARANCE_LEVEL* message from the *left*[1] channel when a user has invoked `initcl` (see Figure 6.8). If the *permit_lower_level_login* parameter is set, the user can specify a level below the level assigned to him, which the clearance level should be initialized to. This possibility has, however, not been included in the specification.

VM

Once a user has "logged in" by running the `initcl` program, the `visibility_manager` process behaves like the `VM` process, see Figure 6.5. Four variables in `VM` are initialized when this transition takes place. First of all, a mapping of type *WindowTable* is initialized to the empty mapping. Secondly, the sequence `subject_list` is initialized so that it only contains the clearance level. The sequence *subject_list* will be used to store all subject levels: Whenever a person enters or leaves the environment, the corresponding environment level will be inserted into or removed from the list, respectively. The user level is not distinguished from the environment levels in the sequence, so it will also be noted if the user leaves the environment. The sequence is sorted in ascending order so that the current clearance level equals the first element in the list. It is assumed that only one subject, namely the user, is present initially and the sequence will therefore only contain one level. If the *subject_list* becommes empty, the clearance level will be set to zero.

Finally, the *no_read_up* and *no_read_down* variables are initialised and the `VM` process is now ready to receive messeages from its clients.

window_manager

Because the window manager plays such an important role in our system, we have chosen to include it in the specification even though we have not developed a window manager ourself. Many different window managers exists, ranging from the very basic ones with only the essential functionality to the very complex ones with many (more or less useful) extra features. All those which adhere to the *Inter-Client Communication Conventions Manual*[32] will, however, maintain the state of a window. The state can either be normal, iconic or withdrawn. Only the normal and withdrawn states are interesting in our system, so we have modeled a very simple wm that only maintains the state of windows as a boolean value: when the wm is notified about a window that is mapped, it will associate the value true with the window ID of the window that is mapped. Likewise, when a window is unmapped, it will associate the value false with the ID of the window that is unmapped.

The `window manager` process in Figure 6.8 is used to initialize the window manager, and `WM` is a non-terminating process that models how the state of windows is maintained. It is assumed that the state of a window is consistent with what is actually displayed on the screen.

editor

When a user wants to edit a file, he will initially start an `editor` process. As for the `window manager`, we have not created a editor ourself, but we will make some basic assumptions about how it works so that it can be included in the specification. First of all, it is assumed that the `editor` has a GUI that is build from a top-level window and possibly some subordinate windows that are contained within the top-level window. Only the top-level window is of interest for the system, and it is assumed that it is created using the function `x_create_window()`. Whenever an editor is started and the top-level window has been created, the `VM` should be notified about this so that it can add an entry to the *table* mapping. The `editor`does this by sending an *XCREATE_ WINDOW* message containing the window ID and process ID of the `editor`to the `VM`. A window is not visible until it is mapped, so the next step is to map the vindow and notify the `window manager` about this event. The `editor`sends an *XMAP_ WINDOW* message to the `window manager` which then updates its map, *winmap*, so that it reflects the change in window visibility. Afterwards, the `editor`is ready to be used by the user, who may for instance use it to open, read, write, and close files. This has not been modeled in the specification. Instead, the `editor` will just block until the user kills it by sending an *XDESTROY_ WINDOW* message. When it is killed, it should first of all notify the `window manager` about the state change. Secondly, the `visibility_manager` should be notified about the event so that it can remove the entry corresponding to the window ID from the *table* mapping.

file_ open_monitor

The main concern for the `VM` is not which windows are created by an editor, but rather which files are opened by editor. The sole purpose of the `file_open_monitor` process is to discover when files are opened by an editor and reporting this event to the `VM`. How the `file_open_monitor` can detect which files have been opened depends on how the sfs is implemented. To keep the specification at a high abstraction level, we have modeled this part by the function *block_until_file_opened*(),which simply blocks until the user opens a file. Whenever a new file is opened, the process ID, file level, inode number, and file name is returned. If the file level is negative, it indicates that the `seac_destroy` program has been called and that the `file_open_monitor` therefore should terminate. Otherwise, `file_open_monitor` will just act as a relay in the sense that whenever the *block_until_file_opened*() function returns a tuple, this tuple is relayed to the `VM`.

When `VM` receives a *FILE_ OPEN_MONITOR* message, it will retrieve the entry from the *table* mapping that contains the process ID of the `editor`. If the file is not already open by the `editor`, the entry will be updated so that it is registered that the file is now opened in the editor. As noted previously, it will not be registered when the file is closed since this is not possible to discover in practice.

sensor_client and sensor_server

The `sensor_client` should detect when a person or the user enters or leaves the environment. It will not distingush between persons and the users, so the term subject will be used in the following description, since it covers both.

When a subject enters the environment, the `sensor_client` should detect the level of the subject along with his direction. The `sensor_client` should not interpret these data;

instead, it should relay them to the `sensor_server`. The `sensor_server` is a server for all the different sensor types that may be deployed in the environment. First of all, it checks that the direction is valid, i.e. it must be either 'i' or 'o'. If it is not valid, the level and direction is ignored since an error has occurred. Otherwise, if the environment level is non-negative, it will relay the received data to the `visibility_manager`. As for the `file_open_monitor`, a negative level is used to indicate that the process should terminate.

When `VM` receives a *SENSOR_SERVER* message, it will initially check the direction. If it indicates that a subject entered the environment, the `VM` will examine the *subject_list* sequence. If this sequence is empty, no one was present in the environment when the person entered. The clearance level will always be zero when no one is present, and when the person enters it will most likely be increased. This person could of course just be the user who had left the environment temporarily. Because the clearance level is set to zero when the last person leaves the environment, all windows that show classified files are unmapped. When a person subsequently is detected and the clearance level is updated, some windows may have to be mapped.

The mapping of windows is managed by the `MAP` process (see Figure 6.6). This is done by examining the entries in the *table* mapping and use the stored information to determine whether the visibility of a window should be changed. The process IDs of the running editors are used as keys in the *table* mapping. The `MAP` process iterates though all these by creating a sequence of process IDs and then extracting and processing one process ID at a time. Firstly, the information value in the *table* mapping is extracted; it is a tuple containing the window ID, the application name of the editor, a sequence *file_list* of open files, and a boolean value *is_mapped* that is true if and only if the window currently is mapped. If *is_mapped* is true, the window is already mapped, so the `MAP` process can skip this window and proceed to the next process ID. Likewise, if the editor has no open files the window level is zero and the `MAP` process can skip this window. If the window is currently unmapped and has open files, the first element of the *file_list* sequence is extracted. Since the *file_list* sequence is sorted in descending order, the file level in this *FileInfo* element is equal to the window level. This window level, the clearance level and the security parameters *no_read_up* and *no_read_down* are then used to determine whether the window should be unmapped or not. If the window is mapped, the *is_mapped* will be set to `true` and the `window manager` will be notified about the state change of the window via the *left*[10] channel. The `MAP` process is then ready to extract and process the next process ID, if any is left in the sequence of process IDs.

If a person enters the environment and at least one other subject is present, the window visibility will possibly have to be changed, if the environment level is greater than or equal to the clearance level. The process used to determine this is the `UNMAP` process (see Figure 6.7), which resembles the `MAP` process. It will also process one pid at a time and then contemplate the information value which is extracted from the the *table* mapping. If it indicates that the window is already unmapped or the editor has no open files, the window will be skipped. Otherwise, the window level, clearance level, *no_read_up*, and *no_read_down* are then used to determine whether the window should be unmapped or not. If the window is unmapped the *is_mapped* is set to `false` before the next process ID, if any, is extracted from the list of available process IDs. Regardless of whether the window visibility should be changed, the environment level will be inserted into *subject_list* sequence at the proper place, reflecting that a subject is now in the environment.

Another case arises when the *SENSOR_SERVER* message received by the `VM` indicates that a subject leaves the environment. First of all, the environment level will be removed from the *subject_list* regardless of whether the window visibility should change. If it is the last person in the environment who leaves, all windows with a window level greater than zero will be unmapped. This is managed by the `UNMAP` process. If it is the person with the lowest level who leaves, i.e. the detected environment level equals the clearance level, some windows will possibly have to be mapped. This is managed by the `MAP` process.

setfl

When a new file is created by a user, the file level is set to the clearance level. In some situations it may, however, be desired to change this level afterward to another value. Only the super user is allowed to perform this change using the `setfl` program. In general, the super user should not be working on classified files. A special situation arises if a non-privileged user is capable to log in as the super user, using for example the `su` program. He may then for example have some open windows showing classified files, and the `visibility_manager` program will then have stored the levels of these files. If the user then uses the `setfl` program to change some file levels in the sfs, this will result in inconsistency if these files are open in an editor. To avoid such inconsistency, the `setfl` program will send a message to the `visibility_manager`, informing it about the file level change. (If the `visibility_manager` is not running, such a message will just be discarded by the system.) The *subject_list* sequence cannot be empty when `setfl` is called, since someone must be present in the environment in order for it to be invoked.

The `visibility_manager` will update the *table* mapping so that the file levels are updated. Furthermore, if the file level is less than the current clearance level some windows will now possibly have to be mapped, depending on whether the file level change resulted in a change in the some window levels. If the file level is greater than the current clearance level some windows will now possibly have to be unmapped.

In practice, this scenario should only take place if the user uses some form of physical access control, such as locking the door. Otherwise, a person might enter the environment and force the user to misuse the super user privileges. He could for instance force the user to change the levels of some files, or even worse, he could force him to unmount the sfs so that all files are freely available in the underlying native file system.

The described file level notification part of the system has only been included because we will not make any assumption about whether a user is also capable of logging in as the super user while he has one or more editors open with classified files. This part is, however, a vulnerable part of the system and the system will in general only be secure if the users are not capable of logging in as the super user.

listwl and listsl

The listwl and listsl programs can be used to retrive status information about the currently existing windows and the levels of subjects in the environment, respectively. When the `VM` receives a message from one of these programs, it will create a string representation of the requested inforamtion and return it.

destroy

A program has been provided that should be used to shut down the system. First of all, it should send a *DESTROY* message to the `VM`, indicating that it should terminate. Furthermore, it should ensure that the $block\_until\_file\_opened()$ stops blocking and returns a negative level so that the `file_open_monitor` also will terminate, cf. Figure 6.8.

$$
\begin{aligned}
\text{visibility\_manager} \overset{\text{def}}{=}\ & (SAPR1?(no\_read\_up : \mathbb{B}, no\_read\_down : \mathbb{B}) \rightarrow \\
& (left[1]?(op : \{INIT\_CLEARANCE\_LEVEL\}, user\_level : \mathsf{subject\_level}) \rightarrow \\
& \quad VM[\{\mapsto\}, [user\_level], no\_read\_up, no\_read\_down] \\
& [\!] left[i \in \{2,3,4,5,8\}]?\_ \rightarrow visibility\_manager \\
& [\!] left[i \in \{6,7\}]?\_ \rightarrow left[i]!INVALID\_MESSAGE \rightarrow visibility\_manager \\
& [\!] left[9]?(op : \{DESTROY\}) \rightarrow STOP)
\end{aligned}
$$

Figure 6.4: CSP specification of the visibility manager process. The visibility_manager initializes the system by means of input from the super user and a non-privileged user.

```
$VM[table : WindowTable, subject\_list : \mathsf{subject\_level}^*, no\_read\_up : \mathbb{B}, no\_read\_down : \mathbb{B}]$
    $\stackrel{\text{def}}{=}$ $(left[2]?(op : \{XCREATE\_WINDOW\}, pid : \mathsf{PID}, win : \mathsf{WID}) \rightarrow$
            $VM[table \dagger \{pid \mapsto (win, get\_application\_name(window), [], \mathsf{true})\}, subject\_list]$
        $\| left[3]?(op : \{FILE\_OPEN\_MONITOR\}, file\_name : \mathsf{string}, pid : \mathsf{PID}, level : \mathsf{object\_level},$
            $inode : \mathsf{inode}) \rightarrow$
        $\mathbf{let}\ (win : \mathsf{WID}, app\_name : \mathsf{string}, file\_list : FileInfo^*, is\_mapped : \mathbb{B}) = table(pid)$
        $\mathbf{in\ if}\ (\neg\, contains\_inode(file\_list, inode))$
            $\mathbf{then}\ VM[table \dagger \{pid \mapsto (win, application\_name,$
                        $insert\_sorted\_decr(file\_list, (file\_name, level, inode)), is\_mapped)\},$
                $subject\_list, no\_read\_up, no\_read\_down]$
            $\mathbf{else}\ VM[table, subject\_list, no\_read\_up, no\_read\_down]$
        $\| left[4]?(op : \{SENSOR\_SERVER\}, env\_level : \mathsf{subject\_level}, direction : \mathsf{char}) \rightarrow$
        $\mathbf{if}\ (direction = \text{`i'})$
        $\mathbf{then\ if}\ (subject\_list = [])$
                $\mathbf{then}\ MAP\_WINDOWS[set\_to\_seq(\mathsf{dom}\ table), table, [env\_level],$
                    $no\_read\_up, no\_read\_down]$
                $\mathbf{elseif}(env\_level < \mathsf{hd}\ subject\_list)$
                $\mathbf{then}\ UNMAP\_WINDOWS[set\_to\_seq(\mathsf{dom}\ table), table,$
                    $[env\_level]^\frown subject\_list, no\_read\_up, no\_read\_down]$
                $\mathbf{else}\ VM[table, insert\_sorted\_incr(subject\_list, env\_level),$
                    $no\_read\_up, no\_read\_down]$
        $\mathbf{elseif}(direction = \text{`o'})$
                $\mathbf{then\ if}\ (\mathsf{len}\ subject\_list == 1)$
                    $\mathbf{then}\ UNMAP\_WINDOWS[set\_to\_seq(\mathsf{dom}\ table), table, [],$
                        $no\_read\_up, no\_read\_down]$
                    $\mathbf{elseif}(env\_level = \mathsf{hd}\ subject\_list)$
                    $\mathbf{then}\ MAP\_WINDOWS[set\_to\_seq(\mathsf{dom}\ table), table,$
                        $remove\_env\_level(subject\_list, env\_level), no\_read\_up, no\_read\_down]$
                    $\mathbf{else}\ VM[table, remove\_env\_level(subject\_list, env\_level),$
                        $no\_read\_up, no\_read\_down]$
                $\mathbf{else}\ VM[table, subject\_list, no\_read\_up, no\_read\_down]$
        $\| left[5]?(op : \{SET\_FILE\_LEVEL\}, inode : \mathsf{inode}, level : \mathsf{object\_level}) \rightarrow$
        $\mathbf{if}\ (level < \mathsf{hd}\ subject\_list)$
        $\mathbf{then}\ MAP\_WINDOWS[set\_to\_seq(\mathsf{dom}\ table),$
                $update\_file\_levels(table, inode, level), subject\_list, no\_read\_up, no\_read\_down]$
        $\mathbf{else}\ UNMAP\_WINDOWS[set\_to\_seq(\mathsf{dom}\ table),$
                $update\_file\_levels(table, inode, level), subject\_list, no\_read\_up, no\_read\_down]$
        $\| left[6]?(op : \{LIST\_WINDOW\_LEVELS\}) \rightarrow$
            $left[6]!table\_to\_string(table) \rightarrow$
            $VM[table, subject\_list, no\_read\_up, no\_read\_down]$
        $\| left[7]?(op : \{LIST\_SUBJECT\_LEVELS\}) \rightarrow$
            $left[7]!list\_to\_string(subject\_list) \rightarrow$
            $VM[table, subject\_list, no\_read\_up, no\_read\_down]$
        $\| left[8]?(op : \{XDESTROY\_WINDOW\}, window : \mathsf{WID}) \rightarrow$
            $VM[remove\_window(table, window), subject\_list, no\_read\_up, no\_read\_down]$
        $\| left[9]?(op : \{DESTROY\}) \rightarrow STOP)$
```

Figure 6.5: CSP specification of the VM process, which acts as a server for many types of clients.

```
$MAP\_WINDOWS[pid\_list : PID^*, table : Window\_Table, subject\_list : \mathsf{subject\_level}^*$
                    $no\_read\_up : \mathbb{B}, no\_read\_down : \mathbb{B}]$
  $\stackrel{\text{def}}{=}$ $(\textbf{if}\ (pid\_list = [])$
     $\textbf{then}\ VM[table, subject\_list, no\_read\_up, no\_read\_down]$
     $\textbf{else let}\ pid : \mathsf{PID} = \mathsf{hd}\ pid\_list,$
          $(window : \mathsf{WID}, app\_name : \mathsf{string}, file\_list : FileInfo^*, is\_mapped : \mathbb{B}) = table(pid)$
        $\textbf{in if}\ (is\_mapped \vee file\_list == [])$
          $\textbf{then}\ MAP\_WINDOWS[\mathsf{tl}\ pid\_list, table,$
               $subject\_list, no\_read\_up, no\_read\_down]$
          $\textbf{else let}\ (\_, window\_level : \mathsf{object\_level}, \_) = \mathsf{hd}\ file\_list,$
               $clearance\_level : \mathsf{subject\_level} = \mathsf{hd}\ subject\_list$
            $\textbf{in if}\ (\neg\,(no\_read\_up \wedge window\_level > clearance\_level \vee$
                 $no\_read\_down \wedge window\_level < clearance\_level))$
              $\textbf{then}\ left[10]!(XMAP\_WINDOW, wid) \rightarrow$
                   $MAP\_WINDOWS[\mathsf{tl}\ pid\_list, table\dagger$
                   $\{pid \mapsto (window, app\_name, file\_list, \mathsf{true})\},$
                   $subject\_list, no\_read\_up, no\_read\_down]$
              $\textbf{else}\ MAP\_WINDOWS[\mathsf{tl}\ pid\_list, table,$
                   $subject\_list, no\_read\_up, no\_read\_down])$
```

Figure 6.6: CSP specification of the MAP_WINDOWS process.

```
$UNMAP\_WINDOWS[pid\_list : PID^*, table : Window\_Table, subject\_list : \mathsf{subject\_level}^*$
                    $no\_read\_up : \mathbb{B}, no\_read\_down : \mathbb{B}]$
  $\stackrel{\text{def}}{=}$ $(\textbf{if}\ (pid\_list = [])$
     $\textbf{then}\ VM[table, subject\_list, no\_read\_up, no\_read\_down]$
     $\textbf{else let}\ pid : \mathsf{PID} = \mathsf{hd}\ pid\_list,$
          $(window : \mathsf{WID}, app\_name : \mathsf{string}, file\_list : FileInfo^*, is\_mapped : \mathbb{B}) = table(pid)$
        $\textbf{in if}\ (\neg\, is\_mapped \vee file\_list == [])$
          $\textbf{then}\ UNMAP\_WINDOWS[\mathsf{tl}\ pid\_list, table,$
               $subject\_list, no\_read\_up, no\_read\_down]$
          $\textbf{else let}\ (\_, window\_level : \mathsf{object\_level}, \_) = \mathsf{hd}\ file\_list$
               $clearance\_level : \mathsf{subject\_level} = \mathsf{hd}\ subject\_list$
            $\textbf{in if}\ (no\_read\_up \wedge window\_level > clearance\_level \vee$
                 $no\_read\_down \wedge window\_level < clearance\_level)$
              $\textbf{then}\ left[10]!(XUNMAP\_WINDOW, wid) \rightarrow$
                   $UNMAP\_WINDOWS[\mathsf{tl}\ pid\_list, table\dagger$
                   $\{pid \mapsto (window, app\_name, file\_list, \mathsf{false})\},$
                   $subject\_list, no\_read\_up, no\_read\_down]$
              $\textbf{else}\ UNMAP\_WINDOWS[\mathsf{tl}\ pid\_list, table,$
                   $subject\_list, no\_read\_up, no\_read\_down])$
```

Figure 6.7: CSP specification of the UNMAP_WINDOWS process. The process unmaps all windows according to the read-security policy.

$$
\begin{array}{lcl}
\mathit{initcl} & \overset{\text{def}}{=} & (\mathit{right}[1]!(\mathit{INIT\_CLEARANCE\_LEVEL}, \mathit{get\_clearance\_level}()) \rightarrow \\
 & & \mathit{STOP}) \\
\mathit{window\_manager} & \overset{\text{def}}{=} & \mathit{WM}[\{\mapsto\}] \\
\mathit{WM}[\mathit{winmap} : \mathsf{WID} \xrightarrow{m} \mathbb{B}]) & \overset{\text{def}}{=} & (\mathit{right}[j \in \{1,2\}]?(\mathit{XMAP\_WINDOW}, \mathit{window} : \mathsf{WID}) \rightarrow \\
 & & (\mathit{WM}[\mathit{winmap} \dagger \{\mathit{window} \mapsto \mathsf{true}\}] \\
 & & [\!]\mathit{right}[j \in \{1,2\}]?(\mathit{XUNMAP\_WINDOW}, \mathit{window} : \mathsf{WID}) \rightarrow \\
 & & \mathit{WM}[\mathit{winmap} \dagger \{\mathit{window} \mapsto \mathsf{false}\}])) \\
\mathit{editor} & \overset{\text{def}}{=} & (\textbf{let } \mathit{wid} : \mathsf{WID} = \mathit{x\_create\_window}() \\
 & & \textbf{in } (\mathit{right}[2]!(\mathit{XCREATE\_WINDOW}, \mathit{getpid}(), \mathit{wid}) \rightarrow \\
 & & \quad \mathit{left}!(\mathit{XMAP\_WINDOW}, \mathit{wid}) \rightarrow \\
 & & \quad \mathit{SAP2}?(\mathit{op} : \{\mathit{XDESTROY\_WINDOW}\}) \rightarrow \\
 & & \quad \mathit{left}!(\mathit{XUNMAP\_WINDOW}, \mathit{wid}) \rightarrow \\
 & & \quad \mathit{right}[8]!(\mathit{XDESTROY\_WINDOW}) \rightarrow \mathit{STOP})) \\
\mathit{file\_open\_monitor} & \overset{\text{def}}{=} & (\textbf{let } (\mathit{file\_name} : \mathsf{string}, \mathit{pid} : \mathsf{PID}, \mathit{level} : \mathsf{object\_level}, \mathit{inode} : \mathsf{inode}) = \\
 & & \quad \mathit{block\_until\_file\_opened}() \\
 & & \textbf{in if } (\mathit{level} >= 0) \\
 & & \quad \textbf{then } \mathit{right}[3]!(\mathit{FILE\_OPEN\_MONITOR}, \mathit{file\_name}, \\
 & & \qquad\qquad \mathit{pid}, \mathit{level}, \mathit{inode}) \rightarrow \mathit{file\_open\_monitor} \\
 & & \quad \textbf{else } \mathit{STOP}) \\
\mathit{sensor\_client} & \overset{\text{def}}{=} & (\mathit{SAP4}?(\mathit{level} : \mathsf{subject\_level}, \mathit{direction} : \mathsf{char}) \rightarrow \\
 & & \mathit{right}!(\mathit{level}, \mathit{direction}) \rightarrow \mathit{STOP}) \\
\mathit{sensor\_server} & \overset{\text{def}}{=} & (\mathit{left}?(\mathit{level} : \mathsf{subject\_level}, \mathit{direction} : \mathsf{char}) \rightarrow \\
 & & \textbf{if } ((\mathit{direction} \neq \text{`i'} \wedge \mathit{direction} \neq \text{`o'})) \\
 & & \textbf{then } \mathit{sensor\_server} \\
 & & \textbf{elseif}(\mathit{level} >= 0) \\
 & & \textbf{then } \mathit{right}[4]!(\mathit{SENSOR\_SERVER}, \mathit{level}, \mathit{direction}) \rightarrow \mathit{sensor\_server} \\
 & & \textbf{else } \mathit{STOP}) \\
\mathit{setfl} & \overset{\text{def}}{=} & (\mathit{SAPR2}?(\mathit{inode} : \mathsf{inode}, \mathit{level} : \mathsf{object\_level}) \rightarrow \\
 & & \textbf{if } (\neg\, \mathit{set\_file\_level}(\mathit{inode}, \mathit{level})) \\
 & & \textbf{then } \mathit{STOP} \\
 & & \textbf{else } \mathit{right}[5]!(\mathit{SET\_FILE\_LEVEL}, \mathit{inode}, \mathit{level}) \rightarrow \mathit{STOP}) \\
\mathit{listwl} & \overset{\text{def}}{=} & (\mathit{right}[6]!\mathit{LIST\_WINDOW\_LEVELS} \rightarrow \\
 & & \mathit{right}[6]?(\mathit{window\_info} : \mathsf{string}) \rightarrow \\
 & & \mathit{SAP6}!\mathit{window\_info} \rightarrow \mathit{STOP}) \\
\mathit{listsl} & \overset{\text{def}}{=} & (\mathit{right}[7]!\mathit{LIST\_SUBJECT\_LEVELS} \rightarrow \\
 & & \mathit{right}[7]?(\mathit{subject\_list} : \mathsf{string}) \rightarrow \\
 & & \mathit{SAP7}!\mathit{subject\_list} \rightarrow \mathit{STOP}) \\
\mathit{seac\_destroy} & \overset{\text{def}}{=} & (\mathit{right}[9]!\mathit{DESTROY} \rightarrow \mathit{STOP})
\end{array}
$$

Figure 6.8: CSP specification of the client processes. The `sensor_server`is both a client and a server.

Chapter 7

Implementation

A system that conforms to the model described in Chapter 5 and the design described in Chapter 6 has been implemented. Many different technologies have been used for this implementation, ranging from a file system description using the FiST language at the lowest layer to GUI programming in Java at the highest layer. In this chapter, the implementation of the SEAC system is described, starting at the lowest abstraction layer and ending at the highest. The focus will be on how the system is implemented, and not on how it should be used since these details are described in the user's guide in Appendix C. For instance, all the programs that were described at a high level in Chapter 6 are reviewed in the user's guide, where we also introduce the command line arguments.

In Section 7.1, we will describe how a stackable file system can be used to store and retrieve file levels and user levels and provide logical access control based on these levels and a security policy. The combination of the logical and physical access control is handled by the `window management` subsystem. In Section 7.2, we will describe how this part is implemented, using the CSP specification developed during the design phase as a starting point. Furthermore, some issues regarding the handling of backup files that were not foreseeable during the design phase are also described. The detection of persons is performed using two web-cameras, a motion detection program, and programs that analyze the motion detection output and use parameters to determine whether persons have entered or left the environment (see Section 7.3). Finally, the implementation of the Security Management GUI, which can be used to indirectly run command-line programs, is presented in Section 7.4.

7.1 The Stackable File System

A stackable file system that must enforce logical access control has been implemented using a FiST input file and C kernel code when the FiST language did not suffice. More precisely, the code related to the storage of the file and user levels and the reading of the security policy parameters have been written in three separate C files. The functions provided by these files are invoked in the FiST input file, for example when a user level has to be retrieved from the user level file. The three files where the file levels, user levels, and security policy are stored must be specified when the SEAC system is started up and the `seac_init` program is run. File levels should not be associated with these files, and they should therefore not be stored in `macfs`; instead, they should preferably be stored in a directory where only the super user has write access to them. For further details about the system startup, see Appendix C.1.

7.1.1 Storage of Levels

In order for the stackable file system to provide MAC, it must associate a file level with each inode number in the file system. This is implemented using a hashtable, where the key is the inode number and the information value is the file level (see Appendix F.2.3). When the SEAC system is in use, this hashtable is stored in memory; when the system is shut down, the hashtable will be stored in a regular file which is specified by the super user at system startup. The main reason for storing all the file levels in memory is that this yields better performance when the file levels must be retrieved frequently. Furthermore, the memory requirements by the system are relatively low: each entry in the hashtable requires 8 B (4B for the user level and 4B for the user ID). Unless many millions of files are created, this approach should not course memory consumption problems.

The users obtains access to objects via processes, and it is the user ID associated with the process that is used to determine whether the access should be granted or denied. To enforce a MAC policy, the stackable file system must therefore store a user level for each user in the system. This is done by the super user with the `setfl` program, which will store the user level in a file along with the user ID. Unlike the file levels, the user levels are not cached in memory, so when a user level should be retrieved, it is read directly from this file (see Appendix F.2.4).

7.1.2 FiST Input File

In this section, we will describe the `macfs` file system using the FiST input file shown in Appendix F.2.1 as a starting point. The details regarding the FiST language will not be described in this thesis; instead, we refer to [44]. We will, however, describe the parts of the language that we have used.

The FiST input file specifies the functionality required by our stackable file system. It is divided into four sections as shown in Figure 7.1. The first section is enclosed between %{ and %}, and it contains *C declarations* for the used variables (e.g. for the the security policy parameters) and function prototypes.

%{
C Declarations
%}
FiST Declarations
%%
FiST Rules
%%
Additional C Code

Figure 7.1: FiST grammar outline

The second section contains *FiST declarations*, which are declarations that globally affect the behavior of the code produced by the FiST generator. Initially, this section specifies which other files contain C kernel code, and which files contain user space code that should use the functionality provided by the stackable file system. Most importantly, it defines special data structures that are used by the rest of the stackable file system code.

The third section contains *FiST Rules*, which describe rules for controlling the behavior of file system functions. The rules can also specify the actions taken by the stackable file system when an ioctl system call is invoked.

The fourth section contains *Additional C code*, and its main purpose is to provide a flexible extension mechanism that enables the integration of C and FiST code. The section may contain arbitrary C code that can be called from anywhere. For example, it contains the function that is invoked whenever the access permissions for a given inode is checked in the VFS layer.

Defining New I/O Controls

The FiST declarations are mainly data structures that are used for declaring new ioctl requests. Ioctls (I/O controls) are used in the FiST system as an operating extension since they can be used to pass arbitrary data between user space and the kernel. Each FiST declaration of an ioctl specifies the variables that should be passed between user space and kernel space. When the FiST generator `fistgen` is run on the FiST input file, it will generate a C struct definition that contains the required variables and a code that can be used to identify the ioctl. For example, the `GET_FILE_LEVEL` ioctl declaration specifies a `level` and an `inode` variable, and `fistgen` will generate the C struct `_fist_ioctl_GET_FILE_LEVEL` which can hold these two variables, and an ioctl code stored in the variable `FIST_IOCTL_GET_FILE_LEVEL` of type `int`.

For each of the FiST ioctl declarations, the FiST rules section contains a corresponding FiST rule. For example, when the `getfl` program invokes an ioctl system call to retrieve a file level, it will initially set the inode number of the file in the C struct. When the operating system makes a context switch and transfers control to macfs, the `%op:ioctl:SET_FILE_LEVEL` FiST rules is used. This rule will extract the inode from the C struct, find the corresponding file level and store this in the C struct. When control is transferred back to the `getfl` program, it can extract the file level from the C struct and print it to standard output, making it visible to the user who initially started the program.

The ability to define new ioctl codes and implement their associated actions is the functionality provided by FiST that we have used most extensively: For each of the user space programs that must pass data to or receive data from `macfs`, a FiST ioctl declaration and corresponding rule has been implemented. There are 12 such programs: `getul`, `setul`, `listul`, `getfl`, `setfl`, `listfl`, `visibility_manager`, `getcl`, `initcl`, `seac_init`, `seac_destroy`, `file_open_monitor`. When a user space program needs to interact with the stackable file system via an ioctl system call, it must first of all open a file and thereby retrieve a file descriptor to the file system. When the ioctl system call is invoked, this file descriptor is used as argument in addition to the C struct and ioctl code generated by `fistgen`. An arbitrary file in `macfs` can be used, since all the MAC is implemented in the kernel, independently of which file was opened. The only requirement is that the programs will be granted access to opening the file. One file (or rather directory) that exists and should be readable by all is the mount point, and the 12 mentioned programs will all use the mount point when they need to retrieve a file descriptor to `macfs`.

The mount point is defined to be "/mnt/macs/" in the header file `mount_point.h` (see Appendix F.1.1), and this file should be modified if another mount point should be used. This inflexible approach to specifying a constant has been used because there are 12 programs that must know the mount point, and it would make the programs less user-friendly if the mount point should be specified as a command line argument for each of them. Another approach could be to store the mount point in shared memory, but then all the programs should know the shared memory ID. We have taken the rather inflexible approach because we believe that the mount point rarely will need to be changed, if ever.

Extending File System Functions

Besides the rules for all the ioctl declarations, the FiST rules section specifies rules for altering the behavior of some system calls related to files and directories. First of all, the behavior of the file operations `create` and `unlink` are slightly modified after the corresponding operations are called in the lower file system:

%op:create:postcall When a new file is created, its file level will be initialized to the clearance level. The inode number of the file is associated with this file level by inserting an entry into the hashtable. If a hard link subsequently is created to the file, the files will have the same file level since they have the same inode number.

%op:unlink:postcall When the `unlink` operation is invoked, the number of hard links associated with a given inode is reduced by one. If the number of hard links is zero after the `unlink` operation has been invoked in the lower file system, the entry in the hashtable that contains the inode is deleted.

The behavior of the directory operations `mkdir`, `rmdir` and `readdir` have been extended by three FiST rules:

%op:mkdir:postcall When a new directory is created, its file level will be initialized to the clearance level. The rule is identical to the `%op:create:postcall`, except that it works on directories.

%op:rmdir:postcall When a new directory is deleted and the number of hard links becomes zero, its entry in the hashtable is deleted. The rule is identical to the `%op:unlink:postcall`, except that it works on directories.

%op:readdir:call If the security policy *hide_non-readable_files* has been chosen and the caller is not the super user, all the non-readable files will be skipped in when the `readdir` system call is invoked. The effect of this can be seen when a user, for example, calls the `ls` or `listfl` programs and they read a directory entry: all the names of non-readable files will be omitted in the generated file list.

File Open Detection

The program `file_open_monitor` is used to notify the `visibility_manager` about files that are opened in `macfs`, ensuring that the window levels maintained by `visibility_manager` remain updated. The kernel space part of the `file_open_monitor` program is implemented using the `%op:ioctl:OPEN` rule, which blocks on the semaphore `open_sem` until a file is opened. This rule interacts with the `file_open_intercepted()` function in the fourth section of the FiST input file via `open_sem`. `file_open_intercepted()` is invoked whenever a file is opened, i.e. whenever the `open` system call is invoked by an editor. If the opened file is a regular file, `file_open_intercepted()` will copy the editor's process ID, the file level, the inode number and the file name into global variables. `file_open_intercepted()` will subsequently increment the value of the `open_sem` semaphore, indicating that the global variables contain information about a newly opened file. The `%op:ioctl:OPEN` rule will then stop blocking on `open_sem` and copy the values of the global variables into buffers which can be read by the user space part of the `file_open_monitor` process. The `file_open_monitor` will subsequently send a message containing these values to the `visibility_manager`. Afterwards, it will block again until a new file is opened, repeating the described steps until `seac_destroy` is invoked and kills it.

The FiST language cannot be used to add code that should be executed whenever the `open` system call is invoked. Since it is essential for our system that it is detected whenever a file is opened, we added an invocation of `file_open_intercepted()` to the function `wrapfs_open` in the templates. `wrapfs_open` is invoked whenever the system call `open` is invoked. The changes made to the templates are described in detail in Appendix B.1.1.

The global variables, which contain information about an opened file, are read by a single process (`file_open_monitor`) and written by all processes that open a file in `macfs`. This use of global variables in the kernel should not cause concurrency issues, if the used kernel is non-preemptive and the computer only has one processor. The Linux 2.4 kernel which our system is developed for is non-preemptive[27], but SMP (symmetric multiprocessing) issues will arise on computers with multiple processors. On a single processor computer, a process that invokes the `open` system call will not be interrupted while it is assigning new values to the global variables.

Mandatory Access Control

The MAC enforced by the system is implemented in the `inode_permission()` function in the fourth section of the FiST input file. This function should be invoked whenever a process attempts to access a file or directory in the `macfs`. One way to ensure that it is invoked is by modifying the stackable file system templates slightly so that our `inode_permission()` function is invoked whenever the operating system performs access checks. More precisely, the function `wrapfs_permission()` in the templates is invoked whenever a Unix permission check is performed. `wrapfs_permission()` takes a mask of type int as argument, and when read access to a file is checked the mask is 4, and when write access is checked the mask is 2. The `inode_permission()` function uses the same mask to determine whether read or write access should be granted. Furthermore, it will use the *no_read_up*, *no_write_down*, *no_read_down*, and *no_write_down* parameters, the file level associated with the inode, and the clearance level to determine whether the access should be granted. If not, the error code `EPERM` is returned and the user space program will therefore see the error message 'permission denied'. If the user ID associated with the process is zero, i.e. it is the super user, the access will always be granted.

Besides the access control that is determined by the security policy parameters, many of the FiST ioctl rules also enforces access control: for those user space programs that must only be invoked by the super user, it is checked that the user ID associated with the process is zero. It is essential for the security of the system that the user ID is checked in the kernel and not in the user space programs. If the checks were made in a user space programs, a malicious user could easily circumvent the access control by writing his own program where no access control is enforced. The program should use the ioctl system call to obtain access to the file system, and with no access control he could, for example, change file levels and user levels.

Auditing

The macfs provides a primitive audit mechanism: if an erroneous situation occurs, an entry is written in the system log using the `printk` function. This could for example be an 'out of memory' error or that a file level was unexpectedly not associated with an inode. On most Linux systems, the system log is stored in `/var/log/messages` and only the super user has read and write access to this file. Whenever an entry is written to the log, it is automatically preceded by the date and time.

If set DEBUG macro is set, additional debugging information is written to the system log. The `macfs` will record many types of events, such as the creation or deletion of files or directories, the setting of a file or user level and the retrieval of a file or user level. By default, DEBUG is turned of, but it can be set at compile time as explained in Appendix B.1.

7.2 Window Management

The `window management` subsystem involves many processes that send messages to each other, as shown in the CSP specification in Section 6.8. This message passing has been implemented using many different types of IPC mechanism, as described in this section. The IPC mechanisms are XEvents, sockets, semaphores, shared memory, and named pipes. The reasons for choosing these mechanisms will be motivated as the processes that uses them are described.

Three of the programs in the `window management` subsystem (`visibility_manager`, `file_open_monitor`, and `sensor_server`) must be started by the super user and subsequently run as demon processes. It is very important from a security perspective that they are started by the super user, since they then will be protected by the Unix security mechanisms. In particular, a non-privileged user must not be allowed to kill them by sending them a signal, regardless of whether this is done deliberately or not.

7.2.1 The Visibility Manager

The most important program in the `window management` subsystem is the `visibility_manager` (see Appendix F.5.1): it acts as a server for many other processes and is responsible for changing the visibility of editors. In the CSP specification for the `visibility_manager` process, the two data structures *table* and *subject_list* are used to maintain the state of the `window management` subsystem. The *table* stores the window related data and is implemented using a hashtable. The *subject_list* stores the detected subject levels and is implemented using a singly linked list. We have not implemented these data structures ourselves, but have used the `GHashTable` and `GSList` provided by the `glib` library[8]. This library is developed as part of the GNOME project, and we have used it extensively in the user space part of our code since it provides many general purpose functions and data structures.

In the CSP specification, we used a choice expression to separate the different messages that are received from the different types of clients. This part is implemented using an event loop, where a queue of pending events is maintained. The `visibility_manager` will repeatedly extract one event from the queue, process it according to its type, and then possibly block until a new event is added to the queue. By using an event loop, the `visibility_manager` acts as a single threaded server that processes one client request at at time.

One IPC mechanism that can provide a queue of pending events is the Unix message queue. Another, more unconventional, mechanism is to use XEvents. An XEvent is a data structure that is included in the `Xlib` library. It can contain many different types of data because it is the principal method by which clients get information from the X server or other X clients. We have chosen to use XEvents because the `visibility_manager` has to receive events from the X server whenever a window in its table is destroyed. A single event loop can therefore only be implemented using XEvents.

An XEvent can only be sent to an X client, i.e. an application that has created window. As part of its initialization, the `visibility_manager` becomes an X client when it creates a window. The window is not mapped so it is invisible. The other processes must know the window ID of this invisible window before they can send XEvents to the `visibility_manager`. The window ID is written to shared memory, and all the clients of the `visibility_manager` will subsequently read it there.

7.2.2 Intercepting Window Creation and Destruction

The system has been designed to work with editors that only use one top-level window, since there then is no ambiguity about which window should be mapped or unmapped. However, not all editors works this way, and there exists no standard which specifies that all applications should use the same window hierarchy. In fact, the ICCCM warns that "clients must be aware that some window managers will reparent their top-level windows so that a window that was created as a child of the root will be displayed as a child of some window belonging to the window manager". The system can therefore not be used with all editors, and a user must therefore check that his editor can be unmapped before he opens classified files in it. Our tests of the system (see Appendix D.6) showed that the system works with `emacs`, `nedit`, and `mozilla`, but not with `gedit`. We used the GNOME desktop environment during the test with the default window manger, which currently is `metacity`. The `gedit` editor created many top-level windows, and it its behavior was highly unpredictable. `emacs`, on the other hand, was very simple in this respect since it only uses one top-level window. We think that the main reason why the behavior of `gedit` is unpredictable is that it is tightly integrated with the GNOME desktop environment and therefore can use a custom protocol when it communicates with `metacity`. After all, this unspecified behavior is permitted since the X Window System provides mechanism, not policy.

Whenever a new top-level window is created, the `visibility_manager` process must be informed about its window ID and the process ID of the application that created the window. The window ID can be received from the X server since any X client can request to be notified about window creation events[1]. The process ID cannot be retrieved from the X server, but if both the editor and the window manager adheres to EWMH, the process ID can be retrieved from the window manager. Our experiments showed that a process

[1]In fact, an X client can select to be notified about all events for any window. If keystroke events are selected, the X client can eavesdrop a password from any other client on an accessible display[31].

ID for a `gedit` process could be received from `metacity`, but a process ID could not be received for `emacs`. We do not want to make our system depended on a protocol which (at the time of writing) only is supported by applications that are tightly integrated with a desktop environment. Another means of detecting the process ID of a newly started editor process has therefore been used.

The behavior of an editor can be modified by using a preloaded shared library. A *preloaded shared library* is a shared library that is loaded before any other shared library when a dynamically linked program is executed. By creating a preloaded shared library, one can easily overload functions that belong to the real shared libraries. We have used this approach to modify the `XCreateWindow` and `XCreateSimpleWindow` functions in the library `libX11` (see Appendix F.6.1). Whenever a top-level window is created, these functions will send a message containing the process ID and window ID to the `visibility_manager`. The preloaded shared library will only be used if the environment variable `LD_PRELOAD` is set. The details regarding the use of preloaded shared library are described in the SEAC user's guide in Appendix C.1.2. For further reading on shared libraries, see [20, 41].

When the `visibility_manager` receives information from the preloaded shared library about a newly created top-level window, it will add a corresponding entry to its table. Furthermore, it will requests that the X server sends an XEvent when the window is deleted. When this occurs, a message of type `DestroyNotify` is received in the event loop. It contains the window ID of the destructed window, and the entry in the table that contains this window ID is subsequently removed.

7.2.3 Intercepting File Open

An XEvent can only be used to send up to 20B of data. This is sufficient for all the messages, except when the `file_open_monitor` programs has to send a process ID, application name, inode number and file name to the `visibility_manager` (see Appendix F.5.2). Another IPC mechanism is therefore required, and we have chosen to use shared memory, which is the fastest form of IPC[29]. The file name is written to shared memory, whereas the remaining data are sent in an XEvent. To avoid synchronization issues when files are opened rapidly one after another, a binary semaphore is used to protect the shared memory (see Appendix F.1.3). Whenever `file_open_monitor` detects that a file is opened, it will first wait on the semaphore until the semaphore value is positive. Then it decrements the semaphore value by one, writes the shared memory, and sends the XEvent that contains the remaining data to the `visibility_manager`. When the `visibility_manager` receives the XEvent, it will first read the shared memory and then increment the semaphore value by one. The shared memory can then again be written by `file_open_monitor`.

7.2.4 Handling of Backup Files

The handling of backup files depends highly on the editor: some editors create no backup files, whereas others create more than one. The editor that we primarily have used during the development and test of the system is `emacs`. The system is only a prototype, and since the handling of backup file is not an essential part of its functionality, we have chosen to implement it so that it can handle the backup files created by `emacs`. Our experiments with some of the available editors in the Fedora Core 1 Linux distribution showed that `nedit` does not create any backup file; this is probably due to that it is a very simple editor, at least when compared with `emacs`. The other editor we tested was `gedit`, and our experiments showed that it uses the same backup formats as `emacs`.

The behavior of an editor can be examined using the `strace` program. `strace` intercepts and records the system calls which are invoked by a process and the signals it receives. Using `strace`, one can observe how backup files are handled by an editor. We will describe how `emacs` handles backup files by using an example. The file `test_file.txt` will be used in the example. Initially, the `stat` program was used to retrieve its inode number as follows:

```
localhost(s973732) $ stat --format "%n %i" test_file.txt
test_file.txt 340726
```

The creation of backup files involves `open` and `rename` system calls, so `strace` was used to intercept these calls as follows:

```
strace -etrace=open,rename -e signal=none emacs test_file.txt
```

Each line printed by `strace` contains the system call name, followed by its arguments in parentheses and its return value:

```
open("/mnt/macfs/test_file.txt", O_RDONLY|O_LARGEFILE) = 3
...
rename("/mnt/macfs/test_file.txt", "/mnt/macfs/test_file.txt~") = 0
...
open("/mnt/macfs/test_file.txt", O_WRONLY|O_CREAT|O_TRUNC|O_LARGEFILE, 0666) = 3
```

Most of the system calls have been deleted, as denoted by the dots. This output shows that `emacs` creates a backup file be renaming the original file so that a tilde character is appended to the end of the file name. A new file is subsequently created with the same name as the original file. In other words, the inode that was originally associated with `test_file.txt` will after the creation of the backup file be associated with `test_file.txt` , and a new inode is associated with `test_file.txt`. This can be verified using the `stat` program:

```
localhost(root) $  stat --format "%n %i" test_file.txt
test_file.txt 340728
/mnt/macfs
localhost(root) $  stat --format "%n %i" test_file.txt~
test_file.txt~ 340726
/mnt/macfs
```

A file level is associated with an inode number, and a newly created file will acquire a file level equal to the clearance level. If the file level of the edited file and the clearance level are identical, this will course no problems. If, however, they are different the file level of the edited file will become the clearance level. Our model specifies that only the super used is allowed to change the file levels, so it is definitely not desirable that a user started editor can change these. Furthermore, since the `visibility_manager` stores the file level along with the file name, this file level will no longer be consistent with the file level stored by `macfs`.

To solve this issue, we have developed a preloaded shared library that will intercept whenever a `rename` system call is made (see Appendix F.6.2). It will compare the two file names, and if this comparison indicates that a backup file is being created, it will send a message to the `visibility_manager`. This message will include the process ID and application name of the editor and the inode number of the file that is being renamed. If the `visibility_manager` can find an entry in its table that contains these data, it will know that a backup file has been created. Since the inode number after the rename operation will belong to the backup file, it will be marked in the table by setting it to a negative value. As the example using `strace` illustrated, the newly created file will be opened immediately after it is created. The `file_open_monitor` process will discover this and send a message containing the process ID, file level, inode number and file name to the `visibility_manager`. The `visibility_manager` will find the entry in its table corresponding to the edited file and replace the negative inode number with the new inode number. Finally, the file level is updated in the `macfs`. Although the handling of backup files turned out to be a bit complicated, it succeeded: both the edited file and its backup file will get the correct file levels.

In the message that is send from the preloaded shared library to the `visibility_manager`, we have included a constant `EMACS_BACKUP`. This information is used to ensure that a file is only marked as a backup file when the application name is "emacs". In a further development of the system, new constants can be added if new backup file formats should be handled.

As a final note about the handling of backup files, the temporary files created by `emacs` of the form `#test_file.txt#` will also be handled by the system. The `visibility_manager` will discover that this format is used and set the file level of `#test_file.txt#` to the file level of `test_file.txt`.

7.2.5 The Sensor Server

The `sensor_server` is implemented using basic socket programming (see Appendix F.5.3). It creates a server socket that listens on the port that was given as command line argument or the default port if none was specified. Whenever a `sensor_client` connection is accepted, it creates a socket that is used for receiving a level and direction and sending back an error code. The socket is then closed, and the server is then ready to accept a new connection. It will loop until a negative level is received, at which point it closes the server socket and terminates.

Because the `sensor_server` only uses basic socket programming, it is very insecure: anyone who knows the host and port number can send fake directions and levels to the server and thus completely compromise the access control provided by our system. One solution would be to use SSL to provide confidentiality, integrity, and authentication. We have not secured the communication due to lack of time, but it will properly not be difficult using for example the OpenSSL[9] implementation of SSL.

7.2.6 Printing Subject and Window Status Information

When the `listwl` program or the `listsl` program are started, they send an XEvent to the `visibility_manager`, requesting data that subsequently will be printed to standard output. This scenario is unlike the other clients that did not receive data from the visibility_manager, except possibly an error code. In addition, the size of the returned data is not easy to determine since the table and subject_list in visibility_manager process changes frequently. This makes it impractical to use an IPC mechanism that only provides storage for fixed-sized messages, such as shared memory or message queues. A FIFO special file (or a named pipe) does not impose limits on the size of the transmitted data. This IPC mechanism is therefore used when the visibility_manager communicates with either a listwl or listsl process. Another possible IPC mechanism that could have been used is sockets.

The code for the listwl and listsl are listed in Appendix F.5.4 and Appendix F.5.5, respectively. The listwl program prints all the data which are stored in the hashtable maintained by the visibility_manager. The output is formatted so that it easily can be processed by the Security Manager GUI. In figure Figure 7.2, the output corresponding to the screen-shots in Figure E.5 to E.7 are listed. The output printed by the listsl program is rather simple: it is a list of non-negative integers, corresponding to the subject levels stored by the visibility_manager.

7.3 Sensors

The sensor subsystem consists of the cameras themselves as well as some programs to analyze the input from the cameras. A number of constants based on empirical determined data is used.

7.3.1 Web-cameras

We are using two simple web-cameras: a Logitech Quickcam Express (old model) and a Logitech Quickcam Web that works with the qce-usb driver[10].

We need to have two cameras to determine the direction of movement. If we only had one, it would be required that only one person could be in the office at any time, as two consecutive detections of motion would be seen as an enter/leave pair. This would probably be a very unrealistic requirement. When using two web-cameras we can use the time difference between the motion detected in front of each to determine which direction the person is moving. When doing this the placement of the cameras is important. We need the two cameras to be placed so the motion detection in front of each of them has a time difference when people walk by. In our empirical tests we can detect most persons entering at a normal pace with 40-50 cm between the cameras. To detect running persons they should be further apart. It is important to note that the distance between the cameras partly determines some constant values in the programs.

7.3.2 Motion, a Motion Detection Program

Motion is motion detection software for Linux licensed under the GNU general public license (GPL). It grabs images from a web-camera and/or video4linux device and uses them to detect motion.

We chose to use this program as it mostly covers our needs. Our system is modular, so other motion detection software could replace it if needed. For our use, the main feature of Motion is the ability to detect movement and take snapshots of it, as well as the ability to

```
Table content:

Application Name    PID  Window ID     Security Level  Is mapped
emacs              3636   54526168                  4          1
gedit              3637   60817454                  4          1
gedit              3638   56623150                  4          1
mozilla-bin        3573   46137393                  2          1
emacs              3639   58720472                  6          0
nedit              3574   44040197                  5          1

Open files in window 54526168:
File Name           Level
file4.txt               4
file3.txt               3
file2.txt               2
file1.txt               1

Open files in window 60817454:
File Name           Level
file4.txt               4
file3.txt               3

Open files in window 56623150:
File Name           Level
file4.txt               4
file3.txt               3
file1.txt               1

Open files in window 46137393:
File Name           Level
file2.txt               2

Open files in window 58720472:
File Name           Level
'unavailable'           6
file5.txt               5
file4.txt               4
file3.txt               3
file2.txt               2
file1.txt               1

Open files in window 44040197:
File Name           Level
file5.txt               5
```

Figure 7.2: Sample window status information as it is printed by the `listwl` program. The clearance level was 5 when this was printed, and the Bell-LaPadula access control model was used.

control more than one camera. The software also has many other features, such as creation of mpeg videos, live web-cam, ability to save images directly into a database and much more.

Motion captures a picture every time there is a pixel wise change with respect to a reference picture. The reference picture is updated when a new picture is taken. After N pictures has been taken the reference picture will consist of[11]:

$$\text{Reference Picture} = \sum_{n=1}^{N} \frac{1}{2^{N\text{-}n+1}} * \text{Picture } (n)$$

This means that several snapshots are taken while one person enters the room. This will result in the need for some additional processing before we can acquire the knowledge we need, namely the determination of when one person enters or exits the office. Motion has the capability to start a program or script any time a snapshot is taken, and this how we will get data from Motion.

Configuration files

Motion uses configuration files to initialize the processing of each camera output, and the use of these is described in the *Motion Guide* [26]. Different configuration files will be used for each camera. The configuration files contain a lot of different options that can be set, for instance commands to make mpeg movies from the snapshots taken. In the following, we will go through the options and values we have changed or used. Our configuration files, `motion.conf` and `thread1.conf`, are very long and have therefore not been included in this report; instead, they can be found along with the code for the motion programs. We are using two cameras so two configuration files are needed. This tells Motion to use two cameras and will set the options that should be different for the different cameras.

videodevice: The device associated with this configuration thread. We use `/dev/video0` and `/dev/video1` for the two threads.

framerate: This is the maximum number of images that Motion will save per second. The default is 100, we have set to 20, as we do not need any more.

threshold: This is the threshold for pixel wise changes needed to declare motion. The default value is 1500, we use 3000 to eliminate smaller movement. It might be possible to set it even higher.

noise_level: This is to filter random noise from the motion detection. It operates on pixel level and specifies how many intensities the pixel must change to be taken into consideration. We have kept at the default at 32.

lightswitch: A light switch filter that should prevent Motion form classifying sudden light differences as motion. We have set is to `on`, but it does not work perfectly.

quiet: Specifies that Motion should be quiet and not emit a beep every time it is detecting motion. We have no use for an auditive indication of motion, so this is set to `yes`.

onsave: This tells Motion to execute a program when an image is saved. We use it to execute either `event1` or `event2` depending on the thread/camera.

snapshot_overwrite: Specifies that all snapshots should be saved as lastsnap.[jpg/ppm] instead of a path and name determined by the time. This would be a nice feature to use as we then would not have to clean up after all the pictures saved, but it does unfortunately not seem to work.

target_dir: The directory to save images in. We have two different paths depending on which camera it is.

thread: This is for management of more than one camera. Each **thread** `threadname` option starts a separate thread for the second or 3rd camera. This one uses the options set in the new thread. It is important to note that the first camera gets all its options from `motion.conf`, and the second one gets them from the additional thread. Also note that in newer versions of Motion this have changed, please refer to the documentation if using a newer version of Motion.

Considerations when using Motion

There are some general issues we need to consider when using Motion. Firstly, it seems that in the upstart phase, Motion will detect a lot of changes. This is of because of the update of the reference picture, it needs to be established. The way around this is to start Motion before the rest of the system.

Another problem is that Motion takes snapshots of the persons, and we do not wish to make the persons recognizable on them due to privacy concerns. It does seem like Motion has no control over the resolution of the snapshots taken. We need to make use of the **onsave** option to start a program when motion is detected, so images of the persons entering and leaving the environment will be taken. It might be possible to reduce the resolution of the pictures in the driver software, but that will be beyond the scope of this project.

Even with these limitations, it is still an advantage using Motion. It does what is needed, and detects motion based on differential motion analysis, while having many configuration options. There is no point in making our own program when one that meet our needs already exists.

7.3.3 Motion Detection Programs

Some programs are needed to handle the output from Motion, the output we use is the fact that a snapshot was taken, signifying motion has been detected, and the time this happened.

The overview of the software architecture for the camera sensor can be seen in Figure 7.3. Motion reads data from the web-cameras through the driver, and it then uses the functionality of executing a command every time an image is saved. Depending on which camera the image came from it executes either `event1` or `event2`. The 'event' programs then communicate through a pipe with the `motion_handler`, which runs in one instance for each camera. The `motion_handler` will then process the fact that Motion have detected some motion at this point in time. When it has affirmed that person have passed the camera, it will pass this information on to the `camera_client`. The `camera_client` determines the direction of the movement based on the data sent from the two `motion_handler` instances. It will then pass information about the person and the direction on to the `sensor_server`. In a addition to these programs, the program `start_motion` will be used to start the two `motion_handler` instances.

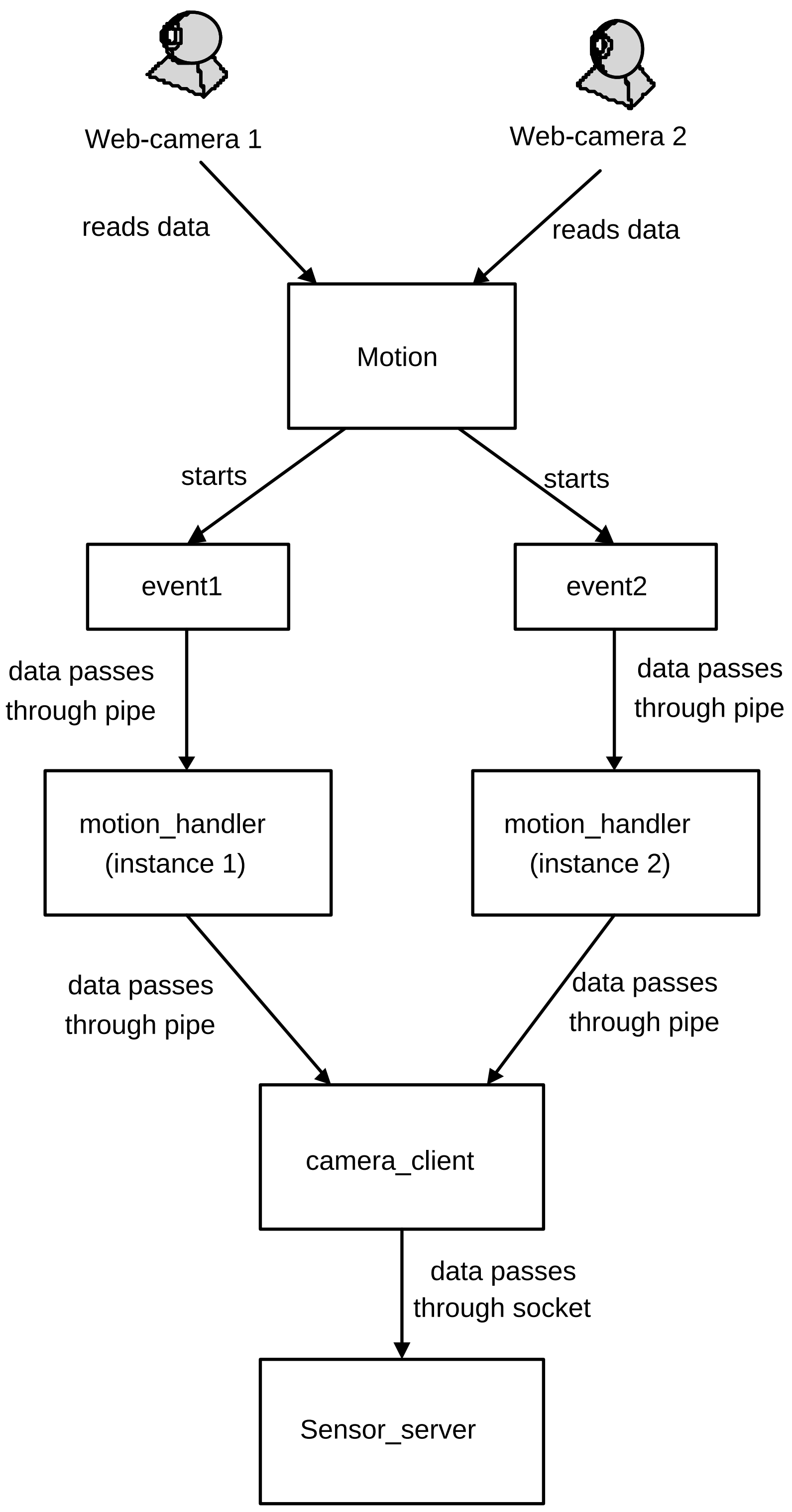


Figure 7.3: The Camera Software Architecture

The programs mainly communicate via named pipes. We have chosen the named pipes since they are easy to implement and they require synchronization of the reading and writing programs. In this way the motion_handler and the camera_client can wait for the data they need in an orderly fashion.

In the following we will describe the programs in more detail, especially the motion_handler, as this has the most interesting logic.

The Motion Handler Program

The main purpose the motion_handler is to receive an event each time snapshot is taken, and from that determine whether or not a person have passed the camera. The result is then passed on to the camera_client. The source code for the motion_handler can be seen in Appendix F.7.2.

The program uses two threads, one to receive events and one to process the data and pass the results on. Because of this the main data structures need to be globally declared. The program takes two arguments, namely the names of the pipes used. The first argument is the name of the pipe via events are received, the second is the one via data to the camera_client is sent. The pipe names are needed as arguments as the motion_handler is run in two instances. The user will not need to be concerned with the arguments as the motion_handler instances are started by start_motion which provides the correct names.

The main data structure of the program is a list in which to store the events received. Each received event signifies that a snapshot was taken by Motion from the associated camera, and the list contains the time the event was received. The list used is a singly linked list, and we have used the GSList from the glib library. We use this as there is no reason to code something that already exists, and the whole system already uses the glib libraries. The oldest snapshots are stored last. The program also contains an auxiliary print function used to print the contents of the list for debugging purposes.

The program uses two threads so we can process events while we receive them. Threads are preferred over using other means of concurrent execution, for instance fork, as the threads have access to shared variables. This is desirable when the different parts of the program needs to communicate. We make of use POSIX threads, using the pthread interface. As the program will be a background process and run until forcibly quit it uses detached threads. Detached threads are non-joinable, and therefore the system automatically frees the thread's resources when the thread is terminated.

Both threads will access the linked list, so to prevent race conditions a mutex is used to protect the list. To ensure that the event receiving thread has access to the list with as little delay as possible, the other thread should try to keep the mutex locked for as short time as possible. To indicate when a snapshot is received a semaphore is used. In this way the processing thread can do a non-busy wait until there are some data to process.

The thread that receives events is started from the main thread. The only purpose of this thread is to wait for an event to occur that signifies that a snapshot have been taken. The thread opens a pipe and waits for event1 or event2 to run and send a token through the pipe. When it has received the token, the system time will be noted, and this will be saved in the linked list. The linked list is of course mutex-locked while the new time is saved. After an event have been saved, the thread signals to the other thread with a semaphore post operation.

In the main thread, after initialization and the start of the event receiving thread, the loop containing the main algorithm starts. The pseudo-code of this loop can be seen in

Figure 7.4. The purpose of the loop is to determine the presence or absence of movement based on the data received in the other thread. The loop first blocks with a semaphore `wait` until an event has been received. Due to light changes sometimes being detected as motion, we need to make sure the snapshots taken are of a person entering. Rather than try to recognize something in the pictures that classifies a person, we found a more simple way to determine this. In empirical tests we found that usually at least 10 snapshots are taken while a person passes the camera. The first thing the program does is to check the number of snapshots taken. The constant `MIN_MOTION_NO` governs how many snapshots there needs to be taken before the program considers the motion detected originating from a person.

```
while (1)
   wait for a snapshot to be taken
   find the no of snapshots
   if (no of snapshots > MIN_MOTION_NO) then
      find the start and the end times
      find the difference
      if (difference > MAX_MOTION_DIFF) then
          find the gap between this and the last motion sequence
         if (gap > NEW_MOTION_DIFF) then
            open pipe
            send time
            close pipe
            set new last time
            remove all
         else
            remove all parts of the previous motion sequence
      else
         remove the oldest and try again
end (while)
```

Figure 7.4: Pseudo-code for the main `motion_handler` loop

Even though the number of snapshots is high enough, it might not signify motion if the snapshots are taken at different points in time. To figure out if this is the case, we find the difference between the first and the last picture, using the `time` data stored when the snapshot event was received. If the time gap is too large we will remove the oldest time from the list and test again. The time it takes to pass a camera we have considered to be maximum 5 seconds. This is of course depending on the pace a person has. The important part is that motion detected that does not originate from a person almost always will result in Motion taking less than 10 snapshots in a 5 second period. The maximum time difference is set in the constant `MAX_MOTION_DIFF`.

If the time difference between the oldest and newest is small enough, we need to test one more thing before we can establish that a person have moved past the camera. The sequence of pictures might belong to the former movement, as a camera passing might generate 20 snapshots or more. The time of the last (oldest) snapshot in the list is found and is compared to the time of the last snapshot in the previous sequence. If the difference between them is too small, the last snapshot is removed from the list as it is deemed part of the previous sequence. The time gap between this picture, which is now considered the last of the previous sequence, and the next is then calculated. This will continue until the the list is empty, or the time gap is big enough to signify that a new motion sequence has begun. The minimum time difference between two sequences of motion is set by the constant NEW_MOTION_DIFF, and we have chosen to set this to 4 seconds. This value is problematic to set, as there can be virtually no time difference between two different motions, e.g. when two persons enters next to each other. However, we need to set is at some value, and not mistake the normal difference between snapshots from the same motion, with two different persons. This means that our system has a limitation on how close after another two persons can pass the camera. When it has been determined that a person have passed the camera, the information is sent to the camera_client via a pipe. The times of this motion sequence is then removed and deallocated form the list. When this have taken place, the main loop stars over again, checking the number of snapshots taken.

To remove elements on the list the program uses the function supplied with the GSList in the glib package, the g_slist_delete_link.

The program is ended with an explicit kill command or a ctrl+c interrupt. To ensure that the list is deallocated, a termination function is provided, using the sigaction interface.

The Camera Client Program

The main purpose of the camera_client is to receive information about motion detected from the two cameras, and decide in which direction the motion have taken place (entry or exit). Motion detected by only one of the cameras should be disregarded. The source code for this program can be seen in Appendix F.7.3.

The camera_client is called with 3 arguments. The first is the default environment level, as this sensor cannot distinguish between different persons so it must assign the same level to them all. The second and third arguments are the hostname and the port where the sensor_server is listening.

The program tries to read from the pipes connected to the two motion_handler instances. When motion have been detected in front of both cameras the time between the two occurrences in found. This time is compared to the constant MAX_PASSING_TIME to make sure that they originate from the same person. The pipe from which the oldest time was received is read until the difference is small enough. In this way it is ensured that a false detection of motion that only takes place in from of one camera does not lead to wrong conclusions about the direction. We have set the MAX_PASSING_TIME to 15 seconds, but it might be possible to set it lower. It is dependent on the distance between the cameras and the pace of the person. When the direction has been established, this information is passed on the sensor_servers via a socket. It sends the direction, 'i' or 'o' for in and out, and the default security level of the person.

As the motion_handler this program also has a termination function using the sigaction interface. This one closes possible open sockets.

Auxiliary Programs and Header File

pipe2.h This header file contains the names of the pipes used in the subsystem. Since the named pipes are used to communicate between several programs, the names are stored in this header file. The header file is listed in Appendix F.7.6. The names are stored in the constants `PIPE_NAME_x` and `PIPE_CAM_x` where x is 1 or 2 for the two cameras. Furthermore, the constant `PIPE_NAME_LENGTH` gives the maximum length of the pipe names. This is needed so the variable to hold the names in the `motion_handler` can be globally declared.

`event1` and `event2` These programs are identical, except which pipe name they use to communicate. Their source code can be seen in Appendix F.7.4 and Appendix F.7.5. The programs are run each time Motion saves an image, which corresponds to each time Motion detects a large enough pixel-wise difference between the scene and the reference picture. They send the information of this fact through a pipe to the `motion_handler`. Pictures taken from each camera starts a different `event` which in turn uses different pipes to communicate with the different instances of the `motion_handler`.

`start_motion` To ensure that the two instances of `motion_handler` are initialized with the correct pipe names, both for reading and writing, they are executed from `start_motion` using `execlp`. The pipe names are taken from the header file `pipe2.h`. This program is listed in Appendix F.7.7.

7.3.4 Known Limitations in the Camera System

The camera system has several limitations that need to be kept in mind when using the system. Some of these has to do with the limitations in the cameras themselves other with Motion and the implementation of the motion detection programs. Some stems from design decisions.

The cameras do not recognize the persons. The do not do this partly because of limitations in the cameras resolution, and partly because we did not want to recognize people due to privacy concerns. A camera would have no way of recognizing a security level without recognizing a person, unlike other sensor systems like smart cards. The main limitation this imposes is that if the user leaves the computer, and the user's level is different from the default level, the system will not know that it is the user that has left. When the user returns, he will be considered a default person. Right now this can only be corrected by the use of the `swsensor` by letting a proxy person with the default level leave, and let a person with the user's level enter. The system and the reality will then be in agreement, only the user will be present in front of the computer.

Another problem has to do with the way motion detected by Motion is considered coming from a person or not. If a person enters and then exits the environment immediately thereafter the system will not discover the exit. This is because the passing of the camera will generate motion that is considered part of the entry. The same problem will arise if two persons enters closely after one another, they will be considered as one person by the system. There might be some optimizing possible in changing the values of the constants in the `motion_handler` governing this, but the problem could be solved by using sensors that could more easily determine the direction of movement or more accurately distinguish a person. e.

A related limitation is that two or more persons cannot enter at once, as the cameras only detect motion, the system does not interpret the pictures to see if there is more than one person. If additional analysis were done on the pictures this might be solved, but then a problem might arise in differentiating between two small persons and one big person. This problem will probably most easily be solved by using another type of sensor.

If a person enters too fast the camera will not be in position to take enough snapshots to trigger the condition for an camera passing to be declared by the `motion_handler`. An solution for this, besides implementing another type of sensor, could be to increase the number of snapshots taken by the camera per second, or to lower the required number of snapshots to be taken before motion is declared to originate in a person. The last solution might lead to other problems such as light changes and similar being detected as persons. In general the limitations in the camera sensor might be most effectively solved by implementing a different type of sensor, as the limitations have to do with the simplicity of the camera. Some problem might be solved by optimizing the constants in the motion detection programs, but in most cases this will probably make other problems arise.

In summary, when using the camera system, care must be taken that people enters at a reasonably slow pace, that more than one person does not enter at the same time and that there is a time difference between each camera passing. Furthermore, it must be noted that manual configuration of the subjects present most likely have to take place if the user leaves his computer and then comes back.

7.4 The Security Manager GUI

User-friendliness is very important within computer security since if the users find it troublesome so use some security feature, they will just avoid using it. As an example, the employees in a company may not bother to use the access control mechanisms if they find it annoying that they have to use many command-line tools to get an overview of which files exists, which windows are mapped or unmapped, and what the current subject levels are. The consequence may be that they store classified files in other file systems than `macfs`.

7.4.1 The GUI Functionality

A GUI program, `SecurityManagerGUI`, has therefore been developed that can perform the same tasks as the command-line tools, but it presents a more user-friendly view of the system. The `SecurityManagerGUI` program is only a presentation layer above the `file level management`, `user level management`, and `window management` sub-systems (as shown in Figure 6.1). It does therefore not add any new essential functionality to the system; in fact, it is merely a program that calls the developed command-line programs. Because mandatory access control in the stackable file system restricts who are allowed to call certain programs, the functionality provided by the `SecurityManagerGUI` depends on whether it is the super user or a regular user who runs it. For instance, only when the super user runs the program is it possible to modify the file levels. Furthermore, what is shown depends on the security policy. For instance, if the *no_read_up* policy is chosen all files with a file level greater than the clearance level will be hidden.

The GUI has been organized so that a menu is presented to the left of the window, and a panel to the right displays corresponding system information. Some screen-shots of the GUI can be seen in Appendix E. The menu items are as follows:

File Level Management is a graphical presentation of the `file level management` subsystem. It contains a text field where the user can enter a file or directory name. If a file name is entered, the `getfl` program is called and the file level of the file is retrieved. If a directory name is entered, the `listfl` program is called and the file levels of all the files in the directory is listed. A `Browse` button is also included, and it can be used to select a file or directory instead of entering it in the text field.

Since the `getfl` and `listfl` programs skips non-readable files if the *hide_non-readable_files* security policy is chosen, the `SecurityManagerGUI` program will also skip these files. For example, if all the files in a directory should be hidden, the message "No files exists." will be displayed.

The file names and corresponding file levels are listed in two columns. If the user who runs the `SecurityManagerGUI` is the super user, the second column with the file levels will be editable: when a new number is entered, the `setfl` program is called and the file level of the file in the first column is updated in the `macfs` file system. The second column is not editable when a non-privileged user runs the `SecurityManagerGUI` program, since he will always get an 'access denied' error message when he calls `setfl`.

User Level Management is a graphical presentation of the `user level management` subsystem. The appearance of the GUI depends on whether the super user started the `user level management` program or a non-privileged user. When the super user runs the program, the `listul` is called and the user names and corresponding user levels are listed in two columns, respectively. The second column is editable, and when a user level is modified the `setul` program is called so that the modifications are stored persistently.

When a non-privileged user runs the `SecurityManagerGUI`, the program will only display the user's own user level. The `getul` program with no argument is used to retrieve this level; all other user level management operations are prohibited.

Unmapped Windows displays information about all the invisible windows. It is presented in a table where the columns show the application name of the application that created the window, the corresponding process ID, the Window ID, the Window level, and the number of files that are or have been opened by the application. When a row in the table is selected, further information about the files that have been open are shown: a table in a dialog window is displayed, and it contains two columns with the file name and file level, respectively. The maximum of all the shown file levels is equal to the window level. If the *hide_non-readable_files* policy is chosen, the file names of non-readable files are replaced by 'unavailable'. The `listwl` program is used to retrieve all this information.

Mapped Windows is similar to the `Unmapped Windows`, except that only information about visible windows is shown.

Current Subject Levels The GUI displays two lines. The first line list all the subject levels for all the currently present persons and the user. The `listsl` program is used to retrieve these levels, and they are sorted in ascending order. The second line contains the current clearance level, which is retrieved by the `getcl` program.

The `SecurityManagerGUI` contains a `Reset` button in the lower right corner, and whenever it is pressed, the appropriate C programs are called so that the GUI can be updated.

If the *hide_non-readable_files* security policy is to chosen by the system administrator, it is important that the user presses `Reset` whenever the clearance level decreases. Otherwise, a person may get access to viewing which files exist if the `File Level Management` or `Mapped Management` menu item was selected prior to the person's entrance.

7.4.2 Interfacing between Java and C Programs

The `SecurityManagerGUI` is implemented in Java, but it calls the C command-line programs. It consists of 12 classes that manages the GUI, and one class that manages the interface between the C programs and the GUI classes. The latter class is denoted `Exec` (see Appendix F.8.1) and it uses an instance of the `java.lang.Runtime` class to provide an interface with the environment in which `SecurityManagerGUI` program runs. Whenever one of the C programs should be executed, the `java.lang.Runtime` instance is used to create a `java.lang.Process` instance which executes the C program in a separate subprocess. No environment variables are set before a C program is executed, so the execution will only be successful if the user has set the global `PATH` environment variable to include all the C programs.

The standard input, output, and error of the subprocess will be redirected to the parent process through a Java output stream, input stream, and error stream, respectively. The `SecurityManagerGUI` process only needs the standard output from the subprocesses, and it will therefore only use the `java.io.InputStream` class to retrieve the output. In the `Exec` class, output from the subprocesses are processed and possibly inserted into appropriate Java data structures before it is returned to the GUI classes.

7.4.3 The GUI Classes

The GUI part of `SecurityManagerGUI` is primarily implemented using `javax.swing` classes. It consists of 12 classes which are listed in Appendix F.8.2 to F.8.13. The `SecurityManagerGUI` class contains the main method, which must be given the `macfs` mount point as argument. The `SecurityManagerGUI` class creates the left-side menu and is responsible for displaying a panel corresponding to the selected menu item. Six different types of panels can be displayed (`InitPanel`, `MessagePanel`, `FileLevelPanel`, `UserLevelPanel`, `WindowPanel`, `SubjectLevelPanel`), and a `java.awt.CardLayout` object is used to switch between the different panels. All the panels are subclasses of the abstract class `BasicPanel` and they therefore have the same basic appearance. For example, they all have a `Reset` button in the lower left corner and a title at the top.

Many of the panels use a table, and the `javax.swing.JTable` is used for this purpose. The functionality required by the table depends on the panel that uses it since some of the only uses it to present information whereas others requires a editable table. Different table models are therefore used, but some common functionality could be collected in a single class, which is denoted `SimpleTableModel`. The other table model classes (`FileLevelTableModel`, `UserLevelTableModel`, and `WindowTableModel`) are subclasses of this class. All the tables that are displayed in the panels uses a `TableSorter` object, which can be used to sort the content of the table when the user clicks on one of the columns.

Chapter 8

Evaluation

The system was run and tested on two computers where the Linux distribution Fedora Core 1 was installed. We have conducted several functional tests to cover the most important aspects of the system's functionality, but it is not a comprehensive test. We do, however, feel we have covered most normal use as well as some extreme cases. We have tested the different subsystems to see if they provide the service required of them. The test cases and results can be seen in Appendix D.

Firstly, the MAC file system was tested, mostly to reveal if it could provide the same services as a normal file system while maintaining the MAC. This consisted of testing things like copying or linking files. The tests can be seen in Appendix D.1. It is important to note that while the super user has access to all possible operations, a normal user should not. The super user can unmount the file system and thereby render the MAC useless. The user should not have all these privileges.

Tests of the user and file level management were conducted to make sure only the correct programs could be executed. This entails testing things like changing the file or user levels by both a normal user and the super user. These tests can be seen in Appendix D.2 for the file level management tests and Appendix D.3 for the user level management tests.

The system can be initialized with different policies, and it needs to be tested with respect to the those. This means testing a user's ability to read or write to files at different levels. The tests conducted to verify correct operation under a specific policy can be seen in Appendix D.4. It should, in particular, be noted that when using a policy originating in the Bell-LaPadula or Biba model, a file that the user can write to is not necessarily one he can read and vice versa. The reading and writing were tested using the command line programs `cat` and `echo`. It should be noted that we have only tested the MAC, as this is in addition to the DAC included in the Linux operation system, and we assume that this works.

The above test concluded the test of the logical access control part of the system. An important part of the system is the window management, where the system keeps track of which files are open in which windows, and takes steps to unmap the windows if the physical access control determines that a person has entered the environment. The window management system tests can be seen in Appendix D.5. The tests consist both of testing auxiliary programs to list window and subject levels, as well as tests to confirm that the right windows are unmapped when a person enters. The important part of this test is to reveal if the system can keep track of the persons present and determine the correct clearance level. The system should also keep track of open files in windows and maintain the correct window level for each window.

The system is designed to function with different editors or file viewers. The extend to which various applications can be used with the system depends on the way they utilize the X Window System. We have tested the system with different editors as seen in Appendix D.6. Unfortunately, not all editors or viewers function in the same way, so full functionality cannot be guaranteed for every editor or viewer. Our tests conclude that at least `emacs`, `nedit` and `mozilla` fully integrates with our system, i.e., windows will be unmapped if they contain files that should not be viewable to an entering person.

We have created a motion detection sensor that uses two web-cameras. The ability of this subsystem to detect motion and correctly pass it on to the window management has been tested as shown in Appendix D.7. The web-camera sensor has a number of limitations as described in Section 7.3.4, but the cases where the camera sensor is expected to detect motion have been tested. It has also been an important part of the test to see if the windows are correctly unmapped when a person enters. These tests also entails testing the whole system, as every part of it must be functioning correctly for the behavior of the system to be as expected.

The GUI has been tested by both normal users and the super user. We have noted that the information provided is in accordance with the information from the command line programs. We have also tested that buttons and other GUI components work as expected.

To summarize, we have functionally tested the different parts of the system and during this executed tests that have required incremental parts of the system to be functional. The tests of the web-camera sensor has also tested the whole system. We believe that we have tested the system to the best of our ability and with a reasonable thoroughness considering the use of the system. We will therefore conclude that it functions as expected.

Chapter 9

Further Developments

9.1 Porting the System to Other Unix Versions

The choice of technologies for the implementation of the SEAC system should make it feasible to port it to another Unix system. Firstly, the FiST system contains templates for many version of Linux, Solaris, and FreeBSD. We have only tested the system using the templates for Linux-2.4, and we had to modify them slightly as described in Appendix B.1.1. If the same changes are made in other templates, the stackable file system should be portable. Furthermore, the stackable file system can (relatively) easy be extended or modified if, for instance, support for encryption or a size-changing file system turns out to be desired.

The window management part of the system is also portable since most Unix systems have a graphical user interface based on the X Window System. Our C programs should also be portable, if the required libraries are available. Finally, the `motion` detection software is platform independent.

9.2 Exportation of Classified Data

The SEAC system cannot prevent that classified data is copied from the stackable file system to another file system. A user can therefore, for example, use the command line program `cp` to copy a file to another file system managed by the operating system. It might even be possible that the user is unaware of this information flow, since the file systems are organized in a hierarchy and the user might not know where the stackable file system is mounted. One way to prevent this information flow is to mount our stackable file system on top of every file system where the non-privileged users have write access.

Another, more severe, vulnerability in the system is that classified data can be exported to external devices that are beyond the boundaries of our system. For example, a user can compose an email, copy classified data into it, and send the it out on the Internet in clear-text. Another example is the use of printers that are located outside the monitored environment.

An easy solution to these vulnerabilities is to disable access to any external devices and disconnect the computer from all networks. A more user-friendly solution could be to extend the SEAC system to utilizes the fact that many types of resources are accessed via special files in the Unix file system. For example, the `/dev` directory will (usually) contain files for accessing hard disks, modems, printers, terminals on remote computers, etc. If access control should be applied to a printer, which often is accessed via the file `/dev/lp0`, a file level should be associated with this file. This file level should depend on the physical access control that protects the printer. The security policy determines which users are allowed to use the printer. If it specifies a 'no write down' rule, a file with a high level cannot be printed by a printer with a low level. This usage will prevent that a classified file is printed in a physically unprotected printer room where an unauthorized person might be waiting for some interesting output.

We have tried to mount a stackable file system on top of the `/dev` directory, and when we used the `listfl` program we saw that all the files in the directory had the default file level, as expected. Unfortunately, the FiST system is not implemented to be to stackable on top of directories containing special files. Although it was possible to perform the mount operation, Linux did not function properly afterwards. For example, the terminal did not work any longer, and we had to restart the computer to undo the unfortunate mount.

9.3 Using other Access Control Models

Our system is based on a multilevel access control model, but the idea of combining a logical access control system with a physical one can be extended to other types of access control models.

One interesting access control model is the Chinese Wall. To use this instead it would be necessary to implement classes of conflicts of interests and company groups. A procedure for how a user selects from the companies depending on which classes of conflicts of interests he has had access to should be designed and implemented. A history function related to the user would also be needed for this. Furthermore, a way of sanitising information or determining who can decide when it has been sanitised needs to be developed. For the physical access control part, either some smart sensors that could uniquely identify the person and which classes of interests he has access to should could be used. Another approach would be to only let sanitised information be shown when a person is present.

Another interesting access control model that can be considered is role based access control. This usually entails not only traditional access operations but actions of the user as such. It could, however, be implemented by saying a person has a certain role, and then the system should only display information availably to that role.

9.4 Extending or Replacing the Sensor Subsystem

As the sensor subsystem communicates with the rest of the system by simple messages via a socket, it is very easy to replace. The only requirements is that it can deliver a direction and a security level to the `sensor_server`. Before any changes are made to the sensor subsystem, it is very important that the sockets are secured as described in Section 7.2.5; otherwise, the physical access control provided by the system can easily be circumvented by a person who knows the host name and port number where the `sensor_server` listens.

The web-camera solution has many limitations, and some advantages would therefore be gained by replacing the sensor subsystem. If the subsystem is not replaced, it could be extended to include more or different types of sensors. The information they gather could be used to enhance the cameras so the motion detection would be more precise.

If the system is replaced it would be very nice to deploy some sensors that could reveal information about the security level of the person. To do so either the person must be fully identified or the system should be able to retrieve the security information without the need for the uniquely identifying the person.

A very interesting sensor system to use would be a RFID system. All users could be equipped with an emitter, and an receiver could be placed so that it could be determined when the person carrying the emitter enters the environment. The emitter can even carry information about the level of the person, in this way making it possible to have high level persons. The system could also be part of a more comprehensive one. Many companies today have physical access control, most use magnetic cards to let employees but not casual bypassed gain access to their building. A system like this could also be used for the SEAC system, but then the magnetic card would have to be swiped every time the user enters an office. It would make more sense to use RFID as it can happen ubiquitously and be integrated into an existing system. A RFID system or one with similar capabilities can stand alone if it can be ensured that all possible persons carry a tag with a security level on. This can for example be done by requiring all visitors to carry guest tags on the premises. It could also be used in combination with a simple sensor such as the camera, so when a person is detected by the camera the default security level is overridden if the person also is detected by the RFID system.

Chapter 10

Conclusion

The main objective of this MSc project was to develop a system which demonstrates that logical and physical access control can be combined to form a smart environment. The system is developed using Linux 2.4, but it should be possible to port it to another Linux version or a Solaris or FreeBSD version. This portability is mainly due to the use of a stackable file system generated with the FiST (File System Translator) system.

The system turned out to be rather comprehensive, and to make it more manageable it was partitioned into a number of subsystems that handle different parts of the system's functionality. These subsystems are `stackable file system`, `file level management`, `user level management`, `window management`, `sensor`, and `security management gui`. Each sub-system consists of between one and six programs. Many of the corresponding processes communicate using various message passing technologies in order to provide the required system functionality.

During the test of the system, the individual subsystems were tested to the extend possible, followed by a test of the interaction between the subsystems. The test showed that the `emacs`, `nedit`, and `mozilla` editors fully integrates with our system, i.e., windows are mapped or unmapped according to the sensitivity of the displayed data and the persons who are present. The system could, however, not unmap the `gedit` editor because the communication between `gedit` and the window manager `metacity` was very unpredictable. Otherwise, the test did not disclose any errors; however, the test was not exhaustive and it is a complex system so we cannot guarantee that no further errors exists.

The security provided by the system depends on the trustworthiness of the super user. The super user must be a trusted authority that enforces a specified security policy and, for instance, sets user and file levels in accordance with this policy. It is essential that this user can be trusted since he will have access to all the files, regardless of their file levels.

10.1 Summary of Contributions

The main contribution of our work is the development of a system which extends traditional mandatory access control (MAC) so that it encompasses subjects and objects that are both logical and physical. A subject in our model can either be a process (i.e. a logical entity) or a physically present person. Likewise, an object can either be a file (i.e. a logical entity) or a window which is physically present on a computer display. Our system mediates the access to objects by subjects and it will thus combine logical and physical access control.

The MAC implementation, the detection of unauthorized persons using sensors, and the combination of the logical and physical access control are handled by different parts of the system. These parts constitute our technical contributions and can be summarized as follows:

A kernel module that is included in a stackable file system so that it can both store files and mediate the access by processes to files. Thus, it implements a simplified reference monitor. The MAC policy enforced by the system can be specified by the system administrator before system startup using security policy parameters. Furthermore, because the system is integrated with a Unix system, the access control can also include the discretionary access control (DAC) that is part of Unix systems via the rwx mode bits. The MAC will, however, take precedence over the DAC. Altogether, the implemented MAC and the Unix DAC provide mechanisms for a wide range of access control policies.

A simple movement sensor that relays information about the physical presence of a person to the computer containing classified data. The sensor is implemented using both hardware and software. The hardware is two web-cameras, and the software is a client/server system where the server resides on the computer containing classified data. The images taken by the web-cameras are used by a motion detection program, which sends a direction and level to the server whenever an person enters or leaves the environment.

The use of the client/server paradigm implies that more than one type of sensor can be used simultaneously; the server will not distinguish between the sensors since it is only concerned with receiving a level and direction. This design decision makes it easy to replace the sensor or even use many different types of sensors simultaneously, if a very fine-grained detection method is required.

A window management module which ensures that the logical access control provided by the kernel module is combined with the physical access control provided by the sensor and the graphical windowing system. More specifically, it uses library functions from the X Window System to ensure that the visibility of windows change according to the data received from the sensor and the level of the displayed information.

The system works with both text editors and pure file readers, such as Internet browsers, provided that the files opened are stored in the stackable file system. The system will also detect which files are opened by a given editor or file viewer. It can, however, not detect which files are closed by the application since this turned out to be very difficult to determine due to the use of file buffers.

10.2 Future Work

Future work on the system will primarily be related to the sensor. First of all, the communication between a `sensor_client` and the `sensor_server` must be secured, for instance by establishing an SSL connection between them. The current implementation is very insecure in this respect since anyone who knows the host address and the used port number can spoof the system by sending a level and direction of a fictitious person to the `sensor_server`. Two malicious persons can exploit this as follows: one of them enters the environment and is of course detected by the sensor. The other person will then send data to the `sensor_server` which imitate that all persons and the user leave the environment and that a person with a very high level subsequently enters the environment. The person in the environment will then have direct access to classified data, possibly after having coerced the user.

Another, more interesting, further development is related to the sensor itself. In practice, a system would have to implement a sensor that can identify and authenticate the persons so that a level can be assigned to each person. In some situations, however, privacy may be a concern if people do not want to be monitored. A solution could be a sensor that only detects which security class a given person belongs to, and not his identity. An obvious choice would be to use a radio frequency identification (RFID) system. The RFID tags would have the ability to transmit information about the person's level and nothing more. The tags could easily be used for the physical access control most companies already have and be integrated in a comprehensive system.

To summarize our work, a fully functioning prototype has been developed. The physical access control part can provide a more usable system if more advanced technologies are deployed for detecting persons. This part of the system leaves an open door for future work on our project, which hopefully will result in a system that one day can be widely used in practice.

Bibliography

[1] How Infrared motion detector components work, http://www.glolab.com/pirparts/infrared.html. 20

[2] X Window System, article in reference library available at http://www.campusprogram.com/reference/en/wikipedia/x/x_/x_window_system.html. 25

[3] The X.Org Foundation home page, http://www.x.org. 26

[4] The XFree86 Project home page, http://www.xfree86.org. 26

[5] Xlib programming manual. Available at http://tronche.com/gui/x/xlib/xlib-manual.tar.gz. 26

[6] Window Managers for X home page, http://xwinman.org/intro.html. 27

[7] Motion home page, http://motion.sourceforge.net/. 49

[8] GTK+ toolkit home page, http://www.gtk.org. 70

[9] The OpenSSL Project home page, http://www.openssl.org. 74

[10] Quickcam Express home page, http://qce-ga.sourceforge.net/. 75, 107

[11] Motion Technology , http://motion.sourceforge.net/tech/. 77

[12] FiST: Stackable File System Language and Templates home page, http://www.filesystems.org/. 107, 139

[13] Department of defence trusted computer system evaluation criteria. Avaliable at http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html, December 1985. The Orange Book. 16, 17, 35

[14] Extended Window Manager Hints. Avaliable at http://freedesktop.org/Standards/wm-spec/1.3/, 2003. Draft version 1.3. 27

[15] D. Elliot Bell and Leonard LaPadula. Secure Computer Systems: Mathematical Foundations. *MITRE Technical Report 2547*, I, March 1973. An electronic reconstruction by Len LaPadula, November 1996. 11

[16] Kevin Boone. File handling in the linux kernel, 2004. Avaliable at http://www.kevinboone.com/linux_kernel_file_0.html. 28

[17] D. Brewer and M. Nash. The chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, May 1989. 15

[18] Jens Micheal Carstensen, editor. *Image Analysis, Vision and Computer Graphics*. IMM, Technical University of Denmark, first edition, 2001. 21

[19] Dorothy E. Denning. A lattice model of secure information flow. *Communincations of the ACM*, 19(5):236–243, May 1976. 8

[20] Marius Aamodt Eriksen. Introduction to snoopy. Avaliable at http://linux.omnipotent.net/article.php?article_id=11528, 2001. 72

[21] Dieter Gollmann. *Computer Security*. Wiley, first edition, 1999. 7, 8, 10, 11, 12, 14, 15

[22] IFAD. *VDMTools – The IFAD VDM-SL Language*, 2000. 101

[23] Butler W. Lampson. Protection. In *Proc. Fith Princeton Symposium on Information Sciences and Systems*, pages 437–443. Princeton University, March 1971. Reprinted in Operating Systems Review, 8,1, January 1974, pages 18 - 24. 11

[24] Carl E. Landwehr. Formal models for computer security. *ACM Computing Serveys*, 13(3):247–278, September 1981. 10

[25] Leonard LaPadula and D. Elliot Bell. Secure computer systems: A mathematical model. *MITRE Technical Report 2547*, II, May 1973. An electronic reconstruction by Len LaPadula, November 1996. 12, 14

[26] Kenneth Lavrsen. *Motion Guide*, 1.49 edition, February 2004. Avaliable at http://www.lavrsen.dk/sources/webcam/motion_guide.htm. 77

[27] Robert Love. Kernel korner: Kernel locking techniques. Avaliable at http://www.linuxjournal.com/article.php?sid=5833, 2002. 69

[28] Daniel Manrique. X window system architecture overview howto. Avaliable at http://www.linux.org/docs/ldp/howto/XWindow-Overview-HOWTO/index.html, 2001. 27

[29] Jeffrey Oldham Mark Mitchell and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, 2001. Avaliable at http://www.advancedlinuxprogramming.com. 28, 72

[30] Vaclav Hlavac Milan Sonka and Roger Boyle. *Image Processing, Analysis and Machine Vision*. Chapman & Hall, 1993. 21, 22

[31] National Institute of Standards and Technology. Security in open systems. Avaliable at http://csrc.nist.gov/publications/nistpubs/800-7/main.html, 1994. 71

[32] David Rosenthal. *Inter-Client Communication Conventions Manual*. Sun Microsystems, Inc., version 2.0 edition, 1994. Avaliable at ftp://ftp.x.org/pub/R6.6/xc/doc/hardcopy/ICCCM/icccm.PS.gz. 27, 56

[33] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. 2nd edition, 2001. Avaliable at http://www.xml.com/ldd/chapter/book. 25, 30

[34] Peter Jay Salzman and Ori Pomerantz. *The Linux Kernel Module Programming Guide*. 2001. Avaliable at http://www.tldp.org/LDP/lkmpg/2.4/html/lkmpg.html. 30

[35] Ravi S. Sandhu. Lattice-based access contol models. *IEEE Computer*, 26(11):9–19, November 1993. 14, 15, 16

[36] Ravi S. Sandhu and Pierrangela Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994. 10

[37] Robin Sharp. *Principles of Protocol Design*. 2002. Draft second edition. 50, 51

[38] William Stallings. *Cyptography and Network Security – Principles and Practices*. Prentice Hall, third edition, 2003. International Edition. 13

[39] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001. 13, 17, 26, 28

[40] Thomas A. Wadlow. *The Process of Network Security*. Addison-Wesley, first edition, 2000. 9, 10

[41] David A. Wheeler. Program library howto. Avaliable at `http://www.tldp.org/HOWTO/Program-Library-HOWTO/`, 2003. 72

[42] Billibon Yoshimi. On sensor frameworks for pervasive systems. Avaliable at `citeseer.ist.psu.edu/285279.html`. 19

[43] E. Zadok. Stackable file systems as a security tool. Technical Report CUCS-036-99, Computer Science Department, Columbia University, December 1999. `http://www.cs.columbia.edu/~library`. 110

[44] E. Zadok. *FiST: A System for Stackable File System Code Generation*. PhD thesis, Computer Science Department, Columbia University, May 2001. `www.cs.columbia.edu/~ezk/research/thesis`. 28, 66

[45] E. Zadok and J. Nieh. Fist: A language for stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–77, June 2000. 28

Appendix A

CSP and VDM-SL Notation

This chapter is intended to give a short description of the various symbols present in the CSP and VDM-SL modeling language that may not be easily recognized. The basic mathematical operations concerning sets and logical statements have been left out.

A.1 CSP Process Expressions

There are two possibilities for defining new processes in CSP:

$$\begin{aligned} p &\stackrel{\text{def}}{=} P \\ p[i : D] &\stackrel{\text{def}}{=} P \end{aligned}$$

In the first notation, p is a process identifier that is defined by the process expression P. The second notation defines p as a parametrized process with one or more parameters, i, in some domain D. The process expression P may depend on p and i. The syntactic class of process expressions, P, is defined by the grammar:

$$\begin{aligned} P ::= {} & \mathit{STOP} \mid p \mid p[e] \mid c!e \rightarrow P \mid c?x : M \rightarrow P \mid P \talloblong P \mid (\textbf{if } b \textbf{ then } P \textbf{ else } P) \mid \\ & (\textbf{let } x : M = e \; \{,\; x : M = e\} \textbf{ in } P) \end{aligned}$$

It is assumed that $\rightarrow$ has higher precedence than $\talloblong$ so that for example $(c!e \rightarrow P \talloblong Q)$ should be read as $((c!e \rightarrow P) \talloblong Q)$.

In Table A.1, the CSP notation that has been used in the specification is listed.

The notation $\mathit{left}[i \in D]?a : M$ is used as a shorthand for the processes that accepts input of a value in the domain M on any of the channels $\mathit{left}[i]$ for $i \in D$. Furthermore, the underscore character is used as a wild-card when the input value is discarded.

A.2 VDM-SL Symbols

In Table A.2, the used data types are listed, and in Table A.3 the used operators are listed. For further details, see [22].

Expression	*Description*
$STOP$	The simplest CSP process is $STOP$, which halts without communicating.
$c!e \rightarrow P$	This process expression will initially take part in a communication event by outputting the expression e on the channel c, and then it will behave like P.
$c?x : M \rightarrow P$	This process expression will initially take part in a communication event by inputting any value of type M to the variable x from the channel c, and then it will behave like P.
$P \mathbin{[\!]} Q$	A process which behaves like either P or Q. It is the environment that determines which process expression is chosen, depending on which events are received.
$(\textbf{if } b \textbf{ then } P \textbf{ else } Q)$	If the boolean expression b is true, the process behaves like P; otherwise, it behaves like Q.
$(\textbf{let } x1 : M1 = e1 \; \{, \; x2 : M2 = e2\} \textbf{ in } P)$	The let process expression is used to assign the expression $e1$ of type $M1$ to the variable $x1$ and optionally assign the expression $e2$ of type $M2$ to the variable $x2$, etc. After the assignment(s), the behavior of the process is as specified by P.

Table A.1: CSP notation for processes and process expressions.

Data Type	*Description*
$\mathbb{B}$	The boolean values $\{\mathsf{true}, \mathsf{false}\}$.
$\mathbb{N}_0$	The non-negative integers: $\{0, 1, 2, \ldots\}$.
$\mathbb{N}_1$	The positive integers: $\{1, 2, \ldots\}$.
char	An arbitrary character.
token	A countably infinite set of distinct values whose structure is insignificant in the specification.
$S\text{-}\mathsf{set}$	A set with elements of type S.
S^*	A possibly empty sequence of values of type S.
$S \xrightarrow{m} T$	A mapping from elements of type S to elements of type T. The value of type S is denoted the *key* and the value of type T, which is associated with it, is denoted the *information value*.
$S \times T \times \ldots$	The product type whose values are denoted tuples. A tuple is a fixed length list where the first element is of type S, the second of type T, etc.

Table A.2: VDM-SL data types.

Operator	*Name*	*Description*
len l	Length	Yields the number of elements in the sequence l.
hd l	Head	Yields the first element in the non-empty sequence l.
tl l	Tail	Yields the subsequence of the non-empty sequence l where the first element is removed.
$l1 \frown l2$	Concatenation	The sequence $l1$ is concatenated with the sequence $l2$, i.e., it yields a sequence that consists of the elements in $l1$ followed by those in $l2$, in order.
dom m	Domain	Yields the domain (i.e., the set of keys) of the map m.
rng m	Range	Yields the range (i.e., the set of information values) of the map m.
$m1 \dagger m2$	Override	The map $m1$ is overridden by the map $m2$, i.e., it yields a map combined by $m1$ and $m2$ where the elements of dom $m1$ are mapped as by $m1$, and the elements of dom $m2$ are mapped as by $m2$. Any common elements are mapped as by $m2$.
$m(d)$	Map apply	Yields the information value whose key is d. d must be in the domain of the map m.

Table A.3: VDM-SL operators.

Appendix B

Installation Guide

The Sensor Enhanced Access Control (SEAC) system is packed in the `seac.zip` file. When this file is unzipped the following directory structure is created:

```
                              seac
                               |
    -------------------------------------------------------
    |        |        |        |        |        |        |
    |        |        |        |        |        |        |
    |        |        |        |        |        |        |
templates   out     macfs     gui     test     webcam   report
    |        |
    |        |
Linux-2.4   Linux-2.4
             |
             |
            macfs
```

`templates/Linux-2.4` contains the FiST stackable templates for Linux 2.4. These files will only be needed if the stackable file system should be modified (see Appendix B.1.1).

`out/Linux-2.4/macfs` contains all the source code for the SEAC system, except the web-camera sensor part. This code includes the stackable file system code for the `macfs`, the code for the `window management` subsystem, and scripts for startup and shutdown. The stackable file system code can only be on Linux 2.4, and new code must be generated if another Unix version is used.

`macfs` contains the FiST input file, `macfs.fist`, and all the C code for the stackable file system and `window management` part of the system. When the `fistgen` is run with `macfs.fist` as argument, it will use the templates in `templates/Linux-2.4` and the code in `macfs` to produce file system code, which are stored in `out/Linux-2.4/macfs`. To avoid having to set the `PATH` environment variable to include more directories than necessary, the `Makefile` we have created will copy the remaining code from `macfs` to `out/Linux-2.4/macfs` and create the binaries and shared libraries in this output directory as well.

`gui` contains the Java source code for the `security management gui` and the file `seac.jar` which contains the compiled class files.

`test` contains the scripts that were used during the test of the system. Furthermore, it contains a sample security policy file `policy.txt`.

`webcam` contains the source code to the motion detection programs, as well as the configuration files for Motion.

`report` contains the latex files for this thesis as well as versions in pdf-format and ps-format.

The SEAC system will possibly have to be installed on two different computers where one stores the classified files and is used by a regular user, and the other runs the motion detection program and has web-cameras attached. The installation guide has therefore been divided in two sections, reflecting this deployment of the system.

B.1 Installation of the Access Control Part

If no changes to the stackable file system is required and the Unix version used is Linux 2.4, the stackable file system code that we have generated using `fistgen` can be used. The following steps should then be performed when installing the system.

1. **Set macro definitions (Optional)**

   A number of macros in the source code can be set at compile time using the -D preprocessor option. The kernel space part of the code contains two macros: the `HASH_TABLE_SIZE` denotes the size of the hashtable (the default is 10), and `DEBUG` can be used to turn on or off debug output to the system log (the default is off). These macros should be set in the `SEAC_KERNEL_DEFINITIONS` variable in the makefile `out/Linux-2.4/ macfs/Makefile`.

   The user space part of the code contains the following macros that can be set: `MOUNT_POINT` denotes the mount point for `macfs` (the default is `/mnt/macfs`); `DEBUG` can be used to turn on or off debug output to standard error (the default is 0); `libX11_PATH` is the path to the X11 library (the default is `/usr/X11R6/lib/libX11.so`); `libc_PATH` is the path to the c library (the default is `/lib/tls/libc.so.6`); and `SHELL_PATH` is the path to the valid login shells (the default is `/etc/shells`, cf. the man page 'shells', section 5). These macros should be set in the `SEAC_DEFINITIONS` variable in the makefile `out/Linux-2.4/ macfs/Makefile`.

2. **Compile the code**
   The code is compiled by changing to the `SEAC` directory and running

   ```
   make seac
   ```

   This will compile the code in the `out/Linux-2.4/macfs` and `gui` directories.

3. **Update the `PATH` variable (Optional)**
   The `PATH` environment variable should be set so that it includes all the command line programs. If the `seac.zip` file is unzipped in the user's home directory, this can be done with the command:

```
export PATH=$PATH:$HOME/seac/out/Linux-2.4/macfs:$HOME/seac/test:
```

Additionally, the super user should set the `PATH` variable so that it includes the path to the programs for loading (`insmod`) and unloading (`rmmod`) kernel modules; this is typically `/sbin`. Instead of setting the `PATH` variables each time the operating system is started, it can be set automatically if the `export` command is added to a startup script, such as `.bashrc` (if the `bash` shell is used).

B.1.1 Changes made to the Stackable File System Templates

If the the stackable file system should be modified as part of a further development of our system, or if it should be ported to another Unix version than Linux 2.4, some changes will have to be made to the templates. First of all, the latest version of the FiST system should be downloaded from the FiST home page[12]. Depending on the functionality that should be provided by the new file system, the following should be inserted in the templates. In the file `templates/Linux-2.4/inode.c`, the tags `FIST_OP_CREATE_POSTCALL`, `FIST_OP_UNLINK_POSTCALL`, `FIST_OP_MKDIR_POSTCALL`, and `FIST_OP_RMDIR_POSTCALL` must be inserted near the end of the functions `wrapfs_create()`, `wrapfs_unlink()`, `wrapfs_mkdir()`, and `wrapfs_rmdir()`, respectively. Furthermore, the statements

```
err = inode_permission(inode->i_ino, mask);
if(err) goto out;
```

must be inserted at the beginning of the function `wrapfs_permission()`. In the file `templates/Linux-2.4/file.c`, the statement

```
file_open_intercepted(inode->i_ino, inode->i_mode, file->f_dentry->d_name.name)
```

must be inserted at the beginning of the function `wrapfs_open()`.

Finally, the file `templates/Linux-2.4/Makefile` should be changed. We will not describe our changes in details here; instead, we recommend looking in our makefile and making corresponding changes in the new make file where required. If the Linux 2.4 templates are used, our makefile can just be used. The main changes that we have made is that the variables `SEAC_KERNEL_DEFINITIONS` and `SEAC_DEFINITIONS` are added, and the variables `CFLAGS` and `UCFLAGS` are changed so that they make use of these variables.

B.2 Installation of the Web-cameras and Motion Detection Programs

Installation of the Web-cameras

The web-cameras should be placed next to the area that needs to be monitored. They should be placed on a stable surface or fixed to the ceiling or wall to prevent shaking. The cameras should be placed approximately 30-50 cm apart. After being connected to the computer, the drivers for the cameras should be installed. Use the driver supplied with the cameras or a general Linux driver, such as the `qce-usb` driver[10], and follow the instructions given. The cameras will most likely be installed as `/dev/video0` and `/dev/video1`, but note the exact location for your specific installation.

Installation of Motion

The Motion program, version 3.0.7-1, has been provided in an archive file on the attached CD. This version can be used, or a newer version can be downloaded from http://motion.sourceforge.net. Extract the files and follow the instructions given in the `INSTALL` file. When Motion has been installed, the configuration files should be set up as explained in Section 7.3.2. The configuration files are stored in the `webcam` directory. If using these files, you will have to change the *target_dir* where the snapshots are saved to a directory valid on your system. The *onsave* needs to be the correct path to the `event1` and `event2` programs, most likely ∼`/seac/webcam/event1` and ∼`/seac/webcam/event2`. Also make sure that the *vidoedevice* option is correct.

If using a newer version of Motion be aware that the configurations files have a slightly different layout and way of handling two cameras. Be sure to read the documentation to set up the program.

Installation of Motion Detection Programs

1. **Update the pipe paths (Optional)**
   The file `pipe.h` (Appendix F.7.6) specifies where the pipes used for communication will be created. This can be changed if needed.

2. **Compile the code**
   Enter the `webcam` directory and run `make`.

3. **Update the `PATH` variable**
   The `PATH` environment variable should be set so that includes the path to the motion detection programs.

   ```
   export PATH=$PATH:$HOME/seac/webcam:
   ```

   The path to the `webcam` directory might be different.

Appendix C

User's Guide

In this appendix, a user's guide for both the super user and a non-privileged user is provided. First of all, it is described how the core of the system consisting of the stackable file system and window management part is started up and shutdown (see Appendix C.1). Since the motion detection program will possibly be running on a different machine than the one containing the classified files, the startup and shutdown of this part of the system is described in a separate section (see Appendix C.2).

The SEAC system consists of many command-line programs, and their functionality were described in Chapter 6. A regular user will, however, not be interested in these details, but rather in how the programs should be used, so a user's guide is provided in Appendix C.3. The users can of course also use the Security Manager GUI instead of the command line programs (see Appendix C.4).

C.1 System Startup and Shutdown

In a secure deployment of the system, a regular user should not be able to login as the super user since an unauthorized person then can use physical coercion to circumvent all the mandatory access control provided by the system. Furthermore, the principal who has super user privileges should not present near the environment since he then is susceptible to kidnapping. If an unauthorized person in some way is able to login as the super user, he can unmount the stackable file system and obtain unrestricted access to all the classified files in the underlying files system. To avoid that the principal who has super user privileges is present, the startup and shutdown of the system should preferably be made part of the Linux startup and shutdown procedure. This can be done by inserting the required commands in scripts that are run automatically at startup and shutdown.

C.1.1 Guide to the Super User

Initializing the system will on most systems require root privileges since a non-privileged user is not allowed to mount file systems and load kernel modules. As motivated in Section 7.1.2, the mount point is specified in the header file `mount_point.h`, and if a non-default mount point should be used, the code must be recompiled after the modification.

The stackable file system comes in the form of a loadable kernel module, `macfs`, and the first task for the super user is to install it in the running kernel. This is done with the command:

```
insmod macfs.o
```

provided that the file `macfs.o` is in the current working directory (after the installation, this file is in `/out/Linux-2.4/macfs`).

The next step is to mount the file system. The most secure mount style is an overlay mount where the mount point and the underlying file system have the same absolute path[43]. User processes must pass through `macfs` before they can access files, and the lower file system is therefore hidden from user processes. `macfs` should be mounted with the command:

```
mount -t macfs -o dir=MOUNT_POINT MOUNT_POINT MOUNT_POINT
```

Because the underlying file system can be used before the `macfs` is mounted on it, there can possibly exists some files where no associated file level exists. In this case, the lowest level will be returned by the programs `listsl` and `getfl`, ensuring that the system works even if files already exist below the stackable file system.

The insecure mount style is a regular mount where two paths (LOWERDIR and MOUNT_POINT) are specified, and the mount command is used as follows:

```
mount -t macfs -o dir=LOWERDIR LOWERDIR MOUNT_POINT
```

This mount style should only be used when testing the system since it makes all the security mechanisms provided by the stackable file system worthless: a unauthorized person will have direct access to all the files in `LOWERDIR`, regardless of their file level, and he can use any editor he wishes to open the files, since the system will not unmap a window unless it has opened files from the stackable file system.

The command `lsmod` lists information about all loaded modules and should include the `macfs` module. If the command `mount` is used without any command line arguments, it should include the `macfs` file system. The stackable file system uses the system log for auditing, and its content can be seen in the file `/var/log/messages` or using a special application such as `redhat-logviewer` (see the menu item system log).

After the mount, the stackable file system must be initialized by running the `seac_init` program:

`seac_init POLICY_FILE FILE_LEVELS_FILE USER_LEVELS_FILE` - initialize file system
: The super user must use this program to initialize the file system. All the command line arguments are files. The file `POLICY_FILE` must contain the security policy that should be used by the system. The format must be as follows:

```
Hide non-readable file names: x
No read up: x
No read down: x
No write down: x
No write up: x
Permit lower level login: x
```

where x is either y or n, indicating that the policy is chosen or not, respectively.

The file `FILE_LEVELS_FILE` must either be an empty file or it must store inodes and corresponding file levels in the binary format that is used when the `seac_destroy` stores the system state. Similarly, the file `USER_LEVELS_FILE` must be empty or store user IDs and corresponding user levels.

To avoid that non-privileged users corrupt the files, they should be protected using the security mechanisms provided by Unix so that only the super user can modify the files. The directory containing them should also be protected so that the files are not simply removed; this can be done using sticky bits or by forbidding write access to the directory. The non-privileged user should have read access to the `POLICY_FILE` so that they can see which security policy is enforced by the system.

The program will fail if any of the files does not exists or the format in a file is invalid. It will also fail if it is not the super user that runs the program, or if it has already been run previously.

If the system should make windows visible or invisible according to the detected persons, the `visibility_manager`, `file_open_monitor`, and `sensor_server` programs must subsequently be started. However, this step is optional since the stackable file system part of the system can be used independently of the window management and sensor part, if only logical mandatory access control is required.

When the system should be shutdown, the `seac_destroy` program must be run:

`seac_destroy` - terminate system
: stores all the file and user levels, terminates the stackable file system so that it no longer can be used and kills the `visibility_manager` and `file_open_monitor` process.

If the `sensor_server` was started previously, it is terminated by sending it a negative file level. Furthermore, the stackable file system must be unmounted and the kernel module must be unloaded. This is done as follows:

```
swsensor -1 i
umount MOUNT_POINT
rmmod macfs
```

To easing the job of the system administrator, the script `startup.sh` can be used to startup the system (see Appendix F.9.2), and the script `shutdown.sh` can be used to shutdown the system (see Appendix F.9.3). Furthermore, the script `reset.sh` (see Appendix F.9.1) should be called before the system is used the first time: it will create the mount point if it does not exists and the required empty files for the file levels and user levels. If it is called subsequently, it will delete all the files created in the stackable file system and delete all file levels and user levels; it is of course only the intention that it should be used for testing purpose.

C.1.2 Guide to a Non-Privileged User

A user must first log in to the system using the `initcl` program:

`initcl` [LOGIN_LEVEL] - initialize clearance level
: Initializes the clearance level to the user level of the user who runs this program. If the security policy permits it, the user can use the command line argument `LOGIN_LEVEL` to specify a level below the user's level that the clearance level should be initialized to. Whether this is permitted can be read in the security policy file, which is maintained by the super user.

The system will only detect created windows if the `LD_PRELOAD` environment variables is set to the directory containing the shared libraries `x_create_window_interceptor.so` and `backup_interceptor.so`. If only logical access control should be provided by the system, this variable should not be set. If the shared libraries are stored in the directory `~/seac/macfs`, this can be done with the command

```
export LD_PRELOAD=~/seac/out/Linux-2.4/macfs/x_create_window_interceptor.so:
~/seac/out/Linux-2.4/macfs/backup_interceptor.so
```

in a terminal, and then subsequently starting the editor from that terminal. If the stackable file system is mounted on `/mnt/macfs` the user can open an editor with the command

```
cd /mnt/macfs
emacs file.txt
```

or, alternatively:

```
emacs /mnt/macfs/file.txt
```

If the system should be notified about all the created windows, and not just those started from a terminal, the `LD_PRELOAD` variable can be set in a startup script such as `.bashrc` (if the `bash` shell is used). The system will then be notified about all the created windows, and not just those that are used to edit classified files. For testing purposes, a script (such as `test/usertest.sh`) can be used to set `LD_PRELOAD`, call `initcl` and start one or more editors. By setting `LD_PRELOAD` in a script, it is avoided that all `rename`, `XCreateWindow` and `XCreateSimpleWindow` calls made from the terminal are intercepted by the shared libraries. This approach may be preferred if many GUI programs are started from the terminal and only a few of them will be used to open classified files; the user can then easily determine which window the SEAC system should be notified about.

C.2 The Web-Camera Sensor

Startup

The camera sensor can be started by any user, but for the system to be effective it should be started by root. This is to prevent unauthorized termination of the sensors.

The system is started using the following the steps:

1. Run Motion with the command `motion -D`.
2. Run `start_motion`. This starts the two `motion_handler` instances.

3. Run `camera_client ENVIRONMENT-LEVEL HOST-NAME PORT`. `ENVIRONMENT-LEVEL` is the level a default person detected by the camera should be assigned. `HOST-NAME` and `PORT` is the host name and the port where the `server_client` is listening.

Alternatively, the camera system can be started with the script `startcam.sh` found in the `webcam` directory. This executes all the above steps, but the variables for the `camera_client` must be changed to reflect the actual system.

Shutdown

If the system should be terminated `ps -ef | grep motion` and `ps -ef | grep camera` can be used to obtain the process IDs of the programs started (`motion`, `camera_client` and two instances of `motion_handler`), and they can be terminated using the `kill` command.

Since the *snapshot_overwrite* option for Motion does not function as expected, Motion will take a lot of snapshots during operation and save them in the directory set in *target_dir*. These images should be removed at regular intervals.

C.3 The Command Line Programs

Several command line programs have been developed, as described in this section. Where not specified otherwise, they can be used by all users.

C.3.1 File Level Management

The `file level management` sub-system provides three command line programs. They all take a file or directory as command line argument, which must exist in the stackable file system; otherwise, an error message is returned.

`setfl FILE FILE_LEVEL` - set file level
: Sets the file level associated with `FILE` to `FILE_LEVEL`. If the `visibility_manager` is running and a user has the file open in an editor, the visibility of the editor's window will change if the new level implies that the window should be mapped or unmapped. Only the super user can use this program.

`getfl FILE` - get file level
: Retrieves the file level associated with `FILE`. If the *hide_non-readable_files* security policy is chosen and the file is not readable according to the security policy, no file level can be retrieved and an error message will indicate that the file does not exist.

`listfl [DIR]` - list file levels
: Lists the file name and corresponding file level for every file in `DIR`. If no directory is specified, the file level for the current working directory will be printed. If the *hide_non-readable_files* security policy is chosen, the non-readable files will be skipped in this listing.

C.3.2 User Level Management

The `user level management` sub-system provides three command line programs:

`setul USER_NAME USER_LEVEL` - set user level
: sets the user level for the user with the user name `USER_NAME` to `USER_LEVEL`. The level can be any positive integer. Only the super user is allowed to change the user level.

`getul [USER_NAME]` - get user level
: Retrieves the user level for a given user. A user can retrieve his own level, but not the level of another user.

`listul` - list user levels
: lists the user name and corresponding user level for every user in the system. Only the super user is allowed to list the user levels.

C.3.3 Window Management

The `window management` sub-system provides three command line programs that require super user privileges. They should all run as a demon processes.

`visibility_manager POLICY_FILE`
: Starts the visibility manager which is responsible for changing the visibility of windows whenever an unauthorized person enters the environment. The security policy that should be enforced is read from the text file `POLICY_FILE`.

`file_open_monitor`
: Starts the file open monitor. Whenever a regular file or a link to a file is opened, it will notify the `visibility_manager` about this.

`sensor_server[PORT]`
: Starts the sensor server. If a non-default port should be used, it must be specified as the command line argument `PORT`. The default port number is 33333.

The `window management` sub-system provides three command line programs that can be used by all users:

`listwl` - list window levels
: List all the window levels for all the current windows along with other status information.

`listsl` - list subject levels
: List the subject levels for every subject in the environment.

`getcl` - get clearance level
: Retrieves the current clearance level.

C.3.4 Sensors

`swsensor LEVEL DIRECTION [HOST] [PORT]` - software sensor client

The software sensor client can be used to simulate that a person enters or leaves an environment. The first command line argument (`LEVEL`) is the environment level of a person. If the level is negative, it will shut down the `sensor_server`. The second (`DIRECTION`) is the direction of the person which must be either 'i' or 'o', indicating that the person enters or leaves the environment, respectively. The third argument (`HOST`) is the host where the `sensor_server` runs. If no host is specified the default host `localhost` will be used, and the `sensor_client` can then only be used on the host where the `sensor_server` runs. Finally, the fourth argument (`PORT`) is the port number used by the `sensor_server`. If no port is specified, the default port 33333 will be used.

C.4 The Security Manager GUI

Many of the developed command line programs can be used indirectly via the Security Manager GUI. The GUI contains five menu items (as can be seen in Appendix E), and they can be used to run the following programs: **getfl**, **setfl**, **listfl**, **getul**, **setul**, **listul**, **listwl**, **getcl**, **listsl**. If the mount point is **/mnt/macfs**, the GUI can be started as follows:

```
java -jar seacGUI.jar  /mnt/macfs &
```

If a command line program can only be used by the super user, it cannot be used in the GUI either. It will, for instance, only be possible to modify user or file levels and see the levels of all the users, when the super user runs the GUI.

Appendix D

Testing

We have tested the different part of the system for various cases on the Linux distribution Fedora Core 1. The tests are managed and executed a bit differently depending on which subsystem we are testing. Table D.1 shows the users and files that constitute the default state of the system before any tests are started. The files all contain a text string, and are readable and write-able by all users with respect to the Linux DAC permissions. No persons are assumed to be present when the tests start. In Table D.2, a set of policies are shown, in each test or case a specific one will be used. The policies are the ones from the security policy file loaded when the system is started.

user	level
`alice`	3
`bob`	20
`root`	superuser

file	level
`a.txt`	1
`b.txt`	5
`c.txt`	20

Table D.1: Default Users and Files

Policy	read up	read down	write up	write down	file names	lower level login
A	no	yes	yes	no	hidden	yes
B	yes	no	no	yes	hidden	no
C	no	yes	yes	no	shown	no

Table D.2: Policies used in testing

D.1 Stackable File System Test

The stackable file system is tested to disclose whether normal file operations can take place while still maintaining the properties of a MAC file system. The tests made by the super user can be seen in Table D.3. Furthermore, movement and copying was tested with an unprivileged user; these tests can be seen in Table D.4.

Command	Test Case	Expected Result	OK
`link c.txt c2.txt`	Link a file to another	`c2.txt` has the same level as `c.txt`	√
`unlink c2.txt`	Unlink a file	`c2.txt` is removed	√
`mkdir test`	Create a directory	Directory `test` created	√
`mv a.txt test/a.txt`	Move a file	`a.txt` is in the directory `test` with its level intact	√
`cp b.txt test/b2.txt`	Copy a file	`b2.txt` is created and assigned the clearance level	√
`rm b2.txt`	Remove a file	`b2.txt` is removed	√
`rmdir test`	Remove a directory	`test` is removed	√

Table D.3: Test of the Stackable File System by the Super User

Command	Test Case	Expected Result	OK
`mv a.txt /home/alice/a.txt`	Move a file with a lower level than the user outside the file system	Not permitted	√
`mv c.txt /home/alice/c.txt`	Move a file with a higher level than the user outside the file system	Not permitted	√
`cp a.txt /home/alice/a.txt`	Copy a file with a lower level than the user to outside the file system	Permitted	√
`cp c.txt /home/alice/c.txt`	Copy a file with a higher level than the user to outside the file system	Not permitted	√
`cp c.txt c2.txt`	Copy a file with a higher level than the user	Not permitted	√
`cp a.txt a2.txt`	Copy a file with a lower level than the user	Permitted	√

Table D.4: Test of the Stackable File System by a User

D.2 File Level Management Test

The file level management programs `setfl`, `getfl`, `listfl` are tested both with a normal user and super used logged in, as well as with different security policies. The test cases can be seen in Table D.5.

User	Policy	Command	Test Case	Expected Output	OK
`alice`	**A**	`setfl a.txt 6`	Change a file level	Not permitted	√
`alice`	**A**	`listfl`	List the files	List shows only the file `a.txt`	√
`alice`	**A**	`getfl a.txt`	Get the level of a file below the users level	The level of `a.txt` is shown (3)	√
`alice`	**A**	`getfl c.txt`	Get the level of a file above the users level	Not permitted	√
`alice`	**C**	`listfl`	List the files	List shows all filenames	√
`root`	**A**	`listfl`	List the files	List shows all filenames	√
`root`	**A**	`setfl a.txt 6`	Set a file to a new level	`a.txt` is assigned level 6	√
`root`	**A**	`getfl c.txt`	Get a file's level	The level of `c.txt` is shown (20)	√

Table D.5: Test of the File Level Management

D.3 User Level Management Test

The user level management programs `setul`, `getul`, `listul` are tested both with a normal user and super used logged in while using policy **A**. The test cases can be seen in Table D.6.

User	Command	Test Case	Expected Result	OK
`alice`	`setul alice 5`	Change users own level	Not permitted	√
`alice`	`setul bob 21`	Change another users level	Not permitted	√
`alice`	`getul`	Get the users own level	`alice`'s level is shown (3)	√
`alice`	`getul bob`	Get the level of another users	Not permitted	√
`alice`	`listul`	List all user levels	Not permitted	√
`root`	`listul`	List all user levels	A list of the user levels is shown	√
`root`	`getul bob`	Get the user level of a user	`bob`'s user level is shown (20)	√
`root`	`setul alice 5`	Change a users level	`alice`'s user level is 5	√

Table D.6: Test of User Level Management

D.4 Mandatory Access Control Test

The Mandatory Access Control is tested by attempting to read and write different files under different policies, all with the user `alice` (level 3). The tests can be seen in Table D.7.

Command	Policy	Test Case	Expected Result	OK
`cat a.txt`	**A**	Read a file below the users level	Permitted	√
`cat b.txt`	**A**	Read a file above the users level	Not permitted	√
`echo x > a.txt 123`	**A**	Write to a file below the users level	Not permitted	√
`echo x > b.txt 123`	**A**	Write to a file above the users level	Permitted	√
`touch d.txt`	**A**	Create a file	`d.txt` created with level 3	√
`rm d.txt`	**A**	Remove a file	`d.txt` is removed	√
`cat a.txt`	**B**	Read a file below the users level	Not permitted	√
`cat b.txt`	**B**	Read a file above the users level	Permitted	√
`echo x > a.txt 123`	**B**	Write to a file below the users level	Permitted	√
`echo x > b.txt 123`	**B**	Write to a file above the users level	Not permitted	√
`echo x > a.txt 123`	**A**	Write to a file below the users level while logged in at a lower level	Permitted	√

Table D.7: Test of the Mandatory Access Control

D.5 Window Management Test

The window management subsystem needs to be tested in two ways. The programs `getcl`, `listwl` and `listsl` need to be tested, and these tests can be seen in Table D.8. The mapping and unmapping of windows depending on the clearance level need to be tested as well, this is done using the `swsensor`. These tests can be seen in Table D.9. All tests were conducted using policy **A**.

In this test is is assumed that the `swsensor` functions as expected, it is assumed that it can send the correct value and direction via the socket communication. This is not an unrealistic requirement, as it is a very simple program, and the correct transmission of data was tested during development. It is necessary to use this to test the window management system, as changes in the environment level needs to be tested.

Command	Test Case	Expected Result	OK
`getcl`	Only the user in the room	Returns the user's level	$\surd$
`getcl`	A person in the room	Returns the lowest of the person's and user's levels	$\surd$
`listsl`	Only the user in the room	Returns the user's level	$\surd$
`listsl`	A person in the room	Lists the user's and the person's level	$\surd$
`listwl`	No open windows	Notes that no windows are open	$\surd$
`listwl`	Files in a mapped window	Lists the level of the window as the maximum of the open files	$\surd$
`listwl`	Files in an unmapped window	Lists the level of the window as the maximum of the files open, and lists the high level file names as 'unavailable'	$\surd$
`listwl`	Multiple windows, both mapped and unmapped	Lists the levels of the windows as the maximum of the files open in that window, and the high level file names as 'unavailable'	$\surd$

Table D.8: Test of Window Management Tools

D.6 Editor Test

We have tested the system with different editors and viewers. In Table D.10 it can be seen which cases were tested with each, and in Table D.11 the results for each editor or viewer can be seen. The editors were the standard versions included in Fedora.

D.7 Web-camera Sensor Test

At last we test the web-camera sensor to see if the persons passing by are registered correctly and windows are mapped or unmapped correspondingly. The tests can be seen in Table D.12.

Event	Test Case	Expected Output	OK
A person enters	No open files	Nothing	$\surd$
A person enters	Open files with higher level than the person	The window(s) containing the files is unmapped	$\surd$
A person enters	Open files with lower level than the person	Nothing	$\surd$
A person enters	Open files with both higher and lower level than the person	All windows containing files with higher level than the person are unmapped	$\surd$
The person leaves	For all of the above	The system is reverted to its previous state	$\surd$
Two persons enters after one another	Files open with higher level than one of the persons	Windows containing the files are unmapped	$\surd$
One person leaves, one stays	Clearance level rises above some of the files	Windows containing the files becomes mapped	$\surd$
One person leaves, one stays	The clearance level stays the same	Nothing	$\surd$

Table D.9: Test of the Window Management

Test Case	Description	Expected Result
A	Open a file with write and read permitted to the user	Permitted
B	Open file with read only permitted to the user	Should only open in a viewer or in a non-editable mode
C	Open a file with read not permitted to user	Not permitted
D	Create a file	The file should be assigned the clearance level
E	Clearance level is lowered due to a person	Window should disappear
F	Clearance level is raised due to a person leaving	Window should be remapped
G	Open a backup of a file with read not permitted to the user	Not permitted

Table D.10: Editor and Viewer Test Cases

Editor/Viewer	case A	case B	case C	case D	case E	case F	case G
emacs	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
gedit	$\surd$	$\surd$	$\surd$	$\surd$	$\div$	?	$\surd$
nedit	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
mozilla	$\surd$	$\surd$	$\surd$	N/A	$\surd$	$\surd$	$\surd$

Table D.11: Test of Editors and Viewers

Event	Test Case	Expected Output	OK
A person enters	No open files	Nothing	√
A person enters	Open files with higher level than the person	The window(s) containing the files is unmapped	√
A person enters	Open files with lower level than the person	Nothing	√
A person enters	Open files with both higher and lower level than the person	All windows containing files with higher level than the person are unmapped	√
The person leaves	For all of the above	The system is reverted to its previous state	√
Two persons enters close to another	The time difference between two persons is low	`listsl` should list both persons	√
A person enters	The person moves at a fast (not running) pace	The web-cameras detect the person	√

Table D.12: Test of the Camera Sensor

Appendix E

GUI Screen-shots

A few screen-shots have been taken of the `SecurityManagerGUI`, and they are shown in Figure E.1 to E.8. The Bell-LaPadula access control model was used during this test. Furthermore, the non-readable files were hidden i.e., the name of a file where the file level is greater than the clearance level is hidden by the file system and `visibility_manager` process. The content of the security policy file was therefore as follows:

```
Hide non-readable files: y
No read up: y
No read down: n
No write down: n
No write up: n
Permit lower level login: y
```

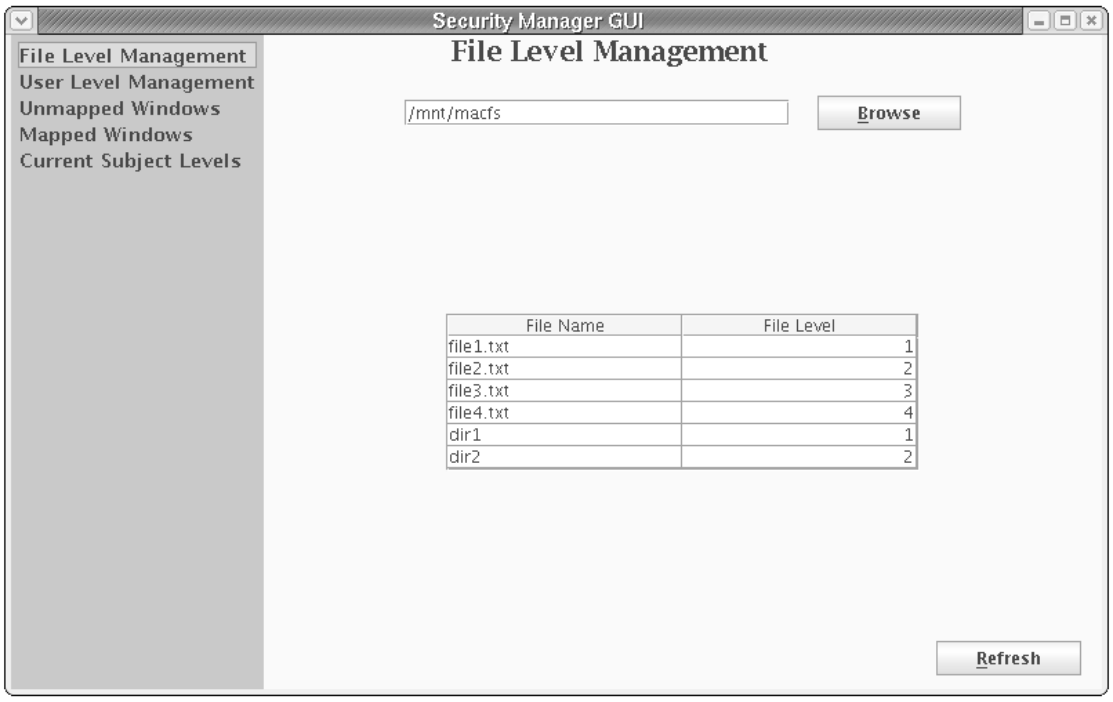
When the file level management menu item is selected (see Figure E.1), the user can select to view the file level of an individual file by writing its name in the text field or clicking on the `Browse` button and selecting the file (see Figure E.2). If a directory is selected, the file levels for all the files and directories in the selected directory is listed. When the super user runs the `SecurityManagerGUI` application, the file levels in the `File Level` column can be modified to any positive integer. When a non-privileged user runs the application, the file levels are not editable.

The user who initially logged in by calling the `initcl` program was `s973732`. This user has clearance level 10, and when the `User Level Management` menu item is selected, only this user level is shown (see Figure E.4). When the super user runs the application, he will have access to all the user levels for all the normal users in the system (see Figure E.3).

Two persons with levels 5 and 9 have been detected in the environment and the clearance level is therefore decreased from 10 to 5 (see Figure E.8). (The detection of persons can be simulated with the `sensor_client` program.)

Only one file (`/mnt/macfs/dir1/file6.txt`) has a file level that is greater than the clearance level. It is open in an `emacs` editor, which is invisible (see Figure E.5). The file name is replaced by 'unavailable' in the dialog window that displays information about the open files in the invisible window (see Figure E.6).

Four editors and a single Internet browser have files open, but none of the files contain information that should be hidden, so the windows are visible (see Figure E.7). An overview of all the information maintained about the visible and invisible windows can be obtained using the `listwl` program; its output after this test is listed in Figure 7.2.



/mnt/macfs

File Name	File Level
file1.txt	1
file2.txt	2
file3.txt	3
file4.txt	4
dir1	1
dir2	2

Figure E.1: The `File Level Management` menu item is selected, and all the file levels of the files and directories in `/mnt/macfs` are listed.

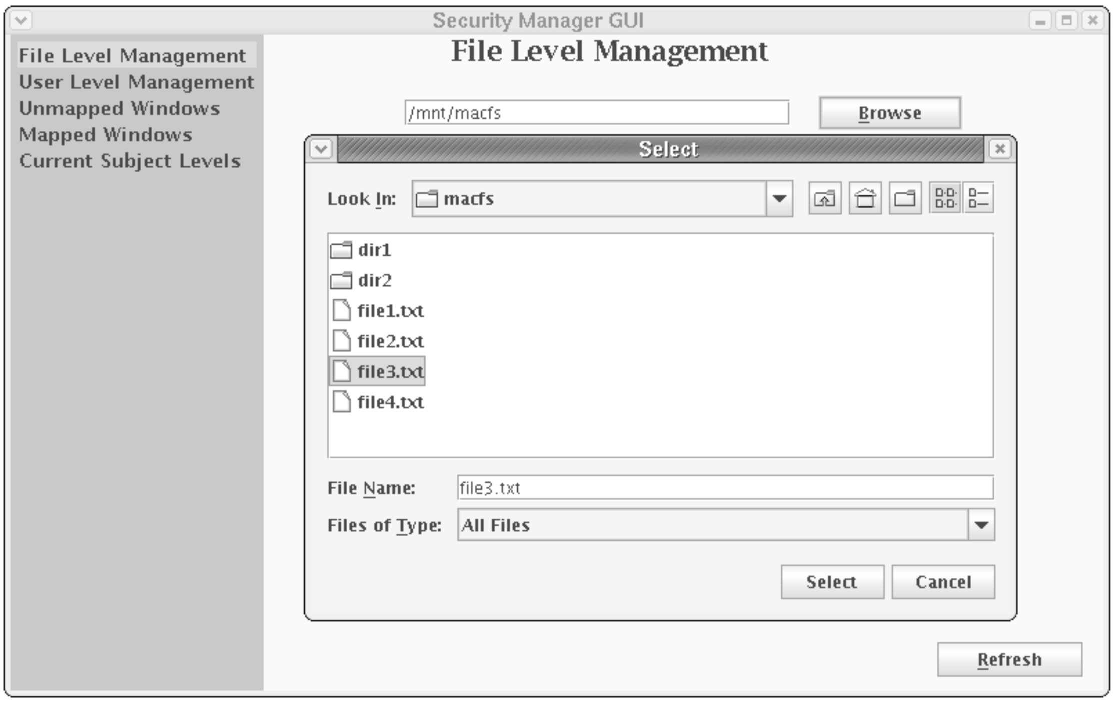
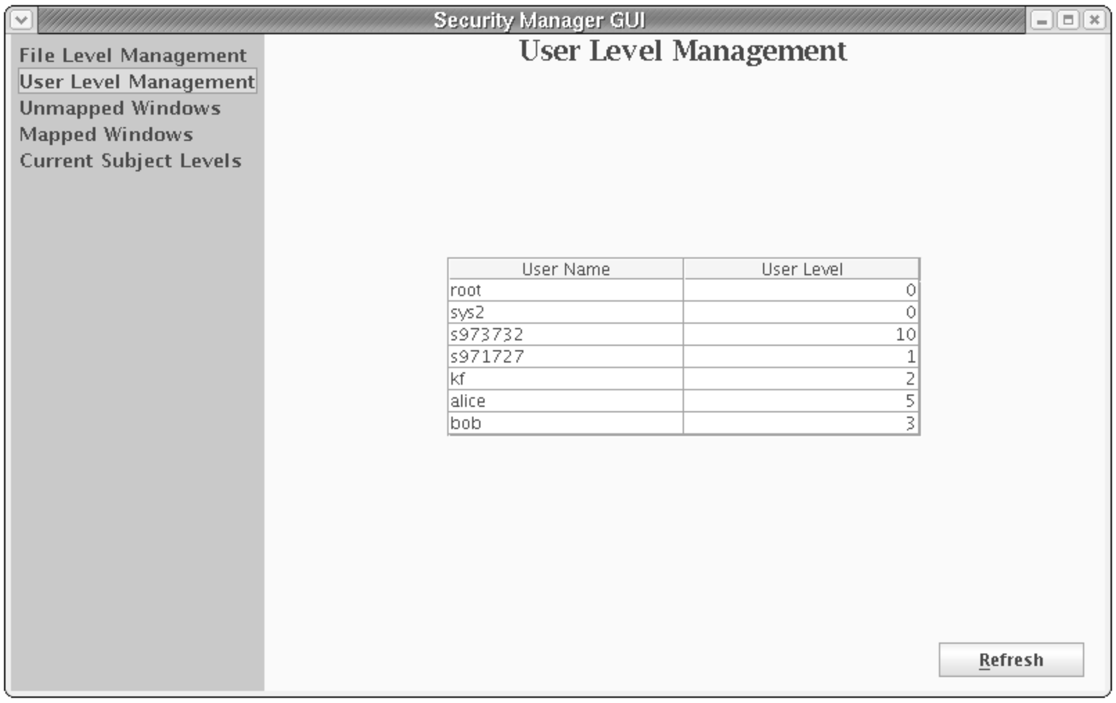
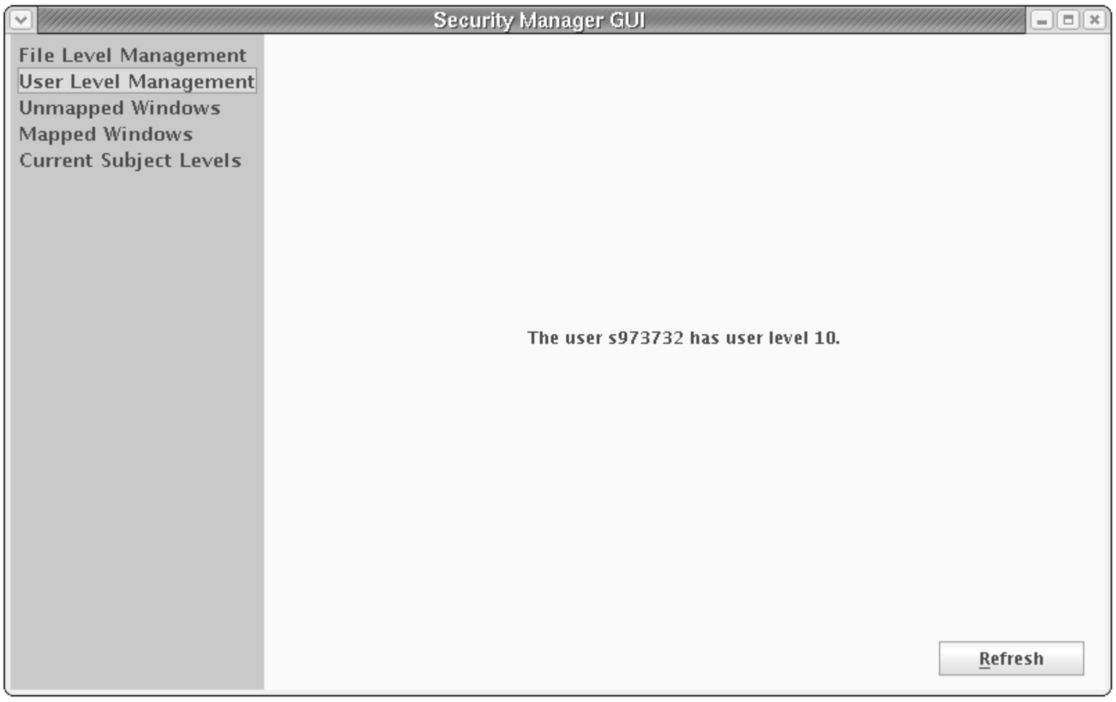


Figure E.2: When the `Browse` button is clicked, the user can select a file or directory. If a file is selected, the corresponding file level is displayed. If a directory is selected, all the file levels of the files and directories in the directory are listed.



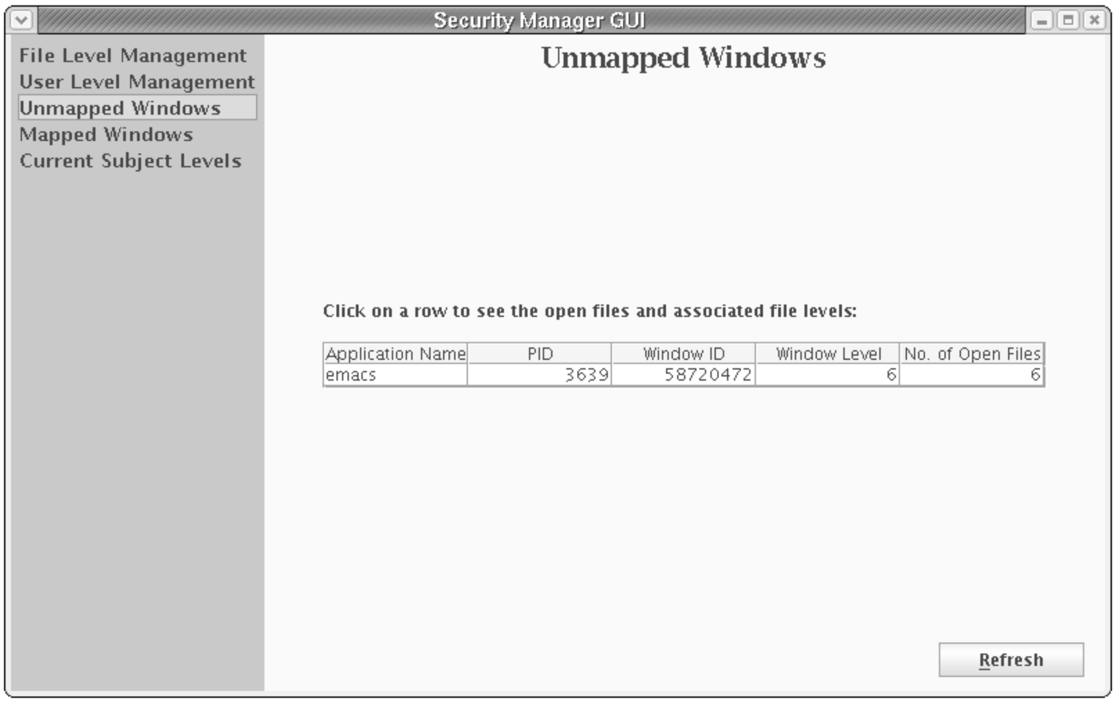
User Name	User Level
root	0
sys2	0
s973732	10
s971727	1
kf	2
alice	5
bob	3

Figure E.3: When the `User Level Management` menu item is selected and the super user runs the application, all the users and corresponding user levels are listed. The user levels are editable, and any changes are stored persistently.



The user s973732 has user level 10.

Figure E.4: When the `User Level Management` menu item is selected and a non-privileged user runs the application, only the user level of the user is displayed.

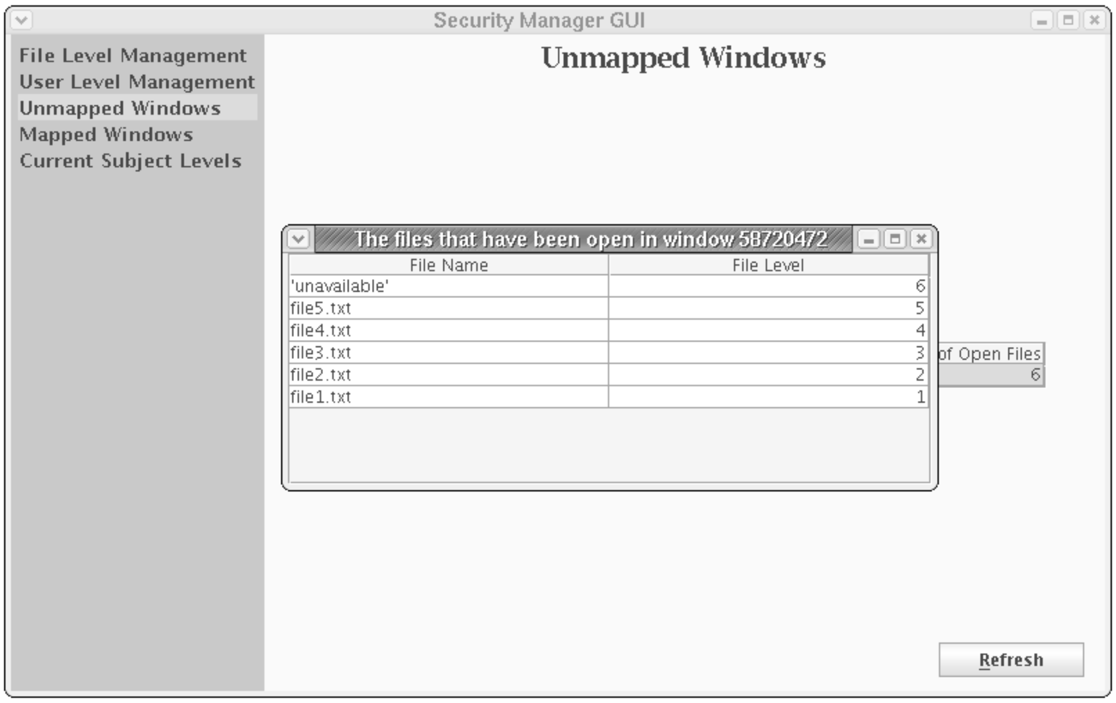


Unmapped Windows

Click on a row to see the open files and associated file levels:

Application Name	PID	Window ID	Window Level	No. of Open Files
emacs	3639	58720472	6	6

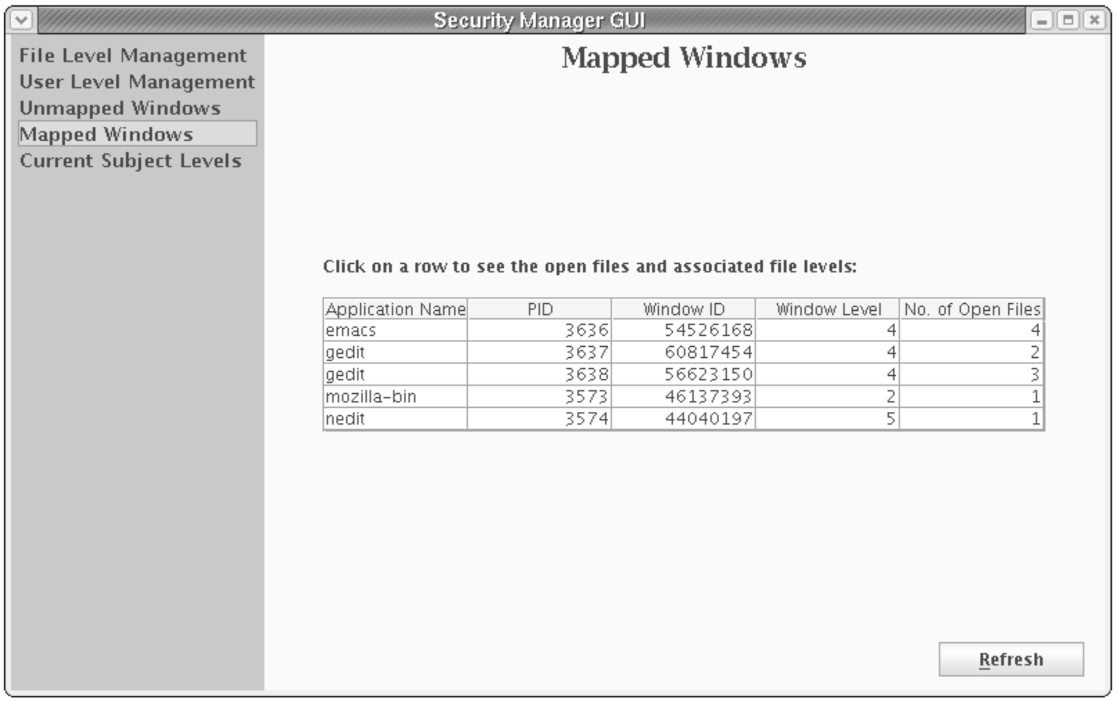
Figure E.5: One window was unmapped when this screen-shot was taken. The clearance level was 5 and the window was unmapped because its window level was 6 and the Bell LaPadula model should be enforced.



The files that have been open in window 58720472

File Name	File Level
'unavailable'	6
file5.txt	5
file4.txt	4
file3.txt	3
file2.txt	2
file1.txt	1

Figure E.6: The file names and corresponding file levels of the open files in the unmapped window. Because the *hide_non-readable_files* policy is chosen, the file name of the file with file level 6 is replaced by ‘unavailable’.

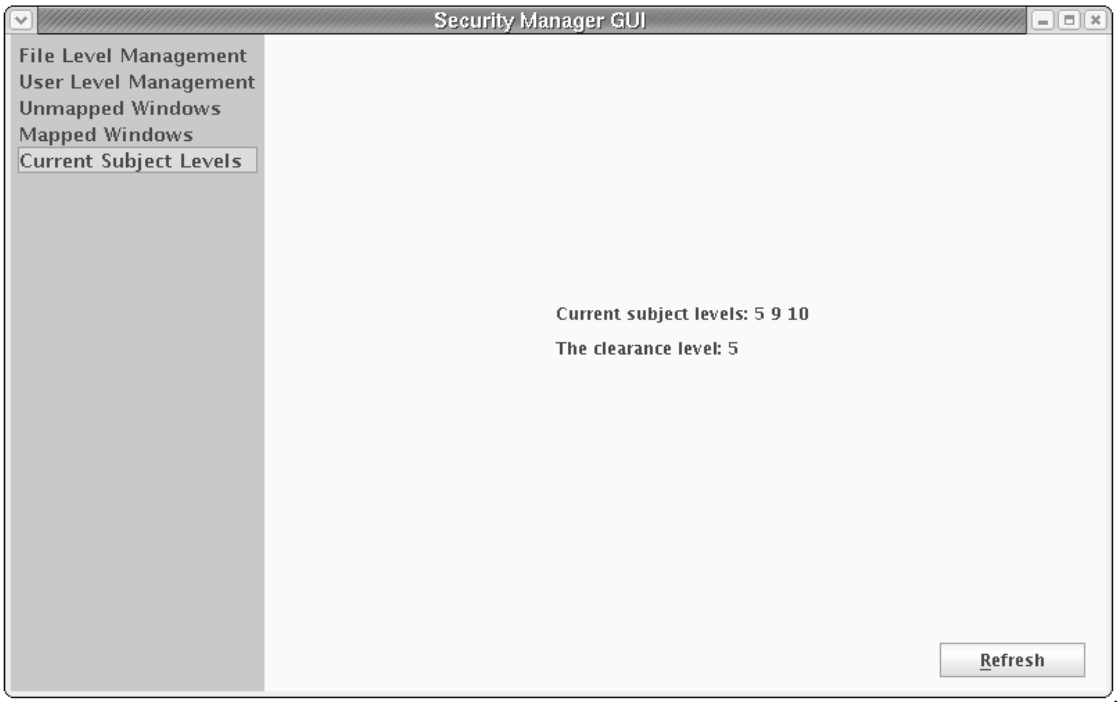


Mapped Windows

Click on a row to see the open files and associated file levels:

Application Name	PID	Window ID	Window Level	No. of Open Files
emacs	3636	54526168	4	4
gedit	3637	60817454	4	2
gedit	3638	56623150	4	3
mozilla-bin	3573	46137393	2	1
nedit	3574	44040197	5	1

Figure E.7: When the `Mapped Windows` menu item is selected, information about the five visible windows are listed.



Current subject levels: 5 9 10

The clearance level: 5

Figure E.8: When the `Current Subject Levels` menu item is selected, the subject levels of all the subjects who are currently present are listed. This list is always sorted in increasing order, and the first element is therefore equal to the clearance level.

Appendix F

Source Code

In this appendix, all the source code for our system is listed. This includes mainly C Code, but also FiST code and Java code. The files are listed according to which subsystem they belong to, starting at the lowest layer in our layered architecture.

F.1 Common Files

F.1.1 mount_point.h

```
#ifndef MOUNT_POINT
#define MOUNT_POINT "/mnt/macfs"
#endif
```

F.1.2 seac_ipc.h

```
#ifndef _SEAC_IPC_H
#define _SEAC_IPC_H

//////////////////////
// X11 declarations //
//////////////////////

#include <X11/Xlib.h>

#define CLIENT_MESSAGE "CLIENT_MESSAGE"

enum message_types {INIT_CLEARANCE_LEVEL = 0,
                    XCREATE_WINDOW_INTERCEPTOR,
                    FILE_OPEN_MONITOR,
                    BACKUP_INTERCEPTOR,
                    SET_FILE_LEVEL,
                    SENSOR_SERVER,
                    LIST_WINDOW_INFO,
                    LIST_SUBJECT_LEVELS,
                    DESTROY};
```

```
// Sends the message 'data' to the visibility_manager.
int send_xclient_event(long data[], Display* display, Window target_window);

enum backup_types {EMACS_BACKUP};


////////////////////////////
// Named pipe declaration //
////////////////////////////

#define FIFO_FILE "/tmp/.fifo%i"


////////////////////////////////
// Shared memory declarations //
////////////////////////////////

#define SHARED_MEMORY_KEY 123

enum shared_memory_offsets {
  SECURITY_MANAGER_SHM_OFFSET = 0,
  FILE_OPEN_MONITOR_SHM_OFFSET = sizeof(Window),
};


////////////////////////////
// Semaphore declarations //
////////////////////////////

#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
/* union semun is defined by including <sys/sem.h> */
#else
/* according to X/OPEN we have to define it ourselves */
union semun {
  int val;                    /* value for SETVAL */
  struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
  unsigned short *array;      /* array for GETALL, SETALL */
  /* Linux specific part: */
  struct seminfo *__buf;      /* buffer for IPC_INFO */
};
#endif

#define FILE_OPEN_MONITOR_SEM_NUM 0

#define SEMAPHORE_KEY 1234

#define NO_OF_SEMAPHORES 1

int semaphore_down(int sem_id, unsigned short sem_num);
int semaphore_up(int sem_id, unsigned short sem_num);

#endif
```

F.1.3 seac_ipc.c

```
#include "seac_ipc.h"

#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <errno.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* Wait on a binary semaphore. Block until the semaphore value is
   positive, then decrement it by one.
*/
int semaphore_down(int sem_id, unsigned short sem_num){
  struct sembuf operations[1];

  operations[0].sem_num = sem_num;

  // Decrement by one
  operations[0].sem_op = -1;

  // Permit undo'ing
  operations[0].sem_flg = SEM_UNDO;

  return semop(sem_id, operations, 1);
}


/* Post to a binary semaphore: increment its value by one. The
   function returns immediately.
*/
int semaphore_up(int sem_id, unsigned short sem_num){
  struct sembuf operations[1];

  operations[0].sem_num = sem_num;

  // Increment by one
  operations[0].sem_op = 1;

  // Permit undo'ing
  operations[0].sem_flg = SEM_UNDO;

  return semop(sem_id, operations, 1);
}


////////////////
// Xevent IPC //
////////////////

int send_xclient_event(long data[], Display* display, Window target_window){
```

```
int created_display = 0;

if(!display)
  // Use default display
  if(!(display = XOpenDisplay(NULL))){
    printf("Error opening display [%s]\n", XDisplayName(NULL));
    return -1;
  }
  else
    created_display = 1;

if(!target_window){
  // Initialize security_manager shared memory
  int shm_id = shmget(SHARED_MEMORY_KEY, sizeof(Window), 0);

  if(shm_id == -1)
    return -1;

  void *shm = shmat(shm_id, 0, 0);
  if(!shm){
    perror("shmat");
    return -1;
  }

  memcpy(&target_window, shm+SECURITY_MANAGER_SHM_OFFSET, sizeof(Window));

  shmdt(shm);
  if(!target_window)
    return -1;
}

Atom client_message = XInternAtom(display, "CLIENT_MESSAGE", False);

if(client_message == None)
  return -1;

Window root = RootWindow(display, DefaultScreen(display));
XEvent event;
event.type = ClientMessage;
event.xclient.display = display;
event.xclient.window = target_window;
event.xclient.message_type = client_message;
event.xclient.format = 32;
event.xclient.data.l[0] = data[0];
event.xclient.data.l[1] = data[1];
event.xclient.data.l[2] = data[2];
event.xclient.data.l[3] = data[3];
event.xclient.data.l[4] = data[4];

int status = XSendEvent(display, target_window, False, 0, (XEvent *) &event)
    ;

if(created_display){
  XFlush(display);
  XCloseDisplay(display);
  }

  return status;
}
```

F.1.4 security_policy_parameters.h

```
#ifndef _SECURITY_POLICY_PARAMETERS_H
#define _SECURITY_POLICY_PARAMETERS_H

#define HIDE_NON_READABLE_FILES "Hide non-readable files:"

#define PERMIT_LOWER_LEVEL_LOGIN "Permit lower level login:"

#define NO_READ_UP "No read up:"
#define NO_WRITE_DOWN "No write down:"

#define NO_READ_DOWN "No read down:"
#define NO_WRITE_UP "No write up:"

#endif
```

F.1.5 sensor.h

```
#ifndef _SENSOR_H
#define _SENSOR_H

#define DEFAULT_HOST "localhost"
#define DEFAULT_PORT 33333

#endif
```

F.2 The Stackable File System Files

The files listed in Appendix F.2.1 to Appendix F.2.4 contains kernel space code that make up our part stackable file system. The reaming code is available with the stackable file systems software package (fistgen), see [12]. The files listed in Appendix F.2.5 to Appendix F.2.7 contain user space code, which are used when the system is initialized or shut down.

F.2.1 macfs.fist

```
%{
  // The global clearance level. It is updated when an unauthorised
  // person enter or leave the environment.
  extern int clearance_level;
  // Is non-zero if the system either has not been initialised or it
  // has been destroyed.
  extern int is_done;

  // If non-zero, permits the user to specify a level below his user
  // level, which should be used to initialise the clearance level.
  extern int permit_lower_level_login;

  // If non-zero, file names of files that are non-readable according
  // to the security policy will be hidden.
  extern int hide_non_readable_files;

  // Bell-La Padula rules:
  extern int no_read_up;
  extern int no_write_down;

  // Biba rules:
  extern int no_write_up;
  extern int no_read_down;

  // The file names of the file levels file, user level file, and
  // security policy file, respectively:
  extern char file_levels[256];
  extern char user_levels[256];
  extern char security_policy[256];

  // Used when the open system call is invoked:
  extern struct semaphore open_sem;
  extern char file_name[256];
  extern int level;
  extern int pid;
  extern long file_inode;

  // Is invoked whenever a file is opened in the file system.
  extern void file_open_intercepted(long inode, int mode, const char *name);

  // Is invoked whenever the permissions of file is checked by the
  // file system.
  extern int inode_permission(long inode, int mask);

// DEBUG is set, debugging information is written to the system log.
#ifndef DEBUG
#define DEBUG 0
#endif
%}

// By filtering on the file names, the files that should be hidden can
// be skipped in a directory reading; this is only used when
// hide_non_readable_files != 0.
filter name;

// Source code that should be compiled and linked with the file system.
mod_src file_levels.c security_policy.c user_levels.c;

// Source code for stand-alone user space programs that use the file
// system via ioctl system calls.
```

```
user_src initcl.c seac_init.c seac_destroy.c setfl.c getfl.c listfl.c  setul.c
     getul.c listul.c getcl.c visibility_manager.c file_open_monitor.c
    seac_ipc.h mount_point.h  security_policy_parameters.h;

add_mk macfs_mk;

// Initialisation and shutdown operations:

ioctl INIT {
  char security_policy[256];
  char file_levels[256];
  char user_levels[256];
};

ioctl INIT_CLEARANCE_LEVEL {
  int level;
};

ioctl:fromuser DESTROY {
  int op;
};


// File level operations:

ioctl SET_FILE_LEVEL {
  int level;
  long inode;
};

ioctl GET_FILE_LEVEL {
  int level;
  long inode;
};


// User level operations:

ioctl SET_USER_LEVEL{
  int uid;
  int level;
};

ioctl GET_USER_LEVEL {
  int uid;
  int level;
};


// Clearance level operations:

ioctl SET_CLEARANCE_LEVEL {
  int level;
};

ioctl GET_CLEARANCE_LEVEL {
  int level;
};


// Open intercept operation:

ioctl OPEN {
  char name[256];
  int pid;
  int level;
  long inode;
};


// File formats that are used when file and user levels are stored in
// files.

fileformat FILE_LEVEL{
  long inode;
  int level;
};

fileformat USER_LEVEL{
  int uid;
  int level;
};

%%


//////////////////////////////////////////////
// Initialization and shutdown functions //
//////////////////////////////////////////////

%op:ioctl:INIT{
  if(current->euid)
    return -EPERM;

  if(fistGetIoctlData(INIT, security_policy, security_policy) < 0 ||
      fistGetIoctlData(INIT, file_levels, file_levels) < 0 ||
      fistGetIoctlData(INIT, user_levels, user_levels) < 0 ||
      !is_done)
    return -EFAULT;

  is_done = 0;

  if(read_hashtable() || read_policy())
    return -EFAULT;

  sema_init(&open_sem, 0);
}


%op:ioctl:INIT_CLEARANCE_LEVEL{
  int login_level;
  if(fistGetIoctlData(INIT_CLEARANCE_LEVEL, level, &login_level) < 0)
    return -EFAULT;

  if(clearance_level == -1){// The clearance level has not been initialised.
    clearance_level = get_user_level(current->uid);
    if(permit_lower_level_login && login_level != -1 &&
       login_level < clearance_level)
      clearance_level = login_level;
  }
  fistSetIoctlData(INIT_CLEARANCE_LEVEL, level, &clearance_level);
}


%op:ioctl:DESTROY {
  int op;

  if(current->euid)
    return -EPERM;

  if(fistGetIoctlData(DESTROY, op, &op) || is_done)
    return -EFAULT;

  write_hashtable();
  free_hashtable();
  is_done = -1; // Denotes that the file_open_monitor should stop running.
  up(&open_sem);

  if(DEBUG) printk("SEAC: the system is destroyed.\n");
}


//////////////////////////
// File level functions //
//////////////////////////

%op:ioctl:SET_FILE_LEVEL {
  long ino;
  int level;

  // Only root is allowed to change the file level.
  if(current->euid)
    return -EPERM;

  if(fistGetIoctlData(SET_FILE_LEVEL, level, &level) < 0 ||
     fistGetIoctlData(SET_FILE_LEVEL, inode, &ino) < 0 ||
     is_done)
    return -EFAULT;

  level = set_file_level(ino, level);

  fistSetIoctlData(SET_FILE_LEVEL, level, &level);
}


%op:ioctl:GET_FILE_LEVEL {
  long ino;
  int level;

  if(fistGetIoctlData(GET_FILE_LEVEL, inode, &ino) < 0 ||
     is_done)
    return -EFAULT;

  level = get_file_level(ino);

  if(level == -1) // A level is not associated with the inode.
    level = 0;

  if(hide_non_readable_files && inode_permission(ino, 4)) // Permission denied
    return -ENOENT;

  fistSetIoctlData(GET_FILE_LEVEL, level, &level);
}


//////////////////////////
// User level functions //
//////////////////////////

%op:ioctl:SET_USER_LEVEL {
  int uid;
  int level;

  // Only root is allowed to change the user level.
  if(current->euid)
    return -EPERM;

  if(fistGetIoctlData(SET_USER_LEVEL, level, &level) < 0 ||
     fistGetIoctlData(SET_USER_LEVEL, uid, &uid) < 0)
    return -EFAULT;

  if(DEBUG) printk("SEAC Set user level: uid = %i, user level = %i\n", uid,
      level);
  level = set_user_level(uid, level);

  fistSetIoctlData(SET_USER_LEVEL, level, &level);
}


%op:ioctl:GET_USER_LEVEL {
  int uid;
  int level;

  if(fistGetIoctlData(GET_USER_LEVEL, uid, &uid) < 0)
    return -EFAULT;

  if(current->euid && current->euid != uid)
    return -EPERM;

  level = get_user_level(uid);
  if(DEBUG) printk("SEAC Get user level: uid = %i, user level = %i\n", uid,
      level);
  fistSetIoctlData(GET_USER_LEVEL, level, &level);
}


///////////////////////////////
// Clearance level functions //
///////////////////////////////

%op:ioctl:SET_CLEARANCE_LEVEL {
  int level;

  // Only root is allowed to change the clearance level.
  if(current->euid)
    return -EPERM;

  if(fistGetIoctlData(SET_CLEARANCE_LEVEL, level, &level) < 0)
    return -EFAULT;

  if(DEBUG) printk("SEAC Set clearance level: level = %i\n", level);

  memcpy(&clearance_level, &level, sizeof(int));

}


%op:ioctl:GET_CLEARANCE_LEVEL {
  fistSetIoctlData(GET_CLEARANCE_LEVEL, level, &clearance_level);
  if(DEBUG) printk("SEAC Get clearance level: level = %i\n", clearance_level);
}


/////////////////////////////
// Open intercept function //
/////////////////////////////

%op:ioctl:OPEN {
  err = down_interruptible(&open_sem);
  // down_interruptible returns 0 if you got the lock, or -EINTR if
  // the process was interrupted with a signal.
  if(!err){
    if(is_done)
      fistSetIoctlData(OPEN, level, &is_done);
    else{
      fistSetIoctlData(OPEN, name, file_name);
      fistSetIoctlData(OPEN, pid, &pid);
      fistSetIoctlData(OPEN, level, &level);
      fistSetIoctlData(OPEN, inode, &file_inode);
    }
  }
}


/////////////////////
// File operations //
/////////////////////
```

```
%op:create:postcall {
  if(DEBUG) printk("SEAC create: file name = %s, inode = %i, level = %i\n",
      dentry->d_name.name, dentry->d_inode->i_ino, clearance_level);

  if(!err)
    insert(dentry->d_inode->i_ino, clearance_level);
}


%op:unlink:postcall {
  if(DEBUG) printk("SEAC unlink: file name = %s, inode = %i, link count = %i\n
      ", dentry->d_name.name, dentry->d_inode->i_ino, dentry->d_inode->i_nlink
      );

  if(!err && dentry->d_inode->i_nlink == 0)
    delete(dentry->d_inode->i_ino);
}


//////////////////////////
// Directory operations //
//////////////////////////

%op:mkdir:postcall {
  if(DEBUG) printk("SEAC mkdir: directory name = %s, inode = %i, file level
       = %i\n", dentry->d_name.name, dentry->d_inode->i_ino, clearance_level);

  if(!err)
    insert(dentry->d_inode->i_ino, clearance_level);
}


%op:rmdir:postcall {
  if(DEBUG) printk("SEAC rmdir: directory name = %s, inode = %i, link count
       = %i\n", dentry->d_name.name, dentry->d_inode->i_ino, dentry->d_inode->
      i_nlink);

  if(!err && dentry->d_inode->i_nlink == 0)
    delete(dentry->d_inode->i_ino);
}

%op:readdir:call {
  if(current->euid && hide_non_readable_files && inode_permission(ino, 4)){
      if(DEBUG) printk("SEAC readdir: skipping file %s\n", decoded_name);
    fistSkipName(decoded_name);
  }
}

%%

int clearance_level = -1;
int is_done = 1;

// Security policy parameters:

int permit_lower_level_login = 0;
int hide_non_readable_files = 0;

// Bell-La Padula:
int no_read_up = 0;
int no_write_down = 0;

// Biba:
int no_write_up = 0;
int no_read_down = 0;

char security_policy[256];
char file_levels[256];
char user_levels[256];

// File open monitor variables:
struct semaphore open_sem;
int pid;
int level;
char file_name[256];
long file_inode;


// Taken from base2fs.fist
int macfs_encode_filename(const char *name, int length, char **encoded_name,
    int skip_dots, const vnode_t *vp, const vfs_t *vfsp){
  int encoded_length = length + 1;
  *encoded_name = fistMalloc(encoded_length);
  fistMemCpy(*encoded_name, name, length);
  (*encoded_name)[length] = '\0';
  return encoded_length;
}

// returns length of decoded string, or -1 if error
int macfs_decode_filename(const char *name, int length, char **decoded_name,
    int skip_dots, const vnode_t *vp, const vfs_t *vfsp){
  int error = 0;
  *decoded_name = fistMalloc(length);
  fistMemCpy(*decoded_name, name, length);
  error = length;
  return error;
}


void file_open_intercepted(long ino, int mode, const char *name){

  if(!S_ISREG(mode) && !S_ISLNK(mode))
    return;

  int new_level = get_file_level(ino);
  if(new_level == -1){
    printk("SEAC file_open_intercepted() error: the inode %i was not found\n")
        ;
    return;
  }
  if(DEBUG) printk("SEAC file open intercepted: file name = %s, inode =  %i,
      file level = %i, PID = %i\n", name, ino, new_level, current->pid);
  memcpy(&pid, &current->pid, sizeof(pid));
  memcpy(&level, &new_level, sizeof(level));
  memcpy(&file_inode, &ino, sizeof(ino));
  memcpy(file_name, name, 256);
  if(DEBUG) printk("SEAC file open intercepted: file name = %s\n", file_name);
  up(&open_sem);
}

// Security policy function. Returns 0 if the inode can be accessed
// with the provided mask.
int inode_permission(long inode, int mask){

  int file_level = get_file_level(inode);
  if(file_level == -1)
    return 0; // A level has not been associated with the file.

  if(current->uid && // Only the access for non-root users is
                     // restricted.
     (// Bell-La Padula rules:
      no_read_up && mask & 4 && file_level > clearance_level ||
      no_write_down && mask & 2 && file_level < clearance_level ||
      // Biba rules:
      no_write_up && mask & 2 && file_level > clearance_level ||
      no_read_down && mask & 4 && file_level < clearance_level)){
    if(DEBUG) printk("Permission denied for inode %i\n", inode);
    return -EPERM;
  }

  return 0;
}
```

F.2.2 security_policy.c

```
#ifdef FISTGEN
#include "fist_macfs.h"
#endif
#include "fist.h"

#include "security_policy_parameters.h"

// Reads a line from the policy file
int read_line(file_t *filp, char *option, int *choice){
  char buf[40];
  int bytes = filp->f_op->read(filp, buf, strlen(option)+2, &filp->f_pos);

  if(!strncmp(buf, option, strlen(option))){
    *choice = buf[strlen(option)+1];

    if(*choice != (int) 'y' && *choice != (int) 'n'){
      printk("SEAC error: invalid format i security policy file. The format
          must be \"%s x\" where x is either y or n\n", option);
      return -1;
      }
      *choice -= 'n';
    }

    while(bytes > 0 && buf[0] != '\n')
      bytes = filp->f_op->read(filp, buf, 1, &filp->f_pos);

    return 0;
}

int read_policy(){
    file_t *filp;
    mm_segment_t oldfs;
    int err = 0;

    if(DEBUG) printk("SEAC security policy:\n");
    filp = filp_open(security_policy, O_RDONLY, 0);

    if (!filp || IS_ERR(filp))
      return -1;

    if (!filp->f_op->read)
      return -2;  // file(system) doesn't allow reads

    filp->f_pos = 0; // start offset
    oldfs = get_fs();
    set_fs(KERNEL_DS);

    if(err = read_line(filp, HIDE_NON_READABLE_FILES, &hide_non_readable_files)
        < 0)
      goto out;
    else
      if(DEBUG) printk("    hide_non_readable_files = %i\n",
          hide_non_readable_files != 0);

    if(err = read_line(filp,  NO_READ_UP, &no_read_up) < 0)
      goto out;
    else
      if(DEBUG) printk("    no_read_up = %i\n", no_read_up != 0);

    if(err = read_line(filp,  NO_READ_DOWN, &no_read_down) < 0)
      goto out;
    else
      if(DEBUG) printk("    no_read_down = %i\n", no_read_down != 0);

    if(err = read_line(filp,  NO_WRITE_DOWN, &no_write_down) < 0)
      goto out;
    else
      if(DEBUG) printk("    no_write_down = %i\n", no_write_down != 0);

    if(err = read_line(filp,  NO_WRITE_UP, &no_write_up) < 0)
      goto out;
    else
      if(DEBUG) printk("    no_write_up = %i\n", no_write_up != 0);
```

```
  if(err = read_line(filp, PERMIT_LOWER_LEVEL_LOGIN, &permit_lower_level_login
      ) < 0)
    goto out;
  else
    if(DEBUG) printk("    permit_lower_level_login = %i\n",
        permit_lower_level_login != 0);

 out:
  set_fs(oldfs);
  fput(filp); // close the file

  return err;
}
```

F.2.3 file_levels.c

```
#ifdef FISTGEN
#include "fist_macfs.h"
#endif
#include "fist.h"

// Hashtable node
typedef struct node *node_t;
struct node{
  long inode;
  int level; // file level
  node_t next;  // Pointer to the next element.
};

#ifndef HASH_TABLE_SIZE
#define HASH_TABLE_SIZE 10
#endif

static node_t hashtable[HASH_TABLE_SIZE];

static int size = 0; // The number of elements in the hashtable.

///////////////////////////
// Hash table functions //
///////////////////////////

static unsigned int hash(int key) { return key % HASH_TABLE_SIZE; }

// Prints the content of the hash table.
static void print(){

  if(!size){
    printk("\nThe hash table is empty.\n");
    return;
  }

  int i;
  for(i = 0; i < HASH_TABLE_SIZE; i++){
    node_t x;
    printk("\ni = %i: ", i);

    for(x = hashtable[i]; x; x = x->next)
      printk("(%u, %i) ", x->inode, x->level);
  }
  printk("\n");
}


static node_t search(int k) {
  node_t x = hashtable[hash(k)];

  while(x && x->inode != k)
    x = x->next;

  return x;
}


int insert(long inode, int level) {

  if(inode < 0 || level < 0)
    return -1;

  node_t x = hashtable[hash(inode)];

  while(x && x->inode != inode)
    x = x->next;

  if(x){  // Update the node
    if(x->level != level)
      x->level =level;
  }
  else{ // Insert a new node
    node_t y;
    y = kmalloc(sizeof(struct node), GFP_KERNEL);
    if(!y){
      printk("SEAC Error: Out of memory.");
      return -1;
    }

    size++;
    int index = hash(inode);
    // Create a new node ''y'' with key value inode
    y->inode = inode;
    y->level = level;
    y->next = hashtable[index];
    hashtable[index] = y;
  }

  if(DEBUG){
    printk("SEAC hashtable after insert:");
    print();
  }
  return 0;
}
```

```
void delete(long inode) {

  node_t x = hashtable[hash(inode)], y = 0;

  for(; x && x->inode != inode; x = x->next)
    y = x;

  if(!x){ // inode does not exist in table
    printk("SEAC delete error: inode %i was not found.\n", inode);
    return;
  }

  size--;

  if (y) // The inode was not the first element in the list.
    y->next = x->next;
  else  // Delete the first element in the list
    hashtable[hash(inode)] = x->next;


  kfree(x);

  if(DEBUG){
    printk("SEAC hashtable after delete:");
    print();
  }
}

// Read the content of a hash table from a file and store it in table
int read_hashtable(){
  struct _fist_fileformat_FILE_LEVEL buf;
  int len = sizeof(struct _fist_fileformat_FILE_LEVEL);

  file_t *filp;
  mm_segment_t oldfs;
  int bytes = 1;

  filp = filp_open(file_levels, O_RDONLY, 0);

  if (!filp || IS_ERR(filp))
    return -1;

  if (!filp->f_op->read)
    return -2;

  filp->f_pos = 0; // start offset
  oldfs = get_fs();
  set_fs(KERNEL_DS);

  // Read an inode and a corresponding level from the file and
  // insert it into the hashtable.
  bytes = filp->f_op->read(filp, (void *) &buf, len, &filp->f_pos);

  while(bytes > 0){
    insert(buf.inode, buf.level);
    bytes = filp->f_op->read(filp, (void *) &buf, len, &filp->f_pos);
  }

  set_fs(oldfs);
  fput(filp); // close the file

  return 0;
}

// Store the content of table in a file.
int write_hashtable(){

  struct _fist_fileformat_FILE_LEVEL buf;
  int len = sizeof(struct _fist_fileformat_FILE_LEVEL);
  file_t *filp;
  mm_segment_t oldfs;
  int bytes, i;

  filp = filp_open(file_levels, O_WRONLY, 0);

  if (!filp || IS_ERR(filp))
    return -1;

  if (!filp->f_op->write)
    return -2;

  filp->f_pos = 0;  // start offset
  oldfs = get_fs();
  set_fs(KERNEL_DS);

  for(i = 0; i < HASH_TABLE_SIZE; i++){
    node_t x = hashtable[i];

    for(;x; x = x->next){
      memcpy(&buf.inode, &x->inode, sizeof(x->inode));
      memcpy(&buf.level, &x->level, sizeof(x->level));
      bytes = filp->f_op->write(filp, (void *) &buf, len, &filp->f_pos);
    }
  }
  set_fs(oldfs);
  fput(filp); // close the file

  return 0;
}


void free_hashtable(){
  int i;

  for(i = 0; i < HASH_TABLE_SIZE; i++){
    node_t x, y;

    for(x = hashtable[i];;){
      if(x)
        y = x->next;
        else
          break;
        if(DEBUG) printk("kfree (%u, %i)\n", x->inode, x->level);
        kfree(x);
        x = y;
      }
    }
}


// Returns the file level for the file or directory with the given
// inode, or -1 if no level has been associated with the inode.
int get_file_level(long inode){
  node_t x = search(inode);

  if(x)
    // inode was in the hashtable.
    return x->level;
  else
    return -1;
}

int set_file_level(long inode, int new_level){

  if(new_level < 0)
    return -1;

  return insert(inode, new_level);
}
```

F.2.4 user_levels.c

```
#ifdef FISTGEN
#include "fist_macfs.h"
#endif
#include "fist.h"

/**
 * Sets the user level for the user with the ID uid.
 */
int set_user_level(int uid, int level){
  struct _fist_fileformat_USER_LEVEL buf;
  int len = sizeof(struct _fist_fileformat_USER_LEVEL);
  file_t *filp;
  mm_segment_t oldfs;
  int uid_found = 0; // Is 0 iff the uid was found in the file

  // open the file
  filp = filp_open(user_levels, O_WRONLY, 0);

  if(!filp || IS_ERR(filp))
    return -1;

  if (!filp->f_op->write || !filp->f_op->read)
```

```
    return -2;

  filp->f_pos = 0; // start offset
  oldfs = get_fs();
  set_fs(KERNEL_DS);

  while(filp->f_op->read(filp, (void *) &buf, len, &filp->f_pos) > 0)
    if(buf.uid == uid){
      // The uid already exists in the file, so the old level is
      // overwritten by the new level.
      memcpy(&buf.level, &level, sizeof(level));
      uid_found = 1;
      filp->f_pos = filp->f_pos - len;
      filp->f_op->write(filp, (void *) &buf, len, &filp->f_pos);
      break;
    }

  if(!uid_found){
    // The uid was not found in the file, so the uid and corresponding
    // level is appended to the file.
    memcpy(&buf.uid, &uid, sizeof(uid));
    memcpy(&buf.level, &level, sizeof(level));
    filp->f_op->write(filp, (void *) &buf, len, &filp->f_pos);
  }

  set_fs(oldfs);
  fput(filp); // close the file
  return 0;
}

/**
 * Gets the user level for the user with the ID uid.
 */
int get_user_level(int uid){

  struct _fist_fileformat_USER_LEVEL buf;
  int len = sizeof(struct _fist_fileformat_USER_LEVEL);
  file_t *filp;
  mm_segment_t oldfs;
  int level = 0;

  filp = filp_open(user_levels, O_RDONLY, 0);

  if (!filp || IS_ERR(filp))
    return -1;

  if (!filp->f_op->read)
    return -2; // file system does not allow reads

  filp->f_pos = 0; // start offset
  oldfs = get_fs();
  set_fs(KERNEL_DS);

  while(filp->f_op->read(filp, (void *) &buf, len, &filp->f_pos) > 0)
    if(buf.uid == uid){
      // The uid was found in the file.
      memcpy(&level, &buf.level, sizeof(level));
      break;
    }

  set_fs(oldfs);
  fput(filp); // close the file
  return level;
}
```

F.2.5 seac_init.c

```
#include <sys/types.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdio.h>

#include "mount_point.h"
#include <wrapfs.h>

int main(int argc, char *argv[]){

  struct _fist_ioctl_INIT val;

  if (argc < 4) {
    fprintf(stderr, "Usage: %s POLICY_FILE FILE_LEVELS_FILE USER_LEVELS_FILE\n
        ", argv[0]);
    exit(1);
  }

  strcpy(val.security_policy, argv[1]);
  strcpy(val.file_levels, argv[2]);
  strcpy(val.user_levels, argv[3]);

  int fd = open(MOUNT_POINT, O_RDONLY);
  if (fd < 0) {
    perror("open");
    exit(1);
  }

  // Set the clearance level in the file system.
  int status = ioctl(fd, FIST_IOCTL_INIT, &val);
  if (status < 0)
    perror("Could not access file system");

  close(fd);
  exit(status);
}
```

F.2.6 initcl.c

```
#include <sys/types.h>
#include <sys/ioctl.h>
```

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#include "mount_point.h"
#include "seac_ipc.h"
#include <wrapfs.h>

int main(int argc, char *argv[]){

  if (argc > 2) {
    fprintf(stderr, "Usage: %s [LOGIN_LEVEL]\n", argv[0]);
    exit(1);
  }

  struct _fist_ioctl_INIT_CLEARANCE_LEVEL val;

  if (argc == 2)
    val.level = atoi(argv[1]);
  else
    val.level = -1; // The level will be set in the kernel to the
                    // user's user level

  int fd = open(MOUNT_POINT, O_RDONLY);
  if (fd < 0) {
    perror("open");
    exit(1);
  }

  int res = ioctl(fd, FIST_IOCTL_INIT_CLEARANCE_LEVEL, &val);
  close(fd);
  if (res < 0) {
    perror("Could not access file system");
    exit(res);
  }

  long data[] = {INIT_CLEARANCE_LEVEL, val.level, 0, 0, 0};
  send_xclient_event(data, 0, 0);
}
```

F.2.7 seac_destroy.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#include "mount_point.h"
#include "seac_ipc.h"

#include <wrapfs.h>
```

```
int main(){

  struct _fist_ioctl_DESTROY val;

  int fd = open(MOUNT_POINT, O_RDONLY);
  if (fd < 0) {
    perror("open");
    exit(1);
  }

  // Store the file system state and stop the file_open_monitor process
  int status = ioctl(fd, FIST_IOCTL_DESTROY, &val);
  if (status < 0)
    perror("Could not access file system");

  close(fd);

  // Shut down the visibility_manager
  long data[] = {DESTROY, 0, 0, 0, 0};
  return send_xclient_event(data, 0, 0);
}
```

F.3 File Level Management Files

F.3.1 getfl.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#include "mount_point.h"
#include <wrapfs.h>

int main(int argc, char *argv[]){

  struct _fist_ioctl_GET_FILE_LEVEL val;
  struct stat stat_buf;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s FILE\n", argv[0]);
    exit(1);
  }

  char *cwd = getcwd(0,0);
  if(strncmp(MOUNT_POINT, cwd, strlen(MOUNT_POINT)) &&
     (argc == 1 || strncmp(MOUNT_POINT, argv[1], strlen(MOUNT_POINT)))) {
    printf("Error: %s is not supported in this directory.\n", argv[0]);
    free(cwd);
    exit(1);
  }
  free(cwd);

  int status = stat(argv[1], &stat_buf);
  if (status < 0) {
    perror("stat");
    exit(1);
  }
  val.inode = stat_buf.st_ino;

  int fd = open(MOUNT_POINT, O_RDONLY);
  if (fd < 0) {
    perror("open");
    exit(1);
  }

  status = ioctl(fd, FIST_IOCTL_GET_FILE_LEVEL, &val);
  if(status == -1){
    if(val.level == -1)
      printf("No file level has been set.\n");
    else
      perror("Could not access file system");
  }
  else
      printf("%u\n", val.level);

  close(fd);
}
```

F.3.2 setfl.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>

#include "seac_ipc.h"
#include "mount_point.h"

#include <wrapfs.h>

int main(int argc, char *argv[]){

  struct _fist_ioctl_SET_FILE_LEVEL val;
  struct stat stat_buf;

  if (argc < 3) {
    fprintf(stderr, "Usage: %s FILE FILE_LEVEL\n", argv[0]);
    exit(1);
  }

  char *file_name = argv[1];
  char *cwd = getcwd(0,0);
  // Check that the path to the file is valid.
  if(strncmp(MOUNT_POINT, cwd, strlen(MOUNT_POINT)) &&
     (argc == 1 || strncmp(MOUNT_POINT, file_name, strlen(MOUNT_POINT)))) {
    printf("Error: %s is not supported in this directory.\n", argv[0]);
    free(cwd);
    exit(1);
  }
  free(cwd);

  // Get the inode number corresponding to the file.
  int status = stat(file_name, &stat_buf);
  if (status < 0) {
    perror("stat");
    exit(1);
  }

  val.inode = stat_buf.st_ino;
  val.level = atoi(argv[2]);
  long data[] = {SET_FILE_LEVEL, val.inode, val.level, 0, 0};

  int fd = open(MOUNT_POINT, O_RDONLY);
  if (fd < 0) {
    perror("open");
    exit(1);
  }

  // Set the file level in the stackable file system.
  status = ioctl(fd, FIST_IOCTL_SET_FILE_LEVEL, &val);
  if (status < 0){
    perror("Could not access file system");
    close(fd);
    exit(1);
  }
  close(fd);

  if(val.level < 0){
    perror("setfl");
    exit(1);
  }

  // Notify the visibility_manager about the level change.
  send_xclient_event(data, 0, 0);
}
```

F.3.3 listfl.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
```

```
#include "mount_point.h"
#include <wrapfs.h>

int main(int argc, char *argv[]){

  struct _fist_ioctl_GET_FILE_LEVEL val;
  struct stat stat_buf;

  if (argc > 2) {
    fprintf(stderr, "Usage: %s [DIR]\n", argv[0]);
    exit(1);
  }

  char *cwd = getcwd(0,0);
  if(strncmp(MOUNT_POINT, cwd, strlen(MOUNT_POINT)) &&
     (argc == 1 || strncmp(MOUNT_POINT, argv[1], strlen(MOUNT_POINT)))) {
    printf("Error: listfl is not supported in this directory.\n");
    free(cwd);
    exit(1);
  }
  free(cwd);

  char *dir_name = (argc < 2) ? "." : argv[1];
  DIR *dir = opendir(dir_name);
  if(!dir){
    perror("opendir");
    exit(1);
  }

  if(chdir(dir_name) == -1){
    perror("chdir");
    exit(1);
  }

  struct dirent *dirent = readdir(dir);
  if(!dirent) {
    perror("readdir");
    exit(1);
  }
   dirent = readdir(dir); // .
   dirent = readdir(dir); // ..

  int fd = open(MOUNT_POINT, O_RDONLY);
  if (fd < 0) {
    perror("open");
    exit(fd);
  }

  for(;dirent; dirent = readdir(dir)){
    char *name = dirent->d_name;

    int status = stat(name, &stat_buf);
    if(status < 0)
      continue;
    val.inode = stat_buf.st_ino;

    status = ioctl(fd, FIST_IOCTL_GET_FILE_LEVEL, &val);
    if (status < 0) {
      perror("Could not access file system");
      break;
    }

    printf("%-15s %10u\n", name, val.level);
  }

  close(fd);
  closedir(dir);
}
```

F.4 User Level Management Files

F.4.1 getul.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <pwd.h>
#include <sys/types.h>

#include "mount_point.h"

#include <wrapfs.h>

int main(int argc, char *argv[]){

  struct _fist_ioctl_GET_USER_LEVEL val;
  struct passwd *p_entry;

  if (argc > 2) {
    fprintf(stderr, "Usage: %s [USER_NAME]\n", argv[0]);
    exit(1);
  }

  if(argc == 1)
    // If no argument is supplied, the user ID of the current user is
    // retrieved.
    val.uid = getuid();
  else{
    p_entry = getpwnam(argv[1]); // get password entry

    if (!p_entry) {
      printf("The user level could not be retrieved.\n");
      exit(1);
    }
    val.uid = p_entry->pw_uid;
  }

  int fd = open(MOUNT_POINT, O_RDONLY);
  if (fd < 0) {
    perror("open");
    exit(1);
  }

  // Get the user level is from the stackable file system.
  int status = ioctl(fd, FIST_IOCTL_GET_USER_LEVEL, &val);
  if (status < 0) {
    perror("ioctl");
  }
  close(fd);

  if(val.level < 0){
    printf("The user level could not be retrieved.\n");
    perror("ioctl");
  }
  else
    printf("%i\n", val.level);
}
```

F.4.2 setul.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <pwd.h>
#include <sys/types.h>

#include "mount_point.h"
#include <wrapfs.h>

int main(int argc, char *argv[]){

  struct _fist_ioctl_SET_USER_LEVEL val;
  struct passwd *p_entry;

  if (argc < 3) {
    fprintf(stderr, "Usage: %s USER_NAME USER_LEVEL\n", argv[0]);
    exit(1);
  }

  int fd = open(MOUNT_POINT, O_RDONLY);
  if (fd < 0) {
    perror(argv[1]);
    exit(1);
  }
  p_entry = getpwnam(argv[1]); // get password entry

  if (!p_entry) {
    printf("The user level could not be set.\n");
    exit(1);
  }

  val.uid = p_entry->pw_uid;
  val.level = atoi(argv[2]);

  int res = ioctl(fd, FIST_IOCTL_SET_USER_LEVEL, &val);
  if (res < 0) {
    perror("Could not access file system");
  }

  if(val.level == -1){
    printf("The user level could not be set.\n");
    perror("setul");
  }

  close(fd);
  exit(res);
}
```

F.4.3 listul.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <pwd.h>
#include <sys/types.h>

#include "mount_point.h"
#include <sys/mman.h>

#include <wrapfs.h>
#include <glib.h>

#define MAX_PATH_LENGTH 50

#ifndef SHELL_PATH
#define SHELL_PATH "/etc/shells"
#endif

static gint compare_strings(gconstpointer a, gconstpointer b){
  return strcmp(a, b);
}

int main(){
```

```
FILE *fp = fopen(SHELL_PATH, "r");
if(!fp){
  perror("fopen");
  exit(1);
}

//  The /etc/shells file is (usually) list of all of the valid
// 'login' shells on the system -- see the man page 'shells',
// section 5. A linked list with these valid shells is created
// initially.
GSList *login_shells_list = NULL;
char buf[MAX_PATH_LENGTH];
while(!feof(fp) && !ferror(fp) && fscanf(fp, "%s\n", buf))
  login_shells_list = g_slist_append(login_shells_list,
                                     g_memdup(buf, strlen(buf)+1));

if(fclose(fp)){
  perror("fclose");
  exit(1);
}


struct _fist_ioctl_GET_USER_LEVEL val;
struct passwd *p_entry;

int fd = open(MOUNT_POINT, O_RDONLY);
if (fd < 0) {
  perror("open");
  exit(1);
}

// The password file is scanned, and all user IDs corresponding to
// valid login shells are printed to standard output along with the
// user level of the user.
for(p_entry = getpwent();  // get a password entry
    p_entry; p_entry = getpwent()){

  if(!g_slist_find_custom(login_shells_list, p_entry->pw_shell, &
      compare_strings) || !strcmp("/sbin/nologin", p_entry->pw_shell))
    continue;

  val.uid = p_entry->pw_uid;

  // The user level is retrieved from the stackable file system.
  int res = ioctl(fd, FIST_IOCTL_GET_USER_LEVEL, &val);
  if (res < 0) {
    perror("Could not access file system");
    break;
  }

  if(val.level < 0){
    printf("Could not read the user levels.\n");
    break;
  }

  // The user name and corresponding user level is printed.
```

```
    printf("%-15s %10u\n", p_entry->pw_name, val.level);
  }

  g_slist_free(login_shells_list);

  endpwent();
  close(fd);
}
```

F.5 Window Management Files

F.5.1 visibility_manager.c

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <glib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/fcntl.h>
#include <fcntl.h>
#include <wrapfs.h>
#include <X11/Xatom.h>

#include "security_policy_parameters.h"
#include "seac_ipc.h"
#include "mount_point.h"

#ifndef DEBUG
#define DEBUG 0
#endif

#define error(str) fprintf(stderr, str);
#define error2(str, arg) fprintf(stderr, str, arg);

// Security policy parameters:
static int hide_non_readable_files;
static int no_read_up;
static int no_read_down;

static GHashTable *table; // A hash table where the key is a process
                          // ID of type int, and the information value
                          // is an element of type WindowInfo.
static GSList *subject_list = NULL;  // The list is sorted in increasing order
    so that the clerance level is the first element in the list.


typedef struct {
  char *file_name;
  int file_level;
  unsigned long inode;
} FileInfo;

typedef struct {
  Window window_id;
  char *app_name;
  GSList *file_list; // A list where the elements are FileInfo structures. The
       list is sorted in decreasing order (according to the file level) so
     that the window level is the file level in the first element of the list
     .
  int is_mapped; // is 1 if and only if the window is mapped.
} WindowInfo;



static int x_error_handler(Display *display, XErrorEvent *error){
  char error_msg[80];
  XGetErrorText(display, error->error_code, error_msg, sizeof(error_msg));
  error2("XError: %s\n", error_msg);
  return 0;
}


// Reads a line from the security policy file.
int read_line(int fd, char *option, int *choice){
  char buf[40];
  int bytes = read(fd, buf, strlen(option)+2);

  if(!strncmp(buf, option, strlen(option))){
    *choice = buf[strlen(option)+1];

    if(*choice != (int) 'y' && *choice != (int) 'n'){
      error2("Invalid format. The format must be \"%s x\" where x is either y
          or n\n", option);
      return -1;
    }
    *choice -= 'n';
  }

  while(bytes > 0 && buf[0] != '\n')
    bytes = read(fd, buf, 1);

  return 0;
}


// GHashTable functions
///////////////////////

static gboolean equalFunc(gconstpointer a, gconstpointer b){
  return *((const pid_t *) a) == *((const pid_t *) b);
}

static guint hashFunc(gconstpointer key){
  return *((const pid_t *) key);
}
```

```
// GSList functions
///////////////////

// Used when elements are inserted in subject_list
static gint compare_incr(gconstpointer a, gconstpointer b){
  return *((const int *) a) - *((const int *) b);
}

// Used by file_list (in a WindowInfo struct) to compare the FileInfo
// elements; its usage ensures that the list is sorted in decreasing
// order.
static gint compare_decr(gconstpointer a, gconstpointer b){
  return ((const FileInfo *) b)->file_level - ((const FileInfo *) a)->
      file_level;
}


// Compares the inodes in a file_list (which is an element in a
// WindowInfo struct).
gint compare_inodes(gconstpointer a, gconstpointer b){
  return (((const FileInfo *) b)->inode) != (((const FileInfo *) a)->inode);
}


// FILE_OPEN_MONITOR functions
//////////////////////////////

gint is_backup_file(gconstpointer a, gconstpointer b){
  char *f1 = ((const FileInfo *) a)->file_name;
  char *f2 = ((const FileInfo *) b)->file_name;

  if(!strncmp(f2+1, f1, strlen(f1)) &&
     !(strlen(f2) <= 2 || f2[0] != '#' || f2[strlen(f2)-1] != '#')){
    // An emacs backup file of the form #file.txt# was found.
    ((FileInfo *) b)->file_level = ((FileInfo *) a)->file_level;
    return 0;
  }

  // returns 0 if an emacs backup file of the form file.txt~ was found.
  return !(!strcmp(((const FileInfo *) b)->file_name, ((const FileInfo *) a)->
      file_name) &&
            ((FileInfo *) a)->inode == -1);
}


// SET_FILE_LEVEL functions
///////////////////////////

// Auxiliary function for update_file_levels
static void set_file_level(gpointer d, gpointer user_data){
  FileInfo *fd = d;
  long *data = ((long *) user_data);
  if(*(data+1) == fd->inode)
    memcpy(&fd->file_level, data+2, sizeof(fd->file_level));
}
```

```
static void update_file_levels(gpointer key, gpointer value, gpointer
    user_data){
  WindowInfo *window_info = value;
  if(window_info->file_list){
    g_slist_foreach(window_info->file_list, &set_file_level, user_data);
    window_info->file_list = g_slist_sort(window_info->file_list, &
        compare_decr);
  }
}


// SENSOR_SERVER and SET_FILE_LEVEL function
/////////////////////////////////////////////

static void unmap_windows(gpointer key, gpointer value, gpointer user_data){
  WindowInfo *window_info = value;
  Display *display = user_data;

  if(window_info->is_mapped && window_info->file_list){
    int window_level = ((FileInfo *) window_info->file_list->data)->file_level
        ;
    int clearance_level = (subject_list) ? *((int *) subject_list->data) : 0;
    if(no_read_up && window_level > clearance_level ||
       no_read_down && window_level < clearance_level){
      window_info->is_mapped = 0;
      XUnmapWindow(display, window_info->window_id);
    }
  }
}


static void map_windows(gpointer key, gpointer value, gpointer user_data){
  WindowInfo *window_info = value;
  Display *display = user_data;

  if(!window_info->is_mapped && window_info->file_list){
    int window_level = ((FileInfo *) window_info->file_list->data)->file_level
        ;
    int clearance_level = *((int *) subject_list->data);

    if(!(no_read_up && window_level > clearance_level ||
         no_read_down && window_level < clearance_level)){
      window_info->is_mapped = 1;
      XMapWindow(display, window_info->window_id);
    }
  }
}


// LIST_WINDOW_INFO functions
//////////////////////////////

static void print_file_details(gpointer data, gpointer user_data){
  FileInfo *fd = data;
  int clearance_level = (subject_list) ? *((int *) subject_list->data) : 0;
  if(hide_non_readable_files &&
      ((no_read_up && fd->file_level > clearance_level) ||
       ((no_read_down && fd->file_level < clearance_level))))
    fprintf((FILE *) user_data, "\'unavailable' %10u\n", fd->file_level);
  else
    fprintf((FILE *) user_data, "%-15s %10u\n", fd->file_name, fd->file_level)
        ;
}

static void print_file_info(gpointer key, gpointer value, gpointer user_data){
  WindowInfo *window_info = value;
  if(window_info->file_list){
    fprintf((FILE *) user_data, "\nOpen files in window %i:\n", window_info->
        window_id);
    fprintf((FILE *) user_data, "%-15s %10s\n", "File Name", "Level");
    g_slist_foreach(window_info->file_list, &print_file_details, user_data);
  }
}

static void print_window_info(gpointer key, gpointer value, gpointer user_data
    ){  pid_t *pid = key;
 WindowInfo *window_info = value;
 int win_level = 0;
 if(window_info->file_list)
   win_level = ((FileInfo *) window_info->file_list->data)->file_level;
 fprintf((FILE *) user_data, "%-16s %6i %10i %18i %10i\n", window_info->
     app_name, *pid, window_info->window_id, win_level, window_info->is_mapped
     );
}


// LIST_SUBJECT_LEVELS function
////////////////////////////////

void print_subject_list(gpointer data, gpointer user_data){
  fprintf((FILE *) user_data, "%i ",  *((int *) data));
}


// DestroyNotify functions
//////////////////////////

static gboolean rm_win(gpointer key, gpointer value, gpointer user_data){
  Window *window_id = user_data;
  WindowInfo *window_info = value;
  return window_info->window_id == *window_id;
}


// DESTROY functions
////////////////////

void key_destroy_func(gpointer data){
  g_free(data);
}
```

```
void value_destroy_func(gpointer data){
  WindowInfo *window_info = data;
  g_free(window_info->app_name);
  g_slist_free(window_info->file_list);
  g_free(window_info);
}



int main(int argc, char *argv[]){

  // X11 initialisation
  /////////////////////

  Display *display;
  if(!(display = XOpenDisplay(NULL))){
    error2("ERROR opening display [%s]\n", XDisplayName(NULL));
    goto out;
  }

  XSetErrorHandler(x_error_handler);

  // Create the target window that the clients should send XEvents
  // to. This window is never mapped and the dimensions and other
  // arguments to XCreateSimpleWindow are arbitrary legal values.
  Window target_window = XCreateSimpleWindow(display, RootWindow(display,
     DefaultScreen(display)), 0,0,10,10,1,1, 1);

  if(target_window == (Window) None) {
    error("Error opening window.\n");
    XCloseDisplay(display);
    goto out;
  }

  Atom client_message = XInternAtom(display, "CLIENT_MESSAGE", False);
  Atom wm_name = XInternAtom(display, "WM_NAME", True);
  Atom net_wm_name = XInternAtom(display, "_NET_WM_NAME", True);
  Atom utf8_string = XInternAtom(display, "UTF8_STRING", True);

  // Initialize shared memory
  ////////////////////////////

  int shm_id = shmget(SHARED_MEMORY_KEY, getpagesize(),
                      IPC_CREAT // | IPC_EXCL
                      | 0666);
  if(shm_id == -1){
    perror("shmget");
    goto out;
  }

  void *shm = shmat(shm_id, 0, 0);
  if(!shm){
    perror("shmat");
    goto out;
  }
  // Store the target window id in the shared memory
  memcpy(shm+SECURITY_MANAGER_SHM_OFFSET, &target_window, sizeof(Window));


  // Initialize semaphore set
  ///////////////////////////

  int sem_id = semget(SEMAPHORE_KEY, NO_OF_SEMAPHORES, IPC_CREAT | 0666 ); //
      | IPC_EXCL
  if(sem_id == -1){
    perror("semid");
    goto out;
  }

  // Initialise all semaphore values to 1.
  unsigned short values[NO_OF_SEMAPHORES];
  int i = 0;
  for (; i < NO_OF_SEMAPHORES; i++)
    values[i] = 1;

  union semun sem_union;
  sem_union.array = values;
  if(semctl(sem_id, 0, SETALL, sem_union) == -1){
    perror("semctl");
    goto out;
  }


  // Read security policy options
  ////////////////////////////////

  if (argc != 2) {
    error2("Usage: %s POLICY_FILE\n", argv[0]);
    goto out;
  }

  int fd = open(argv[1], O_RDONLY);
  if(fd == -1){
    perror("open");
    goto out;
  }

  if(read_line(fd, HIDE_NON_READABLE_FILES, &hide_non_readable_files) < 0){
    perror("read");
    goto out;
  }

  if(read_line(fd, NO_READ_UP, &no_read_up) < 0){
    perror("read");
    goto out;
  }

  if(read_line(fd, NO_READ_DOWN, &no_read_down) < 0){
    perror("read");
    goto out;
  }
```

```
close(fd);


// Wait until a user has logged in
XEvent event;
int not_done = 1;
while(not_done){
  XNextEvent(display, &event); // Get the next event from the X Server.
  switch(event.xany.type){
  case ClientMessage:
    if(event.xclient.message_type == client_message)
      switch(event.xclient.data.l[0]){

      case INIT_CLEARANCE_LEVEL:{
        int cl_level = event.xclient.data.l[1];
        subject_list = g_slist_prepend(subject_list, &cl_level);
        not_done = 0;
        break;
      }

      case LIST_WINDOW_INFO:
      case LIST_SUBJECT_LEVELS:{
        char file_name[256];
        sprintf(file_name, FIFO_FILE, event.xclient.data.l[1]);

        FILE *fifo_file = fopen(file_name, "w");
        if(!fifo_file){
          perror("fopen");
          break;
        }

        fprintf(fifo_file, "Error: No one has logged in yet.\n");

        if(fclose(fifo_file))
          perror("fclose");
        break;
      }
      case DESTROY:
        goto out;

      default:
        error( "Error: No one has logged in yet.\n");
      }
  }
}

// Initialize the hash table
table = g_hash_table_new_full(hashFunc, equalFunc, key_destroy_func,
    value_destroy_func);

// Open a file descriptor to the stackable file system.
fd = open(MOUNT_POINT, O_RDONLY);
if (fd < 0) {
  perror("open");
  goto out;
```

```
}

not_done = 1;
// A user has now logged in, so other events can now be received.

XSelectInput(display, RootWindow(display, DefaultScreen(display)),
    SubstructureNotifyMask);

while(not_done){
  XNextEvent(display, &event); // Get the next event from the X Server.
  switch(event.xany.type){
  case ClientMessage:
    if(event.xclient.message_type == client_message)
      switch(event.xclient.data.l[0]){

      case INIT_CLEARANCE_LEVEL:
        error("Error: A user has already logged in.\n");
        break;

      case XCREATE_WINDOW_INTERCEPTOR:{
        Window window = event.xclient.data.l[2];
        unsigned char *data;
        Atom real_type;
        int real_format;
        unsigned long items_read, items_left;

        pid_t *pid = g_memdup(&event.xclient.data.l[1], sizeof(pid_t));
        WindowInfo *window_info = g_hash_table_lookup(table, pid);
        if((!window_info ||
            window_info && !strcmp("nedit",window_info->app_name)) &&
           XGetWindowProperty(display, window, wm_name,
                                          0, 0x7fffffff, False, XA_STRING,
                                            &real_type, &real_format, &
                                                items_read,
                                            &items_left, &data) == Success
           && items_read >= 1){
          if(DEBUG) printf("CREATE_WINDOW_INTERCEPTOR: pid = %i, window = %u
               \n", event.xclient.data.l[1], window);

          // A WindowInfo struct for the newly created window is
          // inserted into table.
          window_info = g_new(WindowInfo, 1);
          window_info->window_id = window;
          window_info->app_name = g_memdup(data, strlen(data)+1);
          window_info->file_list = NULL;
          window_info->is_mapped = 1;

          g_hash_table_insert(table, pid, window_info);

          // Notify the X server that we would like to receive an
          // 'DestroyNotify' XEvent when the window is destroyed.
          XSelectInput(display, window_info->window_id, StructureNotifyMask)
              ;
        }
        break;
      }
    case FILE_OPEN_MONITOR:{
      if(DEBUG) printf("FILE_OPEN_MONITOR: pid = %i, level = %i, inode = %
          ul\n", event.xclient.data.l[1], event.xclient.data.l[2], event.
          xclient.data.l[3]);

      WindowInfo *window_info = g_hash_table_lookup(table, &event.xclient.
          data.l[1]);

      if(window_info){
        // An editor has opened a file.

        FileInfo *f = g_new(FileInfo, 1);
        f->file_name = g_memdup(shm+FILE_OPEN_MONITOR_SHM_OFFSET, 256);
        f->file_level = event.xclient.data.l[2];
        f->inode = event.xclient.data.l[3];

        GSList *e = g_slist_find_custom(window_info->file_list, f,
            is_backup_file);

        if(e){ // A backfile was created
          struct _fist_ioctl_SET_FILE_LEVEL val;
          struct stat stat_buf;

          val.inode = f->inode;
          if(((FileInfo *) e->data)->inode == -1){
            val.level = ((FileInfo *) e->data)->file_level;
            ((FileInfo *) e->data)->inode = f->inode;
          }
          else
            val.level = ((FileInfo *) e->data)->file_level;

          // The file level of the backup file is updated in the
          // file system.
          if(ioctl(fd, FIST_IOCTL_SET_FILE_LEVEL, &val) < 0) {
            perror("ioctl");
            close(fd);
            break;
          }
          g_free(f);

          if(val.level == -1)
            error("The level could not be set.\n");
        }
        else if(!g_slist_find_custom(window_info->file_list, f,
                                      compare_inodes)){
          // The file is not already open in the editor.
          // Furthermore, the file is not a backup file, so it is
          // inserted in the list of open files.
          window_info->file_list = g_slist_insert_sorted(window_info->
              file_list, f, &compare_decr);
        }
      }

      if(semaphore_up(sem_id, FILE_OPEN_MONITOR_SEM_NUM) == -1)
        perror("semop");
    break;
  }

  case BACKUP_INTERCEPTOR:{
    if(DEBUG) printf("BACKUP_INTERCEPTOR: pid=%i, inode=%i,
        application ID = %i\n", event.xclient.data.l[1], event.xclient.
        data.l[2], event.xclient.data.l[3]);

    FileInfo *f = g_new(FileInfo, 1);
    f->inode = event.xclient.data.l[2];

    WindowInfo *window_info = g_hash_table_lookup(table, &event.xclient.
        data.l[1]);
    if(window_info && window_info->file_list)
      if(event.xclient.data.l[3] == EMACS_BACKUP &&
          (!strcmp(window_info->app_name, "emacs") ||
           !strcmp(window_info->app_name, "gedit"))){
        // Only backup files created by emacs and gedit can be handled.
        GSList *e = g_slist_find_custom(window_info->file_list, f,
                                         &compare_inodes);
        if(e)//Marking backup file
          ((FileInfo *) e->data)->inode = -1;
      }
    break;
  }

  case SENSOR_SERVER:{
    int env_level = event.xclient.data.l[1];
    char direction = event.xclient.data.l[2];
    struct _fist_ioctl_SET_CLEARANCE_LEVEL val;

    if(DEBUG) printf("SENSOR_SIMULATOR: environment level = %i,
        direction = %c\n", env_level, direction);

    if(direction == 'i'){ // A subject has entered the office

      if(!subject_list){ // No one was previously present in the
                         // environment.

        // The clearance level in the file system has to be set.
        val.level = env_level;
        if(ioctl(fd, FIST_IOCTL_SET_CLEARANCE_LEVEL, &val) < 0)
          perror("ioctl");

        // The level is inserted in the subject_list list.
        subject_list = g_slist_prepend(subject_list,
                                       g_memdup(&env_level, sizeof(
                                           env_level)));
        // Some windows will possibly have to be mapped.
        g_hash_table_foreach(table, map_windows, display);
      }
      else if(env_level < *((int *)subject_list->data)){
        // Someone was already present in the environment.

        // The clearance level in the file system must be updated.
```

```
            val.level = env_level;
            if(ioctl(fd, FIST_IOCTL_SET_CLEARANCE_LEVEL, &val) < 0)
              perror("ioctl");

            subject_list = g_slist_prepend(subject_list,
                                           g_memdup(&env_level, sizeof(
                                               env_level)));
            g_hash_table_foreach(table, unmap_windows, display);
          }
          else
            subject_list = g_slist_insert_sorted(subject_list,
                                                 g_memdup(&env_level, sizeof
                                                     (env_level)),
                                                 compare_incr);
        }
        else if(direction == 'o'){// A subject has left the office
          if(!subject_list){
            error("Error: No subjects are present.\n");
            break;
          }

          GSList *e = g_slist_find_custom(subject_list, &env_level,
              compare_incr);
          if(!e){
            error2("Error: there is no subject in the room with the
                clearance level %i.\n", env_level);
            break;
          }

          if(g_slist_length(subject_list) == 1){
            // No one is in the office now
            val.level = 0;
            if(ioctl(fd, FIST_IOCTL_SET_CLEARANCE_LEVEL, &val) < 0)
              perror("ioctl");

            // Removes the single element in the list.
            subject_list = g_slist_remove_link(subject_list, e);
            g_slist_free_1(e);

            g_hash_table_foreach(table, unmap_windows, display);
          }
          else{
            // At least one person is still present in the environment.
            int old_clearance_level = *((int *)subject_list->data);
            subject_list = g_slist_remove_link(subject_list, e);
            g_slist_free_1(e);

            if(env_level == old_clearance_level){
              // The lowest level is to be removed.
              // The clearance level in the kernel must be updated.
              val.level = *((int *) subject_list->data);

              if(ioctl(fd, FIST_IOCTL_SET_CLEARANCE_LEVEL, &val) < 0)
                perror("ioctl");

              g_hash_table_foreach(table, map_windows, display);
          }
        }
      }
      else error("Error: Invalid direction.\n");

      break;
    }

    case SET_FILE_LEVEL:{
      if(DEBUG) printf("SET_FILE_LEVEL: inode = %i, level = %i\n", event.
          xclient.data.l[1], event.xclient.data.l[2]);

      g_hash_table_foreach(table, &update_file_levels, event.xclient.data.
          l);

      if(!subject_list)
        break;

      // Change the visibility of windows, if required.
      if(event.xclient.data.l[2] <= *((int *)subject_list->data))
        g_hash_table_foreach(table, map_windows, display);
      else
        g_hash_table_foreach(table, unmap_windows, display);

      break;
    }


    case LIST_WINDOW_INFO:{
      if(DEBUG) printf("LIST_WINDOW_INFO\n");

      char file_name[256];
      sprintf(file_name, FIFO_FILE, event.xclient.data.l[1]);

      FILE *fifo_file = fopen(file_name, "w");
      if(!fifo_file){
        perror("fopen");
        break;
      }

      // Return the contents of the table to the client process
      // via a named pipe.
      if(g_hash_table_size(table)){
        fprintf(fifo_file, "Table content:\n\n");
        fprintf(fifo_file, "%-16s %6s %10s %18s %10s\n", "Application
            Name", "PID", "Window ID", "Security Level", "Is mapped");
        g_hash_table_foreach(table, &print_window_info, fifo_file);
        g_hash_table_foreach(table, &print_file_info, fifo_file);
      }
      else
        fprintf(fifo_file, "No windows are mapped.\n");

      if(fclose(fifo_file))
        perror("fclose");
      break;
    }
```

```
          case LIST_SUBJECT_LEVELS:{
            if(DEBUG) printf("LIST_SUBJECT_LEVELS\n");
            char file_name[256];
            sprintf(file_name, FIFO_FILE, event.xclient.data.l[1]);

            FILE *fifo_file = fopen(file_name, "w");
            if(!fifo_file){
              perror("fopen");
              break;
            }

            // Return the contents of the subject_list to the client
            // process via a named pipe.
            if(subject_list){
              g_slist_foreach(subject_list, &print_subject_list, fifo_file);
              fprintf(fifo_file, "\n");
            }
            else
              fprintf(fifo_file, "No subjects are in the room.\n");

            if(fclose(fifo_file))
              perror("fclose");
            break;
          }

          case DESTROY: {
            not_done = 0; // Breaks the loop.
            break;
          }

          default:
            error("Unknown client message.\n");
          }
        break;

      case DestroyNotify:
        if(event.xdestroywindow.event == event.xdestroywindow.window){
          if(DEBUG) printf("DestroyNotify: window = %u\n", event.xdestroywindow.
              event, event.xdestroywindow.send_event);

          g_hash_table_foreach_remove(table, rm_win, &event.xdestroywindow.
              window);
        }
        break;
      }
    }

  out:
    if(DEBUG) printf("Closing the visibility manager.\n");
    close(fd);
    semctl(sem_id, 0, IPC_RMID);

    shmdt(shm);
    shmctl(shm_id, IPC_RMID, 0);
  XDestroyWindow(display, target_window);
  XCloseDisplay(display);
}
```

F.5.2 file_open_monitor.c

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdio.h>

#include "seac_ipc.h"
#include "mount_point.h"
#include <wrapfs.h>

int main(){

  struct _fist_ioctl_OPEN val;
  int status = 0;

  // Initialize semaphore
  int sem_id = semget(SEMAPHORE_KEY, 0, 0);
  if(sem_id == -1){
    perror("semid");
    exit(1);
  }

  // Initialize security_manager shared memory
  int shm_id = shmget(SHARED_MEMORY_KEY, getpagesize(), 0);
  if(shm_id == -1){
    perror("shmget");
    exit(1);
  }

  void *shm = shmat(shm_id, 0, 0);
  if(!shm){
    perror("shmat");
    exit(1);
  }

  // Get a file descriptor to the stackable file system.
  int fd = open(MOUNT_POINT, O_RDONLY);
  if(fd < 0) {
    perror("open");
    exit(1);
  }

  Window target_window;
  memcpy(&target_window, shm+SECURITY_MANAGER_SHM_OFFSET, sizeof(Window));
  long data[] = {FILE_OPEN_MONITOR, 0, 0, 0, 0};
  if(!target_window){
    printf("Error: the target window has not been initialized.\n");
    exit(1);
  }

  while(1){
    // Block until a new file is opened.
    status = ioctl(fd, FIST_IOCTL_OPEN, &val);

    if(status < 0) {
      perror("Could not access file system");
      break;
    }

    if(val.level < 0)
      break; // The destroy program has been invoked, so the file_open
             // monitor should stop running.

    // Wait until the shared memory can be written, i.e. when the
    // consumer (the visibility_manager) is ready to read the shared
    // memory.
    status = semaphore_down(sem_id, FILE_OPEN_MONITOR_SEM_NUM);
    if(status == -1){
      perror("semop");
      break;
    }

    // Write to the shared memory.
    strcpy(shm+FILE_OPEN_MONITOR_SHM_OFFSET, val.name);
    data[1] = val.pid;
    data[2] = val.level;
    data[3] = val.inode;

    // Notify the visibility manager that the memory can be read.
    if(send_xclient_event(data, 0, target_window) < 0)
      fprintf(stderr, "Error: file_open_monitor could not send message to
          visibility_manager.\n");
  }

  shmdt(shm);
  close(status);
  exit(0);
}
```

F.5.3 sensor_server.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/shm.h>
```

```
#include "seac_ipc.h"
#include "sensor.h"


#define QUEUE_SIZE 5

int main(int argc, char* argv[]){

  int hSocket, hServerSocket;  // handle to socket
  struct hostent* pHostInfo;   // holds info about a machine
  struct sockaddr_in Address; // Internet socket address stuct
  int nAddressSize=sizeof(struct sockaddr_in);
  int nHostPort = DEFAULT_PORT;

  if (argc > 2) {
    fprintf(stderr, "Usage: %s [PORT]\n", argv[0]);
    exit(1);
  }

  if(argc == 2)
    nHostPort = atoi(argv[1]);

  hServerSocket=socket(AF_INET,SOCK_STREAM,0);
  if(hServerSocket == -1){
    perror("socket");
    exit(1);
  }

  // fill address struct
  Address.sin_addr.s_addr=INADDR_ANY;
  Address.sin_port=htons(nHostPort);
  Address.sin_family=AF_INET;

  if(bind(hServerSocket,(struct sockaddr*)&Address,sizeof(Address)) == -1){
    perror("bind");
    exit(1);
  }

  //  get port number
  getsockname( hServerSocket, (struct sockaddr *) &Address,(socklen_t *)&
      nAddressSize);

  if(listen(hServerSocket,QUEUE_SIZE) == -1){
    perror("listen");
    exit(1);
  }

  // Initialize security_manager shared memory
  int shm_id = shmget(SHARED_MEMORY_KEY, getpagesize(), 0);

  if(shm_id == -1){
    perror("shmget");
    exit(1);
  }
  void *shm = shmat(shm_id, 0, SHM_RDONLY);
  if(!shm){
    perror("shmat");
    exit(1);
  }

  Window target_window;
  memcpy(&target_window, shm+SECURITY_MANAGER_SHM_OFFSET, sizeof(Window));
  shmdt(shm);

  if(!target_window){
    printf("Error: the target window has not been initialized.\n");
    close(hServerSocket);
    exit(1);
  }

  int err = 0;
  while(!err){
    int env_level;
    char direction;

    hSocket=accept(hServerSocket,(struct sockaddr*)&Address,(socklen_t *)&
        nAddressSize);

    read(hSocket, &env_level, sizeof(env_level));
    read(hSocket, &direction, sizeof(direction));

    if(env_level < 0){
      err = 1;
     }
    else
      if(direction != 'i' && direction != 'o')
        err = -1;
      else{
        long data[] = {SENSOR_SERVER, env_level, direction, 0, 0};
        send_xclient_event(data, 0, target_window);
      }

    write(hSocket, &err, sizeof(err));
    if(close(hSocket) == -1)
      perror("close");
  }

  if(close(hServerSocket) == -1){
    perror("close");
    exit(1);
  }

  exit(0);
}
```

F.5.4 listwl.c

```
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
#include <fcntl.h>
#include <dirent.h>
#include <stdio.h>
#include "seac_ipc.h"

int main(){

  char file_name[MAXNAMLEN+1];
  sprintf(file_name, FIFO_FILE, getpid());

  if(mkfifo(file_name, 0600 | O_CREAT | O_EXCL) == -1){
    perror("mkfifo");
    exit(1);
  }

  // Notify the sm that it can write to the new fifo.
  long data[] = {LIST_WINDOW_INFO, getpid(), 0, 0, 0};
  send_xclient_event(data, 0, 0);

  // Read the data from the fifo
  FILE *fifo_file = fopen(file_name, "r");
  if(!fifo_file){
    perror("fopen");
    exit(1);
  }

  char buf[MAXNAMLEN+1];
  while(!feof(fifo_file) && !ferror(fifo_file) && fgets(buf, sizeof(buf),
      fifo_file))
    fputs(buf, stdout);

  if(fclose(fifo_file)){
    perror("fclose");
    exit(1);
  }

  remove(file_name);
}
```

F.5.5 listsl.c

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <dirent.h>
#include <fcntl.h>

#include "seac_ipc.h"

int main(){

  char file_name[MAXNAMLEN+1];
  sprintf(file_name, FIFO_FILE, getpid());
  if(mkfifo(file_name, 0600 | O_CREAT | O_EXCL) == -1){
    perror("mkfifo");
    exit(1);
  }

  // Notify the sm that it can write to the new fifo.
  long data[] = {LIST_SUBJECT_LEVELS, getpid(), 0, 0, 0};
  send_xclient_event(data, 0, 0);

  // Read the data from the fifo
  FILE *fifo_file = fopen(file_name, "r");
  if(!fifo_file){
    perror("fopen");
    exit(1);
  }

  char buf[MAXNAMLEN+1];
  while(!feof(fifo_file) && !ferror(fifo_file) && fgets(buf, sizeof(buf),
      fifo_file))
    fputs(buf, stdout);

  if(fclose(fifo_file)){
    perror("fclose");
    exit(1);
  }

  remove(file_name);
}
```

F.5.6 getcl.c

```
#include <sys/ioctl.h>
#include <fcntl.h>

#include "mount_point.h"
#include "seac_ipc.h"

#include <wrapfs.h>

int main(){

  struct _fist_ioctl_GET_CLEARANCE_LEVEL val;

  int fd = open(MOUNT_POINT, O_RDONLY);
  if (fd < 0) {
    perror("open");
    exit(1);
  }

  // Get the clearance level from the file system.
  int status = ioctl(fd, FIST_IOCTL_GET_CLEARANCE_LEVEL, &val);
  if (status < 0)
    perror("Could not access file system");
  else
    printf("%i\n", val.level);

  close(fd);
  exit(status);
}
```

F.6 Editor Files

F.6.1 x_create_window_interceptor.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <dlfcn.h>
#include "seac_ipc.h"

#ifndef libX11_PATH
#define libX11_PATH "/usr/X11R6/lib/libX11.so"
#endif

//XCreateWindow creates an unmapped subwindow for the specified parent
//window and returns the window ID of the created window.
static Window (*original_XCreateWindow) (Display *display, Window parent, int
    x, int y, unsigned int width, unsigned int height, unsigned int
    border_width, int depth, unsigned int class, Visual *visual, unsigned long
     valuemask, XSetWindowAttributes *attributes);

static Window (*original_XCreateSimpleWindow) (Display *display, Window parent
    , int x,  int y, unsigned int width,  unsigned int height, unsigned int
    border_width, unsigned long border, unsigned long background);


Window XCreateWindow(Display *display,
                     Window parent,
                     int x,
                     int y,
                     unsigned int width,
                     unsigned int height,
                     unsigned int border_width,
                     int depth,
                     unsigned int class,
                     Visual *visual,
                     unsigned long valuemask,
                     XSetWindowAttributes *attributes){

  void *handle = dlopen(libX11_PATH, RTLD_LAZY);
  if (!handle) {
    fputs(dlerror(), stderr);
    return 0;
  }

  char *error;
  original_XCreateWindow = dlsym(handle, "XCreateWindow");
  if ((error = dlerror()) != NULL)  {
    fputs(error, stderr);
    return 0;
  }

  Window window = (*original_XCreateWindow) (display, parent, x, y,
                                              width, height,
                                              border_width, depth,
                                              class, visual,
                                              valuemask, attributes);

  if(parent == RootWindow(display, DefaultScreen(display))){
    long data[] = {XCREATE_WINDOW_INTERCEPTOR, getpid(), window, 0, 0};
    send_xclient_event(data, display, 0);
  }

  return window;
}


Window XCreateSimpleWindow(Display *display,
                           Window parent,
                           int x,
                           int y,
                           unsigned int width,
                           unsigned int height,
                           unsigned int border_width,
                           unsigned long border,
                           unsigned long background){

  void *handle =  dlopen(libX11_PATH, RTLD_LAZY);
  if (!handle) {
    fputs (dlerror(), stderr);
    return 0;
  }

  char *error;
  original_XCreateSimpleWindow = dlsym(handle, "XCreateSimpleWindow");
  if ((error = dlerror()) != NULL)  {
    fputs(error, stderr);
    return 0;
  }

  Window window = (*original_XCreateSimpleWindow) (display, parent, x, y,
                                                    width, height,
                                                    border_width, border,
                                                    background);

  if(parent == RootWindow(display, DefaultScreen(display))){
    long data[] = {XCREATE_WINDOW_INTERCEPTOR, getpid(), window, 0, 0};
    send_xclient_event(data, display, 0);
  }

  return window;
}
```

F.6.2 backup_interceptor.c

```c
#include <sys/types.h>
#include <sys/stat.h>

#include <stdio.h>
#include <dlfcn.h>
#include "seac_ipc.h"
#include "mount_point.h"

#ifndef libc_PATH
#define libc_PATH "/lib/tls/libc.so.6"
#endif

static int (*original_rename)(const char *oldpath, const char *newpath);

int rename(const char *oldpath, const char *newpath){

  char *error;
  void *handle = dlopen(libc_PATH, RTLD_LAZY);

  if (!handle) {
    fputs (dlerror(), stderr);
    return -1;
  }

  original_rename = dlsym(handle, "rename");
  if ((error = dlerror()) != NULL)  {
    fputs(error, stderr);
    return -1;
  }

  long data[] = {BACKUP_INTERCEPTOR, getpid(), 0, 0, 0};
  struct stat stat_buf;

  if(stat(oldpath, &stat_buf) < 0){
    perror("stat");
    return 0;
  }
  data[2] = stat_buf.st_ino;

  int status = (*original_rename) (oldpath, newpath);

  if(!oldpath || strncmp(MOUNT_POINT, oldpath, strlen(MOUNT_POINT)))
    return 0;

  size_t old_size = strlen(oldpath);
  size_t new_size = strlen(newpath);
  if(old_size + 1 == new_size &&
     !strncmp(oldpath, newpath, old_size) &&
     newpath[new_size-1] == '~'){
    data[3] = EMACS_BACKUP;
    send_xclient_event(data, 0, 0);
  }

  return 0;
}
```

F.7 Sensor Files

F.7.1 swsensor.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

#include "sensor.h"

#define HOST_NAME_SIZE      255

int  main(int argc, char* argv[]){
  int hSocket;                 // handle to socket
  struct hostent* pHostInfo;   // holds info about a machine
  struct sockaddr_in Address;  // Internet socket address stuct
  long nHostAddress;
  unsigned nReadAmount;
  char strHostName[HOST_NAME_SIZE];
  int nHostPort = DEFAULT_PORT;

  if (argc < 3 || argc > 5) {
    fprintf(stderr, "Usage: %s LEVEL DIRECTION [HOST] [PORT]\n", argv[0]);
    exit(1);
  }

  int env_level = atoi(argv[1]);
  char direction = argv[2][0];

  if(argc == 4)
    if(sizeof(argv[3]) >= sizeof(strHostName)){
      fprintf(stderr, "Error: the host name must be less than %i characters.\n
          ", sizeof(strHostName));
      exit(1);
    }
    else
      strcpy(strHostName, argv[3]);
  else
    strcpy(strHostName, DEFAULT_HOST);

  if (argc == 5)
    nHostPort = atoi(argv[4]);

  hSocket=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
```

```c
  if(hSocket == -1){
    perror("socket");
    printf("\nCould not make a socket\n");
    exit(1);
  }

  // get IP address from name
  pHostInfo=gethostbyname(strHostName);
  memcpy(&nHostAddress,pHostInfo->h_addr,pHostInfo->h_length);

  // fill address struct
  Address.sin_addr.s_addr=nHostAddress;
  Address.sin_port=htons(nHostPort);
  Address.sin_family=AF_INET;

  if(connect(hSocket,(struct sockaddr*)&Address,sizeof(Address)) == -1){
    perror("connect");
    exit(1);
  }

  write(hSocket, &env_level, sizeof(env_level));
  write(hSocket, &direction, sizeof(direction));

  int status;
  read(hSocket, &status, sizeof(status));

  if(status == -1)
    fprintf(stderr, "Error: \'%c\' is an invalid direction.\n", direction);

  if(close(hSocket) == -1){
    perror("close");
    printf("\nCould not close socket\n");
  }

  exit(0);
}
```

F.7.2 motion_handler.c

```c
/* motion_handler.c */
/* Date: 04/05-2004 */
/* Author: Ida */
/* last modified: 04/08-2004 */

/* This is the program that collects events (pictures taken) from Motion */
/* It recieve notification when a snapshot has been taken */
/* It will then determine if motion have been detected or not, */
/* based on how many snapshots have been taken */
/* this includes communication of results. */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>  //for pipes
```

```
#include <sys/types.h> //for pipes
#include <sys/stat.h> //for pipes
#include <pthread.h> //for threads
#include <semaphore.h> //for threads
#include <time.h> // for timestamps mv
#include <glib.h> //for linked list
#include <signal.h> //for termination handeling.

#include "pipe2.h" //length of pipes

#define MIN_MOTION_NO 10 // no of pics needed for motion to be detected
#define MAX_MOTION_DIFF 5 //sec of difference for motion - for all. ie
    dependent on above value.
#define NEW_MOTION_DIFF 4  //sec between old a new motion. - from newest!
//NB the headtime then becomes last time

void *eventReciever(void *arg); //the function the thread will call.
// Prototype. Takes a void pointer as arg, and returns a pointer to void.

GSList *eventList; //The list of events
pthread_mutex_t listMutex = PTHREAD_MUTEX_INITIALIZER; //mutex to protect the
    list.
sem_t listSem; //semaphore for the list
char *pipename[PIPE_NAME_LENGTH];
char *campipe[PIPE_NAME_LENGTH];

//aux function to print the GSList
static void printElement(gpointer value, gpointer user_data)
{
  time_t *time = value;
  printf("Timestamp  =%i \n", *time);
}

//function that prints the GSList.
void print()
{
   if(g_slist_length(eventList) > 0)
    {
      printf("------------------------\n");
      printf("List content:\n");
      printf("------------------------\n");
      g_slist_foreach(eventList, &printElement, NULL);
      printf("------------------------\n");
    }
  else
    printf("The List is empty.\n");
}

//termination function
void end(int sig)
{
  printf("Terminating the motion_handler: %s\n", *pipename);
  int no = g_slist_length(eventList);
  int j;
  for(j = 0; j < no; j++)
    {
      eventList = g_slist_delete_link(eventList, eventList);
      // removes & deallocates the first element.
    }
  exit(0);
}

//MAIN
int main(int argc, char *argv[])
{

  if (argc < 3)
    //argv[0] = prog name, argv[1 to argc-1] = arguments
    {
       printf("Usage: %s PIPE-NAME PIPE-CAM-NAME\n", argv[0]);
       exit(1);
    }

  //for civilized termination:
  struct sigaction act;
  act.sa_handler = end;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, NULL);
  sigaction(SIGTERM, &act, NULL);

  int res;
  pthread_t event_recieving_thread;
  pthread_attr_t attr; //attributes of the thread

  *pipename = argv[1];
  *campipe = argv[2];

  printf("motion_handler starting: %s\n", *pipename);

  int no = 0;
  int j; //counter
  time_t lasttime = 0; //the time for the last pic in the last mostion
      detected sequence.
  time_t *pStarttime = g_new(time_t, 1);
  time_t *pEndtime = g_new(time_t, 1);

  int diff = 0; //difference between first and last pic
  int gap = 0; //difference between last sequence of pics and new seq

  struct tm *pNow=NULL; //for nice printing of time
  char Buffer[100]; //for nice printing of time

  sem_init(&listSem, 0, 0); //init the semaphore.

  //initialising the attributes with default
  pthread_attr_init(&attr);
  //setting the specialised attribute
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  //creating thread
  res = pthread_create(&event_recieving_thread, &attr, eventReciever, NULL);
  if(res != 0)
```

```
    {
      perror("MOTION_HANDLER: thread could not be created");
      return -1;
    }

  //destroying thread attribute object.
  pthread_attr_destroy(&attr);

  /********* MAIN LOOP *********/
  //waiting for a sequence of events to begin
  while(1)
    {
      sem_wait(&listSem); //waiting for soemthng to happen
      pthread_mutex_lock(&listMutex);
      no = g_slist_length(eventList);

      if(no > MIN_MOTION_NO) //if there is enough pictures
        {
          pStarttime = g_slist_nth_data(eventList, 0);
          pEndtime = g_slist_nth_data(eventList, no-1);
          pthread_mutex_unlock(&listMutex);

          diff = difftime(*pEndtime,*pStarttime);

          if(diff < MAX_MOTION_DIFF ) //if they are taken close enough
            {
              gap = *pStarttime - lasttime;

              if(gap > NEW_MOTION_DIFF) //if they are far enough from the last
                  motion detected.
                {
                  /*
                  // Printing of detection
                  pNow = localtime(pStarttime);
                  strftime(Buffer, sizeof(Buffer),"Motion detected on %d %m %Y
                       at %H.%M.%S",pNow);
                  printf("***************************************************\
                      n");
                  printf("%s:%s\n", Buffer, *pipename);
                  printf("***************************************************\
                      n");
                  */

                  //open the pipe
                  int camp = open(*campipe, O_WRONLY);
                  //send the time
                  write(camp, pStarttime, sizeof(pStarttime));
                  //close the pipe
                  if (camp != -1)
                    (void) close(camp);

                  lasttime = *pEndtime;

                  //remove all from 0 to no-1:
                  for(j = 0; j < no; j++)
                    {
                        pthread_mutex_lock(&listMutex);
                        eventList = g_slist_delete_link(eventList, eventList);
                        pthread_mutex_unlock(&listMutex);
                        // removes& deallocates the first element.
                      }
                    /*
                       NB: as we already know the original No,
                       we do not want to remove items put on the list after that
                       but as we do not update nN, we do not remove new items.
                       here we just make sure they can be added to the list
                           while remmoving.
                    */
                  }
                else //if they are part of the previouse motion sequence
                  {
                    //remove all that are part of the previous seq.
                    while((gap <= NEW_MOTION_DIFF) && (g_slist_length(eventList)
                        > 0))
                      {
                        //setting a new last time
                        pthread_mutex_lock(&listMutex);
                        pEndtime = g_slist_nth_data(eventList, 0);
                        lasttime = *pEndtime;
                        //removing & deallocating
                        eventList = g_slist_delete_link(eventList, eventList);

                        //calculating next gap
                        if(g_slist_length(eventList) > 0)
                          {
                            pStarttime = g_slist_nth_data(eventList, 0);
                            gap = *pStarttime - lasttime;
                          }
                        pthread_mutex_unlock(&listMutex);
                      }
                  }
              }
            else //not close enough
              {
                pthread_mutex_lock(&listMutex);
                //remove oldest and try again
                eventList = g_slist_delete_link(eventList, eventList);
                pthread_mutex_unlock(&listMutex);
              }
          }
        else
          {
            pthread_mutex_unlock(&listMutex); //as we locked it to read no.
          }
      }//end while(1)

  return 0;

}//end main()

//the second thread of execution
//the event listener
```

```
void *eventReciever(void *arg)
{
  int comm;          //the pipe
  char event = '0'; //the event recieved over the pipe.

  //check if the pipe exists.
  if(access(*pipename, F_OK)==-1)
    {
      //create the pipe
      comm = mkfifo(*pipename, 0777);
      //checking to see if the pipe could be created...
      if (comm != 0)
        {
          fprintf(stderr, "MOTION_HANDLER: Could not create fifo");
          return((void*)-1); //this is moot, since thread is detached
        }
    }

  time_t *pTime;

  while(1)
    {
      comm = open(*pipename, O_RDONLY);
      read(comm, &event, 1);

      if (comm != -1)
        {
          pTime = g_new(time_t, 1);
          *pTime = time(NULL);
          (void) close(comm);
          pthread_mutex_lock(&listMutex);
          eventList = g_slist_append(eventList, pTime);
          sem_post(&listSem); //post that a new event was recieved.
          pthread_mutex_unlock(&listMutex);
        }
    }

  return NULL; //detached thread. will never be used.

}//end eventReciever.
```

F.7.3 camera_client.c

```
/* camera_client.c */
/* Date: 28/04-2004 */
/* Author: Ida */
/* last modified: 29/08-2004 */

/* This is the program that communicates with the sensor_server */
/* It will pick up information from the motion_handler, */
/* and determine if the office was entered or left */
/* Then it will send that information to the sensor_server*/
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <fcntl.h>  //for pipes
#include <glib.h> //for g_new
#include "pipe2.h"
#include <signal.h> //for termination signal handeling

#define SOCKET_ERROR        -1
#define MAX_PASSING_TIME 15 //the time it takes to pass by the 2 cams

int hSocket; //handle to socket

int createPipe(char pipename[])
{
  if(access(pipename, F_OK)==-1) //check if the pipe exists.
    {
      //create the pipe
      int comm = mkfifo(pipename, 0777);
      //checking to see if the pipe could be created...
      if (comm != 0)
        {
          fprintf(stderr, "CAMERA_CLIENT: Could not create fifo:%s\n",
             pipename);
          return -1;
        }
    }
  return 0;
}

void readPipe(char pipename[], time_t *ptime)
{
  int comm = open(pipename, O_RDONLY);
  read(comm, ptime, sizeof(ptime));
  if (comm != -1)
    {
      (void) close(comm);
    }
}

//termination function
void end(int sig)
{
  printf("Terminating the camera_client\n");
  //closing socket
  if(close(hSocket) == SOCKET_ERROR)
        {
          fprintf(stderr,"\nCAMERA_CLIENT: Socket already closed\n");

        }
  exit(0);
```

```
}

int main(int argc, char *argv[])
{
  if (argc < 4)
    //argv[0] = prog name, argv[1 to argc-1] = arguments
      {
        fprintf(stderr,"Usage: %s INTURDER-LEVEL HOST-NAME PORT\n", argv[0]);
        exit(1);
      }

  int res; //pipe creating result
  time_t *ptime1 = g_new(time_t, 1);
  time_t *ptime2 = g_new(time_t, 1);
  struct tm *pNow=NULL; //for nice printing of the time
  char Buffer[100]; //for nice printing of the time

  char *host_name = argv[2]; //host name for delivering results
  int port = atoi(argv[3]); //socket name for delivering results
  //int hSocket; //handle to socket
  struct hostent* pHostInfo;   /* holds info about a machine */
  struct sockaddr_in Address;  /* Internet socket address struct */
  long nHostAddress;

  int intruder = atoi(argv[1]);  //the level a intruder detected by the
     cameras will be assigned.
  char direction; //entered='i' or exited='o'


  //for civilized termination:
  struct sigaction act;
  act.sa_handler = end;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, NULL);
  sigaction(SIGTERM, &act, NULL);




  //get IP address from name
  pHostInfo=gethostbyname(host_name);
  // copy address into long
  memcpy(&nHostAddress,pHostInfo->h_addr,pHostInfo->h_length);
  //fill address struct
  Address.sin_addr.s_addr=nHostAddress;
  Address.sin_port=htons(port);
  Address.sin_family=AF_INET;

  //creating pipes
  printf("-------\ncamera_client starting.\nIntruder level: %i.\nHostname: %s
     .\nPort no: %i\n-------\n", intruder, host_name, port);

  if (res = createPipe(PIPE_CAM_1) != 0)
    return res;
```

```
if  (res = createPipe(PIPE_CAM_2) != 0)
  return res;

while(1)
  {

    readPipe(PIPE_CAM_1, ptime1);
    readPipe(PIPE_CAM_2, ptime2);
    int diff = difftime(*ptime1,*ptime2);

    while( diff > MAX_PASSING_TIME)
      {
        if(*ptime1 < *ptime2)
          {
            readPipe(PIPE_CAM_1, ptime1);
            diff = difftime(*ptime1,*ptime2);
          }
        else
          {
            readPipe(PIPE_CAM_2, ptime2);
            diff = difftime(*ptime1,*ptime2);
          }
      }

    if(*ptime2 > *ptime1)
      {
        //someone entered
        direction = 'i';
        //print to screen:
        pNow = localtime(ptime1);
        strftime(Buffer, sizeof(Buffer),"CAMERA_CLIENT: Someone entered on %
            d %m %Y at %H.%M.%S",pNow);
        printf("%s\n",Buffer);
      }
    else //time1 > time2
      {
        //set exit
        direction = 'o';
        //print to screen
        pNow = localtime(ptime2);
        strftime(Buffer, sizeof(Buffer),"CAMERA_CLIENT: Someone exited on %d
             %m %Y at %H.%M.%S",pNow);
        printf("%s\n",Buffer);
      }



    //make the socket
    hSocket=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

    if(hSocket == SOCKET_ERROR)
      {
        fprintf(stderr,"\nCAMERA_CLIENT: Could not create a socket\n");
      }
    else
      {
          // connect to host
          if(connect(hSocket,(struct sockaddr*)&Address,sizeof(Address)) ==
             SOCKET_ERROR)
            {
              fprintf(stderr, "\nCAMERA_CLIENT: Could not connect to host %s\n
                 ", host_name);
            }

          else
            {
              //write to socket
              write(hSocket, &intruder, sizeof(intruder));
              write(hSocket, &direction, sizeof(direction));

              //umm close the socket, ya?
              if(close(hSocket) == SOCKET_ERROR)
                {
                  fprintf(stderr,"\nCAMERA_CLIENT: Could not close socket on
                     host %s\n",host_name);
                }

            }
        }
    }

  return 0;
}
```

F.7.4 event1.c

```
/* event1.c */
/* Date: 21/04-2004 */
/* Author: Ida */
/* last modified: 21/03-2004 */

/* This is a small program that should run every time Motion takes a pic */
/* It will then send the events to the eventhandelr */
/* So far, named pipes  are used for communication */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "pipe2.h"

int main(char *argv[])
{

  char flag = '1';
  char *pflag = &flag;

  //open the pipe
  //printf("event opening pipe\n");
  int comm = open(PIPE_NAME_1, O_WRONLY);

  //send the flag
  //printf("event writing %c to pipe \n", flag);
   write(comm, pflag, 1);

  //close the pipe
   if (comm != -1)
    (void) close(comm);

  return 0;
}
```

F.7.5 event2.c

```
/* event2.c */
/* Date: 21/04-2004 */
/* Author: Ida */
/* last modified: 21/03-2004 */

/* This is a small program that should run every time Motion takes a pic */
/* It will then send the events to the eventhandelr */
/* So far, named pipes  are used for communication */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "pipe2.h"

int main(char *argv[])
{

  char flag = '1';
  char *pflag = &flag;

  //open the pipe
  //printf("event opening pipe\n");
  int comm = open(PIPE_NAME_2, O_WRONLY);

  //send the flag
  //printf("event writing %c to pipe \n", flag);
   write(comm, pflag, 1);

  //close the pipe
   if (comm != -1)
    (void) close(comm);

  return 0;
}
```

F.7.6 pipe2.h

```
/* pipe2.h */
/* Date: 21/04-2004 */
/* Author: Ida */
/* last modified: 21/04-2004 */

/* contains the pipe names used for communicating */

#define PIPE_NAME_1 "/tmp/test1"
#define PIPE_NAME_2 "/tmp/test2"
#define PIPE_CAM_1 "/tmp/cam1"
#define PIPE_CAM_2 "/tmp/cam2"
#define PIPE_NAME_LENGTH 15
```

F.7.7 start_motion.c

```
/* start_motion.c */
/* Date: 21/04-2004 */
/* Author: Ida */
/* last modified: 04/08-2004 */

/* This is a program that starts the two different motionhandlers, */
/* one for each camera. */
/* uses fork */

#include <stdlib.h>
#include <stdio.h>

#include "pipe2.h"

int main()
{
  pid_t pid;

  pid = fork();

  if(pid != 0)
    {
      execlp("motion_handler","motion_handler",PIPE_NAME_1,PIPE_CAM_1, 0);
      fprintf(stderr, "START_MOTION: an error occcured in parent\n");
    }
  else
    {
      execlp("motion_handler","motion_handler",PIPE_NAME_2, PIPE_CAM_2, 0);
      fprintf(stderr, "START_MOTION: an error occcured in child\n");
    }

  return 0;
}
```

F.8 GUI Files

F.8.1 Exec.java

```java
import java.util.*;
import java.io.*;

class Exec{

    static Runtime rt = Runtime.getRuntime();

    static void getWindowInfo(Vector data, Hashtable details, boolean mapped){
        try{
            Process p = rt.exec("listwl");

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));
            StringTokenizer st;

            String line = in.readLine();
            if(line == null)
                return;

            line = in.readLine();
            line = in.readLine();

            // Read window data
            for(line = in.readLine(); line != null && !line.equals(""); line
                = in.readLine()){
                st = new StringTokenizer(line);
                Object[] s = {st.nextToken(), new Integer(st.nextToken()),
                              new Integer(st.nextToken()), new Integer(st.
                                  nextToken()), new Integer(0)};
                boolean isMapped = (Integer.parseInt(st.nextToken()) == 1);
                if(isMapped == mapped)
                    data.add(s);
            }

            for(line = in.readLine(); line != null; line = in.readLine()){
                int idx1 = line.lastIndexOf(' ');
                int idx2 = line.lastIndexOf(':');
                Integer windowID = new Integer(line.substring(idx1+1, idx2));
                Vector v = new Vector();
                // Read file data
                line = in.readLine();
                for(line = in.readLine(); line != null && !line.equals("");
                    line = in.readLine()){
                    st = new StringTokenizer(line);
                    v.add(new Object[]{st.nextToken(), new Integer(st.
                        nextToken())});
                }
                details.put(windowID, v);
            }

            // Read window data
            for(Enumeration e = data.elements(); e.hasMoreElements();){
                Object[] s = (Object[]) e.nextElement();
                Object files = details.get(s[2]);
                if(files != null){
                    int noOfOpenFiles = ((Vector) files).size();
                    if(noOfOpenFiles > 0)
                        s[4] = new Integer(noOfOpenFiles);
                }
            }
            p.destroy();
        }
        catch(Exception e){
            System.err.println(e);
        }
    }

    static int getClearanceLevel(){
        try{
            Process p = rt.exec("getcl");

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));
            String line = in.readLine();
            p.destroy();
            return Integer.parseInt(line);
        }
        catch(Exception e){
            System.err.println(e);
        }
        return -1;
    }

    static Integer getFileLevel(String filename){
        String line = null;
        try{
            Process p = rt.exec("getfl "+filename);

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));

            line = in.readLine();
            p.destroy();
            return new Integer(line);
        }
        catch(NumberFormatException e){
            System.err.println(e);
            return new Integer(-1);
        }
        catch(Exception e){
            System.err.println(e);
        }

        return null;
    }

    static String setFileLevel(String filename, Integer level){
        try{
            Process p = rt.exec("setfl "+filename+" "+level);

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));
            String line = in.readLine();
            p.destroy();
            return line;
        }
        catch(Exception e){
            System.err.println(e);
        }
        return null;
    }


    static String getFileLevels(Vector data, String dirname){
        String line = null;
        try{
            Process p = rt.exec("listfl "+dirname);
            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));
            StringTokenizer st;
            for(line = in.readLine(); line != null; line = in.readLine()){
                st = new StringTokenizer(line);
                Object[] s = {st.nextToken(), new Integer(st.nextToken())};
                data.add(s);
            }
            p.destroy();
        }
        catch(NumberFormatException e){
            System.err.println(e);
            return "Error: file selection is not supported in this directory."
                ;
        }
        catch(Exception e){
            System.err.println(e);
        }
        return null;
    }


    static String setUserLevel(String uid, Integer level){
        try{
            Process p = rt.exec("setul "+uid+" "+level);

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));
            String line = in.readLine();
            p.destroy();
            return line;
        }
        catch(Exception e){
            System.err.println(e);
        }
        return null;
    }


    static Integer getUserLevel(){
        try{
            Process p = rt.exec("getul");

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));

            String line = in.readLine();
            p.destroy();
            return new Integer(line);
        }
        catch(Exception e){
            System.err.println(e);
        }
        return new Integer(-1);
    }

    static String getSubjectLevel(){
        try{
            Process p = rt.exec("listsl");

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));

            String line = in.readLine();
            p.destroy();
            return line;
        }
        catch(Exception e){
            System.err.println(e);
        }
        return null;
    }

    static String getUserName(){
        try{
            Process p = rt.exec("whoami");

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));

            String line = in.readLine();
            p.destroy();
            return line;
        }
        catch(Exception e){
            System.err.println(e);
        }
        return null;
    }
    static boolean isRootProcess(){
        try{
            Process p = rt.exec("id -u");

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));

            String line = in.readLine();
            p.destroy();
            return line.equals("0");
        }
        catch(Exception e){
            System.err.println(e);
        }

        return false;
    }


    static void getUserLevels(Vector data){
        try{
            Process p = rt.exec("listul ");

            BufferedReader in =
                new BufferedReader(new InputStreamReader(p.getInputStream()));
            StringTokenizer st;
            for(String line = in.readLine(); line != null; line = in.readLine
                ()){
                st = new StringTokenizer(line);
                Object[] s = {st.nextToken(), new Integer(st.nextToken())};
                data.add(s);
            }
            p.destroy();
        }
        catch(Exception e){
            System.err.println("Error:"+e);
        }
    }
}
```

F.8.2 SecurityManagerGUI.java

```
import javax.swing.*;
import java.awt.*;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import java.util.*;
import java.io.*;
import java.awt.event.*;

public class SecurityManagerGUI extends JFrame implements
    ListSelectionListener{
    final JPanel menu = new JPanel();
    /**
     * The index of the selected menu item.
     */
    private int selectedIndex;

    /**
     * The list that contains the available menu items.
     */
    final JList list;
    final CardLayout cardLayout = new CardLayout();
    final JPanel panel = new JPanel(cardLayout);

    final private InitPanel initPanel = new InitPanel();
    private FileLevelPanel fileLevelPanel;
    private UserLevelPanel userLevelPanel = new UserLevelPanel(this);
    final MessagePanel messagePanel = new MessagePanel(this);
    final SubjectLevelPanel subjectLevelPanel = new SubjectLevelPanel(this);
    final private WindowPanel unmappedWindowsPanel = new WindowPanel(this,
        false);
    final private WindowPanel mappedWindowsPanel = new WindowPanel(this, true)
        ;
    final static int width = 700;
    final static int height = 400;

    public SecurityManagerGUI(File mountPoint){
        super("Security Manager GUI");
        setSize(width, height);
        setBounds(100,100, width+100, height+100);

        Container contents = getContentPane();
        contents.setLayout(new BorderLayout());
        contents.add(panel, BorderLayout.CENTER);
        panel.add(initPanel, InitPanel.id);
        fileLevelPanel = new FileLevelPanel(this, mountPoint);
        panel.add(fileLevelPanel, FileLevelPanel.id);
        panel.add(userLevelPanel, UserLevelPanel.id);
        panel.add(unmappedWindowsPanel, "UnmappedWindowsPanel");
        panel.add(mappedWindowsPanel, "MappedWindowsPanel");
        panel.add(messagePanel, MessagePanel.id);
        panel.add(subjectLevelPanel, SubjectLevelPanel.id);

        addWindowListener(new WindowAdapter(){
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });

        setFocusCycleRoot(true);
        setFocusTraversalPolicy(new LayoutFocusTraversalPolicy());

        // Create the menu
        String[] menuItems = new String[]{"File Level Management",
                                          "User Level Management",
                                          "Unmapped Windows",
                                          "Mapped Windows",
                                        "Current Subject Levels"};
        list = new JList(menuItems);
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        //list.setBackground(new Color(204, 204, 255));
        list.setBackground(new Color(204, 153, 255));
        list.setFont(new Font("Helvetica", Font.BOLD, 14));
        list.addListSelectionListener(this);

        menu.add(list);
        menu.setBackground(list.getBackground());
        contents.add(menu, "West");      // Add the menu

        cardLayout.show(panel, InitPanel.id);
  }
    /**
     * This method is called whenever a new item is selected  in the menu.
     */
    public void valueChanged(ListSelectionEvent e){
        if (e.getValueIsAdjusting())
            return;
        updatePanel();
    }

    void updatePanel(){
        selectedIndex = list.getSelectedIndex();
        switch(selectedIndex){
        case 0: // File Level Management
            fileLevelPanel.refresh.doClick();
            break;
        case 1: // User Level Management
            userLevelPanel.refresh.doClick();
            break;
        case 2: // Unmapped Windows
            unmappedWindowsPanel.refresh.doClick();
            break;
        case 3: // Mapped Windows
            mappedWindowsPanel.refresh.doClick();
            break;
        case 4: // Current Subject Levels
            subjectLevelPanel.refresh.doClick();
            break;
        default:
            cardLayout.show(panel, InitPanel.id);

        }
    }


    public static void main(String[] args){

        if(args.length < 1){
            System.err.println("Usage: java SecurityManagerGUI MOUNT_POINT\n")
                ;
            System.exit(1);
        }
        File mountPoint = new File(args[0]);
        if(!mountPoint.isDirectory()){
            System.err.println("\nInvalid mount point: \""+mountPoint+"\" does
                not exist or is not a directory.\n");
            System.exit(1);
        }

        SecurityManagerGUI smGUI = new SecurityManagerGUI(mountPoint);
        smGUI.setVisible(true);
    }
}
```

F.8.3 BasicPanel.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * The BasicPanel is the abstract super class of all the panel classes
 * that are used in the SEAC System.
 */
abstract class BasicPanel extends JPanel implements ActionListener{

    final JPanel instructionPanel = new JPanel(),
        inputPanel = new JPanel(),
        buttonPanel = new JPanel();

    final JLabel instr = new JLabel();

    final JButton refresh = new JButton("   Refresh   ");
    final JButton cancel = new JButton("   Cancel   ");

    final static Font instrFont = new Font("TimesRoman", Font.BOLD, 20);
    final static Color panelColor = new Color(255, 255, 153);


    BasicPanel(){
        super(new BorderLayout());

        // setup of the instructionPanel
        instr.setFont(instrFont);
        instructionPanel.setBackground(panelColor);
        instructionPanel.add(instr);
        add(instructionPanel, "North");

        // setup of the inputPanel
        inputPanel.setBackground(panelColor);
        add(inputPanel, "Center");

        // setup of the buttonPanel
        buttonPanel.setBackground(panelColor);
        buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT, 15, 10));
        buttonPanel.add(refresh);
        add(buttonPanel, "South");

        // Setup of button
        refresh.addActionListener(this);
        refresh.setMnemonic('r');
    }

    void setInstruction(String instruction){
        instr.setText(instruction);
    }
}
```

F.8.4 InitPanel.java

```
import java.awt.event.ActionEvent;
import java.awt.Font;

class InitPanel extends BasicPanel{

    final static String id = "Init";

    InitPanel(){
        buttonPanel.remove(refresh);
        buttonPanel.remove(cancel);

        setInstruction("Sensor Enhanced Access Control System");
    }

    public void actionPerformed(ActionEvent e){}
}
```

F.8.5 MessagePanel.java

```
import java.awt.event.ActionEvent;
import java.awt.GridBagLayout;
import javax.swing.JLabel;

class MessagePanel extends BasicPanel{
    final static String id = "Message";
    private JLabel label = new JLabel();

    final private SecurityManagerGUI frame;

    MessagePanel(SecurityManagerGUI frame){
        this.frame = frame;
        inputPanel.setLayout(new GridBagLayout());
        inputPanel.add(label);
   }

    public void setText(String text){
        label.setText(text);
    }

  public void actionPerformed(ActionEvent e){
      frame.updatePanel();
  }
}
```

F.8.6 FileLevelPanel.java

```
import java.io.File;
import java.awt.event.ActionEvent;
import java.awt.*;
import javax.swing.*;
import java.util.Vector;

public class FileLevelPanel  extends BasicPanel{
    final static String id = "File Level";

    final private SecurityManagerGUI frame;

    final private FileLevelTableModel tableModel = new FileLevelTableModel();
    final private TableSorter sorter = new TableSorter(tableModel);
    final  JTable table = new JTable(sorter);

    static File file;
    final private JFileChooser fc;

    final private JTextField fileTextField  =new JTextField(25);
    final JButton browse = new JButton("   Browse   ");

    final private int tableWidth = 340;
    final private int tableHeight = 100;

    FileLevelPanel(SecurityManagerGUI frame, File mountPoint){
        this.file = mountPoint;
        this.frame = frame;
        setInstruction("File Level Management");

        fc = new JFileChooser(file);
        fc.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);

        // Table Setup
        sorter.setTableHeader(table.getTableHeader());
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        table.getTableHeader().setReorderingAllowed(false);
        table.setCellSelectionEnabled(true);

        // Setup of the inputPanel
        GridBagLayout layout = new GridBagLayout();

        // Textfield
        GridBagConstraints c = new GridBagConstraints();
        c.gridx = 0;
        c.gridy = 0;
        instructionPanel.remove(instr);
        instructionPanel.setLayout(layout);
        layout.setConstraints(instr, c);
        instructionPanel.add(instr);

        c.gridy++;
        c.gridwidth = GridBagConstraints.RELATIVE;
        c.insets.right = 20;
        c.insets.bottom = 20;
        c.insets.top = 20;

        fileTextField.setText(file.toString());
        layout.setConstraints(fileTextField, c);
        instructionPanel.add(fileTextField);

        // ''Browse'' Button
        c.gridx++;
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.insets.right = 0;

        browse.setMnemonic('b');
        layout.setConstraints(browse, c);
        instructionPanel.add(browse);

        // (File Name, Security Level) Table
        inputPanel.setLayout(layout);
        c.anchor = GridBagConstraints.CENTER;
        c.gridx = 0;
        c.gridy++;

        JScrollPane scrollPane = new JScrollPane(table);
        layout.setConstraints(scrollPane, c);
        inputPanel.add(scrollPane);

        browse.addActionListener(this);
        fileTextField.addActionListener(this);
    }


    boolean updateTable(){

        tableModel.data.removeAllElements();
        if(!file.exists()){
            JOptionPane.showMessageDialog(null,
                                          "No such file or directory: "+file,
                                          "File Selection Error",
                                          JOptionPane.ERROR_MESSAGE);
            return false;
        }

        if(file.isFile()){
            Integer res = Exec.getFileLevel(file.toString());

            if(res.equals(new Integer(-1))){
                JOptionPane.showMessageDialog(null,
                                                "Error: file selection is not
                                                   supported in this directory.
                                                   ",
                                                "File Selection Error",
                                                JOptionPane.ERROR_MESSAGE);
                return false;
            }
            tableModel.data.add(new Object[]{ file.getName() , res});
        }
        else{ // file.isDirectiory() == true
            String res = Exec.getFileLevels(tableModel.data, file.toString());
            if(res != null){
                JOptionPane.showMessageDialog(null,
                                                res,
                                                "File Selection Error",
                                                JOptionPane.ERROR_MESSAGE);
                return false;
            }
        }

        fileTextField.setText(file.toString());
        /*
        if(table.getRowCount() == 0){
            frame.messagePanel.setText("No files exists.");
            frame.cardLayout.show(frame.panel, MessagePanel.id);
            return true;
            }*/

        // Setup of the table size

        int height = table.getRowHeight()*table.getRowCount();
        int width = table.getColumnModel().getColumn(0).getWidth() +
            table.getColumnModel().getColumn(1).getWidth();

        Dimension tableDimension = new Dimension(tableWidth, height);

        if(height > SecurityManagerGUI.height/2)
            tableDimension.height = SecurityManagerGUI.height/2;

        table.setPreferredScrollableViewportSize(tableDimension);
        tableModel.fireTableStructureChanged();
        tableModel.fireTableDataChanged();

        frame.panel.revalidate();
        frame.panel.repaint();
        frame.cardLayout.show(frame.panel, FileLevelPanel.id);
        return true;
    }

    public void actionPerformed(ActionEvent e){
        File curFile = file;
        Object source = e.getSource();
        if(source == refresh){
            if(!updateTable()){
                file = curFile;
                refresh.doClick();
            }
        }
        else if(source == browse){
            int returnVal = fc.showDialog(this, "Select");
            if(returnVal == JFileChooser.APPROVE_OPTION) {
                file = fc.getSelectedFile().getAbsoluteFile();
                if(!updateTable()){
                    file = curFile;
                    refresh.doClick();
                }
            }
        }
        else if(source == fileTextField){
            file = new File(fileTextField.getText());
            if(!updateTable()){
                file = curFile;
                refresh.doClick();
            }
        }
    }
}
```

F.8.7 FileLevelTableModel.java

```
import javax.swing.JOptionPane;
import java.util.Vector;

public class FileLevelTableModel extends SimpleTableModel {

    FileLevelTableModel(){
        super(new String[]{"File Name", "File Level"});
    }

    public boolean isCellEditable(int row, int col) {
        return Exec.isRootProcess() && col == 1;
    }

    public void setValueAt(Object value, int row, int col) {
        Object[] s  = (Object[]) data.elementAt(row);
        String errMessage = Exec.setFileLevel(FileLevelPanel.file.
            getAbsolutePath()+"/"+s[0], (Integer) value);
        if(errMessage != null)
            JOptionPane.showMessageDialog(null, errMessage,
                                          "Set File Level Error",
                                          JOptionPane.ERROR_MESSAGE);
        else{
            s[col] = value;
            data.setElementAt(s, row);
        }
        fireTableCellUpdated(row, col);
    }
}
```

F.8.8 SimpleTableModel.java

```
import javax.swing.table.AbstractTableModel;
import java.util.Vector;

class SimpleTableModel extends AbstractTableModel {
    Vector data = new Vector();
    String[] columnNames;

    SimpleTableModel(String[] columnNames){
        this.columnNames = columnNames;
    }

    public String getColumnName(int col) {
        return columnNames[col];
    }

    public int getColumnCount() {
        return columnNames.length;
    }

    public int getRowCount() {
        return (data == null) ? 0 : data.size();
    }

    public Object getValueAt(int row, int col) {
        return ((Object[]) data.elementAt(row))[col];
    }

    public Class getColumnClass(int c) {
        return getValueAt(0, c).getClass();
    }
}
```

F.8.9 UserLevelPanel.java

```
import java.awt.event.ActionEvent;
import java.awt.*;
import javax.swing.*;
import java.util.Vector;

public class UserLevelPanel  extends BasicPanel{
    final static String id = "User Level";

    final private SecurityManagerGUI frame;

    final private UserLevelTableModel tableModel = new UserLevelTableModel();
    final private TableSorter sorter = new TableSorter(tableModel);
    final  JTable table = new JTable(sorter);

    final private int tableWidth = 340;
final private int tableHeight = 100;

UserLevelPanel(SecurityManagerGUI frame){
    this.frame = frame;
    setInstruction("User Level Management");

    // Table Setup
    sorter.setTableHeader(table.getTableHeader());
    table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    table.getTableHeader().setReorderingAllowed(false);
    table.setCellSelectionEnabled(true);
    table.getColumnModel().getColumn(0).setPreferredWidth(tableWidth/2);
    table.getColumnModel().getColumn(1).setPreferredWidth(tableWidth/2);

    // Setup of the inputPanel
    GridBagLayout layout = new GridBagLayout();

    // Textfield
    GridBagConstraints c = new GridBagConstraints();
    c.gridx = 0;
    c.gridy = 0;
    instructionPanel.remove(instr);
    instructionPanel.setLayout(layout);
    layout.setConstraints(instr, c);
    instructionPanel.add(instr);

    c.insets.bottom = 20;
    c.insets.top = 20;
    c.gridwidth = GridBagConstraints.REMAINDER;
    c.insets.right = 0;

    // (User ID, User Level) Table
    inputPanel.setLayout(layout);
    c.anchor = GridBagConstraints.CENTER;
    JScrollPane scrollPane = new JScrollPane(table);

    layout.setConstraints(scrollPane, c);
    inputPanel.add(scrollPane);
}

void updateTable(){
    if(!Exec.isRootProcess()){
        frame.messagePanel.setText("The user "+Exec.getUserName()+" has
            user level "+Exec.getUserLevel()+".");
        frame.cardLayout.show(frame.panel, MessagePanel.id);
        return;
    }


    tableModel.data.removeAllElements();
    Exec.getUserLevels(tableModel.data);

    // Setup of the table size

    int height = table.getRowHeight()*table.getRowCount();
    Dimension tableDimension  = (height < SecurityManagerGUI.height/2) ?
```

```
            new Dimension(tableWidth, height) :
            new Dimension(tableWidth, SecurityManagerGUI.height/2);

        table.setPreferredScrollableViewportSize(tableDimension);
        tableModel.fireTableStructureChanged();
        frame.cardLayout.show(frame.panel, UserLevelPanel.id);
    }

    public void actionPerformed(ActionEvent e){
        Object source = e.getSource();
        if(source == refresh)
            updateTable();
    }
}
```

F.8.10 UserLevelTableModel.java

```
import javax.swing.JOptionPane;
import java.util.Vector;

public class UserLevelTableModel extends SimpleTableModel {

    UserLevelTableModel(){
        super(new String[]{"User Name", "User Level"});
    }

    public boolean isCellEditable(int row, int col) {
        return col == 1;
    }

    public void setValueAt(Object value, int row, int col) {
        if(value == null)
            return;

        Object[] s  = (Object[]) data.elementAt(row);
        String errMessage = Exec.setUserLevel((String) getValueAt(row, 0), (
            Integer) value);
        if(errMessage != null)
            JOptionPane.showMessageDialog(null, errMessage,
                                          "Set user level error",
                                          JOptionPane.ERROR_MESSAGE);
        else{
            s[col] = value;
            data.setElementAt(s, row);
        }
        fireTableCellUpdated(row, col);
    }
}
```

F.8.11 WindowPanel.java

```
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import java.io.File;
import java.awt.event.ActionEvent;
import java.awt.*;
import javax.swing.*;
import java.util.Hashtable;
import java.util.Vector;

public class WindowPanel extends BasicPanel implements
                                                ListSelectionListener{
    String id;

    final private SecurityManagerGUI frame;

    final WindowTableModel tableModel = new WindowTableModel();

    final private TableSorter sorter = new TableSorter(tableModel);
    final JTable table = new JTable(sorter);
    final private int tableWidth = 520;

    final private SimpleTableModel dialogModel = new SimpleTableModel(new
        String[]{"File Name", "File Level"});
    // final private TableSorter dialogSorter = new TableSorter(dialogModel);
    //final private JTable dialogTable = new JTable(dialogSorter);
    final private JTable dialogTable = new JTable(dialogModel);
    final private JDialog dialog = new JDialog();
    final private Hashtable details = new Hashtable();

    private boolean mappedWindows = false;

    WindowPanel(SecurityManagerGUI frame, boolean mappedWindows){
        id = (mappedWindows) ? "Mapped Windows" : "Unmapped Windows";
        setInstruction(id);
        this.mappedWindows = mappedWindows;
        this.frame = frame;

        // Setup dialog
        dialogTable.getTableHeader().setReorderingAllowed(false);
        dialog.setSize(new Dimension(430, 250));
        dialog.setDefaultCloseOperation(JDialog.HIDE_ON_CLOSE);
        dialog.getContentPane().add(new JScrollPane(dialogTable));

        // Table Setup
        sorter.setTableHeader(table.getTableHeader());
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        table.setRowSelectionAllowed(true);
        table.getTableHeader().setReorderingAllowed(false);

        table.getSelectionModel().addListSelectionListener(this);

        instructionPanel.add(instr);

        GridBagLayout layout = new GridBagLayout();
        inputPanel.setLayout(layout);

        GridBagConstraints c = new GridBagConstraints();
        c.anchor = GridBagConstraints.WEST;
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.insets.bottom = 15;
        JLabel lb = new JLabel("Click on a row to see the open files and
            associated file levels:");
        layout.setConstraints(lb, c);
        inputPanel.add(lb);

        JScrollPane scrollPane = new JScrollPane(table);
        layout.setConstraints(scrollPane, c);
        inputPanel.add(scrollPane);
    }

    void updateTable(){
        tableModel.data.removeAllElements();
        Exec.getWindowInfo(tableModel.data, details, mappedWindows);

        if(table.getRowCount() == 0){
            if(mappedWindows){
                frame.messagePanel.setText("No windows are mapped.");
                frame.cardLayout.show(frame.panel, MessagePanel.id);
            }
            else{
                frame.messagePanel.setText("No windows are unmapped.");
                frame.cardLayout.show(frame.panel, MessagePanel.id);
            }
            return;
        }

        int height = table.getRowHeight()*table.getRowCount();
        Dimension tableDimension  = (height < SecurityManagerGUI.height/2) ?
            new Dimension(tableWidth, height) :
            new Dimension(tableWidth, SecurityManagerGUI.height/2);
        table.setPreferredScrollableViewportSize(tableDimension);
        tableModel.fireTableStructureChanged();
        if(mappedWindows)
            frame.cardLayout.show(frame.panel, "MappedWindowsPanel");
        else
            frame.cardLayout.show(frame.panel, "UnmappedWindowsPanel");

        revalidate();
        repaint();
    }

    public void actionPerformed(ActionEvent e){
        updateTable();
    }

    public void valueChanged(ListSelectionEvent e) {
        //Ignore extra messages.
        if (e.getValueIsAdjusting()) return;

        ListSelectionModel lsm = (ListSelectionModel) e.getSource();
        if (!lsm.isSelectionEmpty()){
            Exec.getWindowInfo(tableModel.data, details, mappedWindows);
            int row = lsm.getMinSelectionIndex();
            Object winID = tableModel.getValueAt(row, 2);

            dialogModel.data = (Vector) details.get(winID);
            dialogTable.clearSelection();
            dialogTable.revalidate();
            dialogTable.repaint();

            dialog.setTitle("The files that have been open in window "+winID);
            dialog.setLocationRelativeTo(frame);
            dialog.setVisible(true);
        }
    }
}
```

F.8.12 WindowTableModel.java

```
import javax.swing.*;
import javax.swing.table.AbstractTableModel;
import java.util.Vector;

class WindowTableModel extends SimpleTableModel {
    WindowTableModel(){
        super(new String[] {"Application Name", "PID", "Window ID", "Window
           Level", "No. of Open Files"});
    }
}
```

F.8.13 SubjectLevelPanel.java

```
import java.awt.event.ActionEvent;
import java.awt.*;
import javax.swing.JLabel;

class SubjectLevelPanel extends BasicPanel{
    final static String id = "SubjectLevel";
    private JLabel label = new JLabel();
    private JLabel label2 = new JLabel();

    final private SecurityManagerGUI frame;

    SubjectLevelPanel(SecurityManagerGUI frame){
        this.frame = frame;

        // Setup of the inputPanel
        GridBagLayout layout = new GridBagLayout();
        inputPanel.setLayout(layout);

        // Textfield
        GridBagConstraints c = new GridBagConstraints();
        c.anchor = GridBagConstraints.WEST;
        c.gridx = 0;
        c.gridy = 0;
        c.insets.bottom = 10;

        layout.setConstraints(label, c);
        inputPanel.add(label);

        c.gridy++;
        layout.setConstraints(label2, c);
        inputPanel.add(label2);
   }

    public void setText(String text){
        label.setText(text);
        label2.setText("");
    }

    public void setText2(String text){
        label2.setText(text);
    }

    public void actionPerformed(ActionEvent e){
        int clearanceLevel = Exec.getClearanceLevel();
        if(clearanceLevel == -1)
            setText("The clearance level has not been initialised.");
        else{
            setText("Current subject levels: "+Exec.getSubjectLevel());
            setText2("The clearance level: "+clearanceLevel);
        }
        frame.cardLayout.show(frame.panel,  SubjectLevelPanel.id);
    }
}
```

F.9 System Administration Scripts

F.9.1 reset.sh

```
#!/bin/bash -norc
set -x
MOUNT_POINT=/mnt/macfs
FILE_LEVELS="/root/.file_levels"
USER_LEVELS="/root/.user_levels"

# Remove old files
rm -rf ${FILE_LEVELS} ${USER_LEVELS} /mnt/macfs/*

if [ ! -f "${MOUNT_POINT}" ]
then mkdir "${MOUNT_POINT}"
     chmod a+twrx /mnt/macfs
fi

if [ ! -f "${FILE_LEVELS}" ]
then touch ${FILE_LEVELS}
     chmod u=rw,go= ${FILE_LEVELS}
```

```
fi

if [ ! -f "${USER_LEVELS}" ]
then touch ${USER_LEVELS}
     chmod u=rw,go= ${USER_LEVELS}
fi
```

F.9.2 startup.sh

```
#!/bin/bash -norc
set -x

POLICY="/home/s973732/seac/test/policy.txt"
FILE_LEVELS="/root/.file_levels"
USER_LEVELS="/root/.user_levels"

MOUNT_POINT=/mnt/macfs
LOWERDIR=${MOUNT_POINT}

# A regular user will be permitted access to retrieving his own user level
chmod +s getul
chown 0:0 getul

if ! access x "${USER_LEVELS}"
then chmod +s initcl
     chown 0:0 initcl
fi

# file system init
insmod /home/s973732/seac/out/Linux-2.4/macfs/macfs.o || exit
mount -t macfs -o dir=${LOWERDIR} ${LOWERDIR} ${MOUNT_POINT} || exit
seac_init ${POLICY} ${FILE_LEVELS} ${USER_LEVELS} || exit

# User space level init
visibility_manager ${POLICY} &
file_open_monitor &
sensor_server &
```

F.9.3 shutdown.sh

```
#!/bin/bash -norc
set -x
MOUNT_POINT=/mnt/macfs

seac_destroy
swsensor -1 i
umount ${MOUNT_POINT}
rmmod macfs
```