Diplomarbeit

# Development of a Prototype for a Security Platform for Mobile Devices

| | |
|---|---|
| *eingereicht von:* | Christian Stüble |
| *geboren am:* | 31. Oktober 1973 |
| *geboren in:* | Hildesheim |
| *Matrikel-Nr.:* | 66740 |
| *eingereicht am:* | 28. April 2000 |
| *Betreuerin:* | Prof. Dr. Birgit Pfitzmann |
| *Gutachter:* | Prof. Dr. Joachim Biskup |

Universität Dortmund,
Lehrstuhl für Informationssysteme und Sicherheit

Universität des Saarlandes,
Lehrstuhl für Kryptographie und Sicherheit

## Danksagungen

Ich möchte mich bei allen bedanken, die mich bei meiner Arbeit unterstützt haben! Zuallererst natürlich bei meinen Eltern Monika und Wolfgang Stüble, die mich während meiner gesamten Ausbildung begleitet and motiviert haben. Ohne ihre Unterstützung wäre weder mein Studium, noch diese Diplomarbeit möglich gewesen.

Herzlich bedanken möchte ich mich auch bei allen Mitgliedern der ISSI Gruppe der Universität Dortmund und des Lehrstuhls für Kryptographie und Sicherheit der Universität Saarbrücken, insbesondere bei meiner Betreuerin Prof. Dr. Birgit Pfitzmann, bei Prof. Dr. Joachim Biskup, Matthias Schunter und Tom Beiler.

Sehr hilfreich war auch die Unterstützung von Frank Mehnert vom Lehrstuhl Betriebssysteme der Technischen Universität Dresden bei Fragen rund um den $\mu$-kernel.

Letztendlich möchte ich mich für die gute Zusammenarbeit während unserens gesamten Studiums bei meinem Freund und Kommilitonen Ralph Kühl bedanken.

# Abstract

*Security-critical applications such as homebanking, digitally signing documents (using e.g. PGP) or various kinds of business over the Internet rely on the security of all components involved. The main problematic components in the current practice are insecure operating systems, which provide only weak security policies and are often very error prone.*

*Hence a secure kernel, small enough to be evaluated based on the Common Criteria or ITSEC and running on a mobile personal device, is required. It must be able to protect applications from each other and provide a trusted path between these applications and the user. However, user devices that do not offer the full functionality of a widespread operating system seem unmarketable.*

*Therefore we propose to run a Client OS as one encapsulated application on the secure kernel. Moreover, by using the Client OS judiciously to perform non-critical tasks, the size of the secure kernel can be significantly reduced compared to a stand-alone secure system.*

*In this diploma thesis, a first design and prototype of such a secure kernel, called PERSEUS, is presented. It is based on the* FIASCO *microkernel and runs* LINUX *as Client OS. The implementation contains all components to securely sign documents created under* LINUX.

*»If you wait for a complete and perfect concept to germinate in your mind, you are likely to wait forever.«*

DeMarco

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Electronic commerce and in general internet-based information technology grows fast and security and privacy is essential for those applications. At this point of time a handful usable solutions and certified implementations of important components like digital signatures and electronic payment systems are already available. Nevertheless all these systems cannot become more secure than the underlying operating system.

Malicious applications (trojan horses and viruses) are able to break into nearly all publicly available operating systems, to modify or spy out especially security-related software and data, even if the user or administrator has made no errors. Even smartcards cannot solve this problem, because untrusted operating systems act as a software layer between user and smartcard interface and process user authentication software. Therefore a secure environment is required which is able to provide a trusted path between users and secure components.

This is the main goal of the PERSEUS[1] (Pervasive security for us) project. Its basic idea is to use existing devices and operating systems and extend them by means of a trusted secure environment which separates user processes, controls the communication between them and enforces security policies. Both, porting an existing operating system to the security kernel and developing secure applications which are separated from the operating system is required. To simplify development of secure applications the secure environment shall provide a framework containing simple implementations of basic functionalities. Simplicity is necessary to make evaluation of security-related components possible. Example evaluation criteria are the *Common Criteria* [37], which partially need formal techniques to evaluate higher security levels.

The proposal of the PERSEUS project [21] suggests a detailed software architecture of the secure environment, illustrated by Figure 1.1. It informally describes approximately twelve components, their contents and relations. Modules between the red and yellow line which provide basic functionality are called the *secure platform (SP)*. It uses functions of the $\mu$-kernel, which abstracts hardware-dependent properties (below red line). Above the yellow line an existing operating system, (hence) called *Client OS*, runs as one task of the $\mu$-kernel. Also an *application framework* which provides support to *secure applications* (above green line) is planned. All security-related layers together, hardware, $\mu$-kernel, secure platform, application framework and secure applications, are called *secure environment (SE)*.

[1] http://www.semper.de/sure

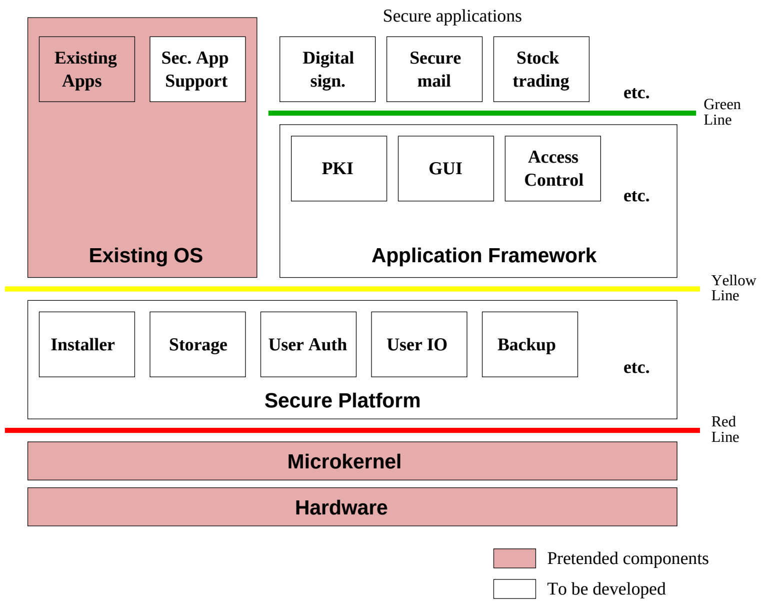

Figure 1.1: Software architecture suggested by the preliminary PERSEUS architecture [21].

## 1.1 Objectives

A personal digital assistant (PDA) or other IT products represent subjects (users) in a digital IT environment. The products are supposed to act in the interest of the user who is the only one who can lay down whether, and how, the IT product represents the user to the rest of the world. This property is generally called the *right of informational self-determination.* In order to be able to guarantee this right some requirements have to be fulfilled by the IT product:

1. It has to protect components (applications, services, etc.) from each other.
2. Absence of undesirable functions.

The first requirement is summarized by the term *security* of the IT product. This means that it has to provide confidentiality, integrity, availability and reliability and should neither by external attacks, nor by internal errors be made to act in other ways than defined by its specifications.

The second requirement significantly depends on the trust in the agreement of the specification and the implementation. The past has shown that the agreement rarely applies. Nearly all applications and operating systems have bugs, some programs send user-information to their developers [29], others have undocumented certification keys which make it possible to bypass security-barriers and replace security-related software [28]. To create and increase trust in the IT product the following five conditions have to be fulfilled:

1. Functional and security-related system requirements have to be defined.
2. The system functions have to be specified based on these requirements.
3. It has to be checked that specified system functions can fulfill demanded system requirements.

4. It has to be checked that specified and implemented system functions are equal.

5. The user (or a trusted third party) has to verify and evaluate development steps of condition 2, 3 and 4.

In general a user or group of users defines the requirements on the IT product, then developers have to fulfill condition 2-4. To make evaluation and comparison of systems possible for users or trusted third parties, developers could, e.g., use the *Common Criteria for Information Technology Security Evaluation* (see Section 3.1). Further steps which increase the trust in IT products are to make the development steps and/or the source code publicly available (open source) or to modularize the system to make it possible for users to select/exchange specific modules.

In the past, a lot of successful attempts have been made to develop provable secure operating systems from scratch. The problem of these systems is that they are binary- and interface-incompatible to existing operating systems. Thus users have to get accustomed to new environments and nearly all applications need to be rewritten. This may be a reason why these secure operating systems live in the shadow.

To avoid this problem the main goal of the PERSEUS project, and thereby of this diploma thesis, is to develop a secure operating system kernel which executes as one task an existing operating system (*Client OS*) which itself provides users and developers a well-known environment (Figure 1.2).

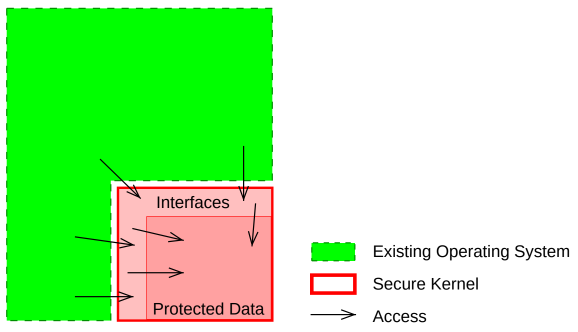

Figure 1.2: A secure operating system kernel protects security-related information.

Also the secure kernel takes over all security-related functions and protects security-related system and user data which can only be accessed by means of a defined set of interfaces. Besides the secure kernel is able to enforce its own security policy, independent of the existing operating system.

## 1.2 Related Work

Similar architectures which provide functionality for electronic commerce applications are SEMPER [24] and the Java Commerce Client [45]. However, they are based on a complete operating system and do not consider restrictions of evaluation criteria of high security levels. Thus the extent of the PERSEUS functionality has to be much smaller.

The approach to use an existing operating system based on a microkernel and provide additional service modules related to the microkernel, as suggested by preliminary PERSEUS architecture, has been successfully performed by two other projects:

The RT-LINUX Project[2] of the University of New Mexico uses a small kernel to provide real-time processes in parallel to the LINUX operating system [6]. Also the DROPS project[3], which uses the L4/FIASCO $\mu$-kernel and an adapted L4-LINUX [27] [20] provides real-time processes.

Based on a predecessor of the L4 $\mu$-kernel, L3, a persistent environment is realized providing separated mutual protected DOS clients [26]. This approach is taken up in Section 4.5.2.1.

## 1.3 Limits

To develop the entire PERSEUS project a lot of man-years are estimated. Thus this diploma thesis is only a very small part of the whole project. The goal of this diploma thesis is to develop the first prototype to help evaluating design-decisions early by means of prototypes. Because of the limited time most of the components will contain only dummy implementations. Howeverthe focus will remain on definitions of flexible interfaces and a good documentation of design decisions to be able to refine the model without global modifications and to use it as a starting point for discussions, extensions and improvements.

Although the PERSEUS project aims to develop a secure environment for mobile devices, implementations of this diploma thesis shall be based on a standard PC. It is planned to use the FIASCO[4] microkernel as the lowest software layer, which is available for free at the University of Dresden, a PERSEUS member which provides support. A FIASCO port of the LINUX[5] operating system is available there[6] also.

Since software evaluation needs not to be considered by this prototype, no concrete design restrictions need to be obeyed.

## 1.4 Fundamentals

This section contains short explanations of fundamental concepts and definitions of terms used in this paper.

### 1.4.1 Fonts

This diploma thesis uses various scriptings to make reading of this document easier.

*Glossary* Terms which have been added to the glossary (Appendix D) are marked using italics.

Classes and Packages Classes and packages (groups of classes) are written using sans-serif scripting.

`Attributes`, `variables`, `functions`, `methods` and `sourcecode` Attributes of classes, programming examples, variables and functions are written in courier. To distinguish attributes and functions, the latter are marked by two brackets at the end of the name.

ROLES Roles have been marked using only capital letters.

[2]http://rtlinux.cs.nmt.edu/~rtlinux
[3]http://os.inf.tu-dresden.de/drops
[4]http://os.inf.tu-dresden.de/fiasco
[5]http://www.linux.org
[6]http://os.inf.tu-dresden.de/L4Linux

### 1.4.2 Naming

Names of classes, packets, functions, methods and attributes have been selected according to UML suggestions, if possible.

Every word of names of classes and packages starts with a capital letter without underscores between words. The first word of attribute and function names begins with a small letter, other words with a capital letter. Examples: MyPacket, MyClass, `myAttribute` and `myFunction()`.

### 1.4.3 Software-Development

Different approaches to develop software products exist and a lot of books have been written which all represent the "best" way to do so. HELMUT BALZERT presents in [5] a good and easy to understand overview of aspects common and different to some of these concepts. This diploma thesis uses the object-oriented (OO) approach, because it is currently a very common way of software-development; further concepts have been used whenever this made it easier to express certain facts. A good introduction to OO-software development is provided in [4].

The object-oriented approach divides development into three phases:

***Object-Oriented Analysis.*** The main goal of the analysis phase is to determine and describe requirements and wishes of customers of the IT product to be developed. It is important to know that design-decisions and implementation details have to be ignored completely and it is assumed that technologies work perfectly. At the end of the analysis phase an OOA model forms the conceptual solution of the product of development.

***Object-Oriented Design.*** The design phase has to realize the OOA model and consider requirements and restrictions of used hard- and software components. The main goal is to develop an OOD model while using efficiency and standards as points of view.

***Object-Oriented Implementation.*** Also known as *Object-Oriented Programming* (OOP). The goal of this phase is to implement the OOD model under constraints and restrictions of programming languages and underlying components (operating system, compiler, etc.). It is not absolutely necessary, though easier, to use object-oriented programming languages to implement an object-oriented design model. Thus object-oriented development steps can be re-used even if later versions use non-OO programming languages.

Decisions of the analysis and design phase have to be evaluated early by means of prototypes to be able to make improvements and to recognize faults early. This is the focusof this diploma thesis.

### 1.4.4 Concepts

Analysis and design models contain descriptions of static and dynamic relations between elements which can be represented by a wide range of concepts. This section mentions and explains, if necessary, those concepts which have been used in this diploma thesis.

In order to express static and dynamic relations between elements of all three development phases the *Unified Modeling Language* (UML), developed in 1997 by BOOCH, RUMBAUGH and JACOBSON [36], has been used. [35] contains an extensive description of the *Unified Modeling Language*. The overall software architecture is additionally represented similar to the figures of [21] to be able to illustrate differences and commons.

Dependencies between functions of design or implementation models are explained by function trees. Nodes of a function tree, drawn as rectangles and assigned with a name, represent functions. A function is a parent of other functions if its abstraction level is higher ("contains" or "calls"). Figure 1.3 shows an example of a function tree.

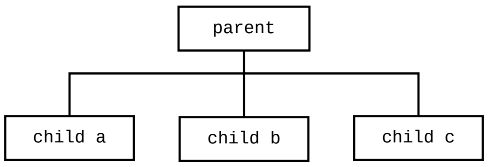

Figure 1.3: Function tree: Function `parent()` *calls* or *contains* three child functions `a()`, `b()` and `c()`.

Further concepts have been used to describe dynamic concepts. In order to illustrate Program Control Structures (PCS), language-independent symbols defined by DIN 66001 are used (see Figure 1.4). Data Flow Diagrams (DFD) according to DE-

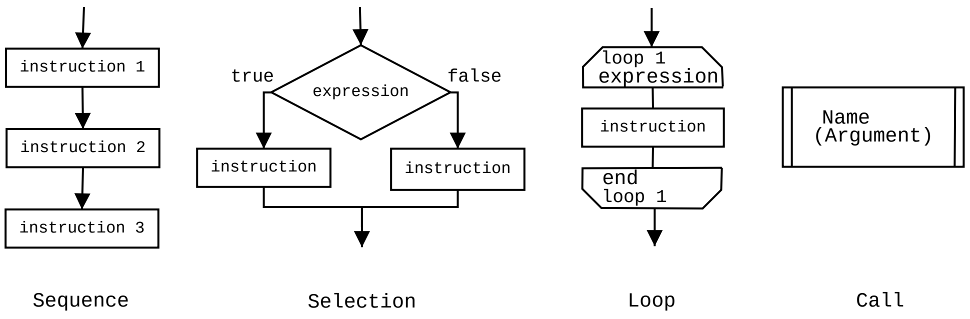

Figure 1.4: Expressions of program control structures.

MARCO [13] are used to describe flows and transformations of data and information between functions, storage and interfaces.

## 1.4.5 Patterns

In general, *patterns* describe classes of problems and explain at least one general solution of the problem. Patterns can be used in the analysis (*analysis pattern*), design (*design pattern*) and implementation phase (*implementation pattern*) of software development. Furthermore, patterns form abstractions which make communication between developers and understanding complex diagrams easier. According to *UML* a pointed oval, which contains the name of the pattern, points to elements which form a pattern and connects related components by means of lines. HEIDE BALZERT explains in [4] some analysis patterns and GAMMA, HELM, JOHNSON and VLISSIDES present in [16] a catalog of design patterns. A good starting point to look for C++ implementation patterns are [44] and [33].

## 1.4.6 Microkernel

The phrase *kernel* is used to denote the part of the operating system that is mandatory and common to all other software. It often runs in privileged execution levels and can therefore use all features of underlying hardware components (e.g. the processor).
Most early operating systems were implemented by means of monolithic kernels. Loosely speaking, the complete operating system was packed into one kernel. In

contrast to this, the basic idea of the *microkernel* (or $\mu$-kernel) is to minimize the kernel, which provides a general messaging system, and to implement whatever possible as servers outside of the kernel (Figure 1.5). All servers, even device drivers,

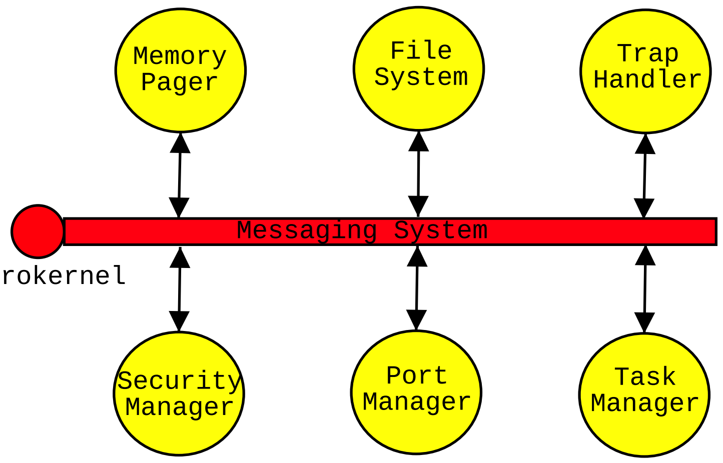

Figure 1.5: Minimized microkernel (red, protected mode) and system services (yellow, user mode).

run in user mode and are treated exactly like any other application. Since each server has its own address space, all these objects are protected against each other.

### 1.4.7 Subsystem

A *subsystem* is the smallest execution unit that can be protected by the hardware and it is similar to the *protected domain* defined by the *Common Criteria* in Section 3.1. In most cases a subsystem is a task that is protected from other tasks by virtual address spaces.
Two subsystems $S_1$ and $S_2$ are called **independent** if subsystem $S_1$ cannot be disturbed or corrupted by subsystem $S_2$.
A communication channel between two subsystems $S_1$ and $S_2$ provides **integrity** if it can neither be corrupted nor eavesdropped by another subsystem $S_3$.

### 1.4.8 Session

A *session* indicates the time interval between user log-in (opening a session) and log-out (closing a session). Accessing other subsystems than the login procedure is only possible if users have opened a session.

*Session Management* means that underlying components store the session's state (e.g. the state of applications, position and contents of windows) if the session is closed and restore them if the user opens a new session.

## 1.5 Overview

The structure of this document and the development-steps of the prototype are mostly similar to the processes described in [5]. Chapter 2 summarizes descriptions of functional requirements of the product under development, partially expressed by use cases in Section 2.2. This chapter also contains informal descriptions of security requirements from the user's point of view.

The next chapter defines the system behaviour by functional analysis of use cases and security-related examinations. Results of this chapter are expressed by an OOA model in Section 3.5.

Design decisions are discussed in Chapter 4 which leads to an OOD model, presented in Section 4.13.

Appendix A summarizes results of Chapter 2 and Appendix B contains a complete list of functional and security-related requirements of Chapter 3. Finally, Appendix C explains abbreviations used by this work and Appendix D contains a glossary.

I tried to keep the analysis and design sections independent of designs and decisions assumed by the PERSEUS proposal. At the end of each chapter the current system model is compared with the model of the proposal (Figure 1.1) by discussing important differences and commonalities.

Bear in mind that analysis, design and implementation of a software product are iterative tasks, which cannot be represented in this linear document completely. Therefore the next chapters contain only the latest version of appropriate development phases.

# Chapter 2

# Requirements Analysis

A *System Requirements Specification*, short SRS, defines an abstract description of the services and functions an IT product should provide and the constraints under which it must operate, and it shall be written in such a way that it is understandable by customers without special knowledge[1]. The main objective of this chapter is to create a system requirements specification, which is appended in Appendix A.

Section 2.1 contains informal (user-view) descriptions of functions of the product to be developed. Section 2.2 contains use cases which define the system behaviour by user view. Many of them should be readable without special knowledge, but some describe more specific cases which are important to motivate some security-related requirements. Hasty readers may skip this section. Section 2.3 contains informal descriptions of security demands, defines general requirements on access control and defines a simple role hierarchy to be enforced by the security policy. These definitions are required to use the *Common Criteria* (see Section 3.1)

## 2.1 Functional Requirements

This section contains informal descriptions of functional requirements of secure devices by user view:

- The IT product should contain a *Client OS* which provides a usual environment to users and developers.
- It should contain a *secure environment* which can be used to store security-related information and protects this information against external and internal attacks.
- To protect them against internal attacks the secure environment should be able to enforce its own security policy, independent of security policies of the *Client OS*.
- Users should be able to refine access permissions concerning to their objects without disturbing the security of the whole system.
- Users should be able to choose between security and usability, therefore the system should not force users to use local security policies.
- Users should be able to use the system without experience in security. Thus meaningful default values should be provided, but it should be possible for individuals to change them.

[1]We will see that this is not possible in all cases, because development of an operating system requires sometimes low-level knowledge.

- All functions which need access to security-critical information (e.g. secret keys) have to be provided by secure applications.
- Users should be able to verify that the product and/or applications are secure.

## 2.2 Use Cases

"A *use case* specifies the behaviour of a system or a part of the system and is a description of a set of actions, including variants, that a system performs to yield an observable result of value to an actor" [10]. They should not be too general, but neither contain design decisions nor implementation details.

The following subsections present use cases which have been organized into three parts: A general description of the goal of the use case and (optional) extensions and/or alternatives of the description and general security-related demands. The analysis of the use cases, based on security requirements of Section 3.1, follows in Section 3.2.

### 2.2.1 Use Case: **Signing an email**

***Description:*** The user has written an email using an ordinary mailtool and selects the "sign this message" button to inform the mailtool that s/he wants the email to be signed. To send the email s/he invokes the "send"-button. If more than one possible secret key exists, the system asks the user which key s/he wants to be used.

***Alternatives and extentions:*** The user or the mailtool can select the key to be used when the "sign this message" box is selected, or a default key may be specified. To calculate the signature, both a software implementation or a smartcard can be used.

***Demands:*** The user expects that nobody else is able to generate a valid signature concerning her/his key pair.

### 2.2.2 Use Case: **Generation of a new key pair**

***Description:*** The user wants to generate a new key pair. S/he invokes a key generation program to generate a new key pair. The program asks for key attributes (e.g. algorithm, key-id, key-length, etc.) and generates a new key pair.

***Alternatives and extentions:*** The program could hide the key attributes and provide a more abstract view (security-level etc.).

***Demands:*** The user wants to be able to distribute the public key, but requires that no one can observe the key generation process or copy the generated secret key.

### 2.2.3 Use Case: **Installing new services/applications**

***Description:*** The administrator of the system wants to install an application or service. S/he invokes a subsystem management program to install it by selecting a destination where the code of the new application can be found. The management program has to translate general access control rules into local security policies.

***Alternatives and extensions***: The (security-) administrator wants to be able to grant/restrict permission to install new applications/services.

***Demands:*** The user expects that the reliability and security of the existing system is not disturbed by the installation process and the new application.

### 2.2.4 Use Case: **Updating a system service**

***Description:*** The administrator wants to update an existing system service (e.g. the encryption service) by another (e.g. more trusted) one.
***Alternatives and extensions***:

- The new service implementation can have a higher version or patchlevel.
- The new service implementation can have a lower version or patchlevel.

***Demands:*** The user expects that the reliability and confidence of the existing system is not disturbed by the updating process.

### 2.2.5 Use Case: **Opening a session**

***Description:*** The user wants to open a new session. She/he has to identify herself/himself to make it possible for the system to grant correct permissions to the user and to reload the state of the last session.

***Demands:*** The user expects that he can continue at the point when he closed his session (*session management*). S/he also expects that another person cannot abuse the system if it is stolen or something like that.

### 2.2.6 Use Case: **Closing a session**

***Description:*** The user wants to close the current session.

***Alternatives and extensions:*** The user deactivates the system. After a primarily defined time interval of user inactivity the system closes the session itself.

***Demands:*** The user expects that the state of the current session is stored and that no one else is able to continue her/his session.

### 2.2.7 Use Case: **Activate system**

***Description:*** Somebody turns the system on to boot it.

***Alternatives and extensions:*** The user starts the system the first time.

***Demands:*** Users expect that the reached system state corresponds to the latest consistent state before the system has been deactivated. They also expect that another person cannot abuse the system, even if it is stolen or something like that.

### 2.2.8 Use Case: **Deactivate system**

***Description:*** The administrator wants to halt the system and invokes a system function to shut it down.

***Demands:*** The user expects that s/he can re-activate the system without loss of data.

### 2.2.9 Use Case: **Rollback**

***Description:*** If an unexpected error occurs which cannot be caught by the secure environment or if the secure environment itself has software bugs a defined set of users (role) should be able to rollback the system into its latest coherent state.

***Alternatives and extensions:*** If the system detects unrecoverable errors it can invoke the rollback operation itself.

***Demands:*** Users expect that the security of the system cannot be disturbed while the rollback operation is executed.

### 2.2.10 Use Case: **Enforcing local security policies**

***Description:*** The administrator installs, e.g. a file system service, which enforces its own security policy relating to files. Table 2.1 lists file-operations which could be provided by the service. The system-wide access control restricts permissions to

| | |
|---|---|
| 1. | create |
| 2. | delete |
| 3. | read |
| 4. | append |
| 5. | overwrite |
| 6. | move |

Table 2.1: Example file-operations provided by a file system service.

access the service to the role FILESYSTEMUSERS, but of course every user who is the owner of a file wants to be able to individually restrict access. Therefore the file system service has to enforce access control on its own.

***Alternatives***: A user who creates a new subsystem wants to restrict access to a subset of users of the system.

***Demands:*** The security-admin expects that global security policies cannot be bypassed. Users expect that their individual security policies are enforced.

### 2.2.11 Conclusion

Figure 2.1 contains an overview of discussed *use cases* and illustrates assignments to roles.

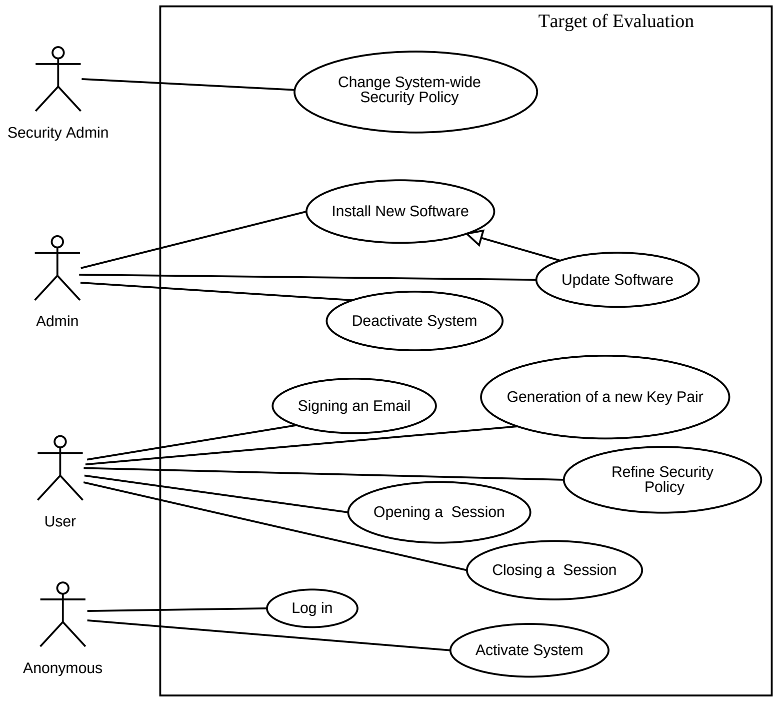

Figure 2.1: Use cases and related roles.

## 2.3 Security Requirements

This section contains an informal description of security requirements. This would be the right place to insert *Protection Profiles* (introduced in Section 3.1) which informally define security requirements by user view. As long as Protection Profiles are not defined only some general security requirements can be specified.

### 2.3.1 Access Control

The secure environment shall be able to enforce security policies using access control. In general access control has to fulfill two requirements which depend on the point of view:

1. The system expects that access control *controls* actions of subjects (or programs running on behalf of subjects) in such a way that they cannot bypass access control rules or attack system services.

2. Users expect that access control *protects* their objects against external attacks.

Figure 2.2 explains the expected behaviour of access control which controls and protects programs/user objects. A program running under control of a user must

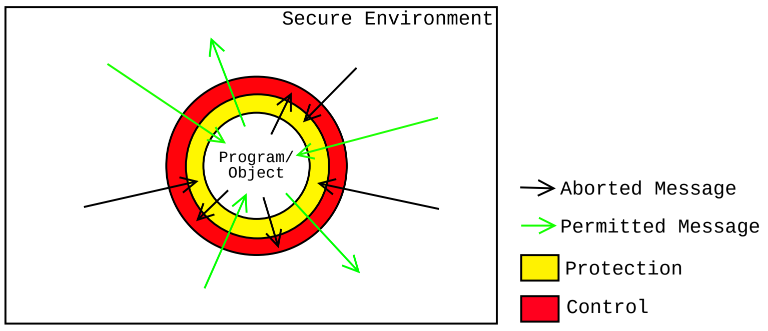

Figure 2.2: Access control protects and controls objects/programs.

not have more permissions than the user to guarantee the enforcement of global security policies. Therefore access control of the secure environment has to enforce that programs keep permissions of their owners. In contrast, users may want to refine access rules on their own. Thus access control should provide a mechanism to refine access to objects.

One way to define access control rules of global security policies is to use role-based access control model as assumed by the *Common Criteria*. A sample role hierarchy is suggested by the next subsection.

### 2.3.2 Roles

The goal of this work is to develop a security-policy independent secure environment. But to evaluate IT products using the *Common Criteria* requires the definition of security-policies and roles. To make this possible this section suggests a very simple role hierarchy which can easily be extended by later improvements. A short explanation of suggested roles follows, outlined by Figure 2.3:

**ANONYMOUS** An unknown subject which has only the permission to change his role (open a new session).

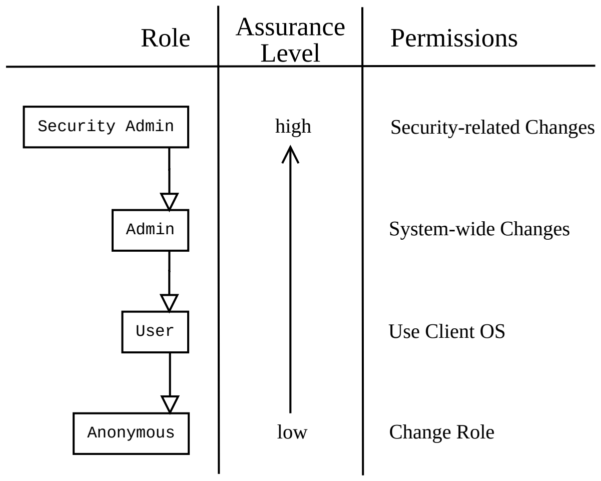

Figure 2.3: Suggestion of a simple role hierarchy.

**USER** Defines permissions and responsibilities of all subjects which have permission to use the *Client OS*. The USER role also defines a set of permissions to use services of the secure environment without further authentication. To be able to restrict access to the *Client OS* every user has to be authenticated by the secure environment. Therefore it is not necessary to distinguish between users which have the permission to access the secure environment and those who don't. More than one subject has the permission to use the role USER, because I decided to develop a multi-user system. This may be changed if the secure environment is ported to a PDA system (Section 4.4.7 discusses a solution to design the secure environment in such a way that it is possible to use it for one- and multi-user environments).

**ADMIN** Defines activities to invoke frequently used system management functions, e.g. system-wide installation of new software or user-management. The ADMIN role is not allowed to do actions which can disturb the security of the system.

**SECURITY-ADMIN** This role has permissions to adjust all security relevant management functions which can disturb the security of the secure environment. Examples are modifications/updates of core components and changes of the global security policy.

The separation of administrator rights into ADMIN and SECURITY-ADMIN makes it possible to use the system in a potentially untrusted environment (e.g. if a company distributes mobile devices to its employees which are able to install new software using the ADMIN role, but cannot bypass global security adjustments). Further roles can be defined to restrict access to subsets of users.

# Chapter 3

# System Definition

This chapter contains a precise analysis of functional and security-related requirements of *use cases* and extracted subservices/properties. Goals of this chapter are to collect and order the various requirements and, if possible, divide them into subsystems, classes, attributes, operations, links and aggregations. User-interfaces and implementation-related problems have to be ignored completely.

Results of this chapter are summarized by an OOA model presented in Section 3.5, which also compares this model with the preliminary high-level PERSEUS architecture. Together with the list of required functions (a complete version can be found in Appendix B) it should be possible to design and implement the product.

In order to avoid defining security-related terms and requirements on my own, the *Common Criteria* (Section 3.1) are presented first, which treats basic security requirements and defines concepts which can then be used by the analysis of the *use cases* and following sections. Hasty readers may skip this section and look up if a concept or word is unclear.

## 3.1 The Common Criteria

This section discusses basic requirements necessary to develop a secure environment as described in Section 1.1. Another task of this section is to provide a set of definitions to prevent misunderstandings. The security functional requirements of the *Common Criteria for Information Technology Security Evaluation* [37], short *CC*, is used as a guide for development of the secure environment. In addition, I hope that later evaluation of secure components is easier if both, defined terms and concepts, are used continuously in the analysis, design and implementation phase.

### 3.1.1 Introduction

The *Common Criteria* is presented as a set of three distinct but related parts; the first part is the introduction to the *CC* and contains the general model. It also presents constructs for writing the high-level specifications *Protection Profile* (PP) and *Security Target* (ST). The second part contains a list of functional components to describe functional requirements of *Targets of Evaluation* (TOE). Part three establishes a set of assurance components to be able to express assurance requirements for TOEs.

Part 2 of the *Common Criteria* divides security functional requirements of *TOE*s into 11 different classes of families containing different components. They are used to get a list of requirements that should be considered when designing a secure IT product. I considered only those requirements which seem to be necessary to develop a secure PDA as informally described in Section 1.1. This examination cannot be complete, because not all design criteria are currently available. Evaluation of an IT product, based on the *CC*, requires a complete specification of security policies and functions, and both are currently not available. If they are available, developers should re-evaluate this section, with respect to all three parts of the *Common Criteria.*

The next subsections contain short descriptions and examinations of security requirements of the *Common Criteria* part 2. If the *CC* desires definition of operations, security policies or something like that, I changed the requirements in that way that the secure environment only has to be able to enforce these requirements. Already known restrictions and properties have been included and the appropriate parts of the security requirements (Appendix A) are marked in italics.

### 3.1.2 Class FAU: **Security audit**

Security auditing involves recognizing, recording, storing and analysing information related to security relevant activities. The resulting audit records can be examined to determine which which security relevant activities took place and who is responsible for them. This class is not considered by this work.

### 3.1.3 Class FCO: **Communication**

This class provides two families specifically concerned with assuring the identity of a party participating in a data exchange. These families ensure that an origin cannot deny having sent the message, nor can the recipient deny having received it. This class has also been ignored, because higher protocols can provide these requirements if necessary.

### 3.1.4 Class FCS: **Cryptographic support**

To satisfy high-level security objectives the *TOE* may employ cryptographic functionality. Some examples of high-level security objectives required to build a secure environment are: Identification and authentication, trusted channels and data separation. Required functions have to be defined by *use cases* of Section 2.2. All specified functions are listed in Appendix B.3.2.

#### Family FCS_CKM: **Cryptographic key management**

*Description:* FCS_CKM defines requirements for cryptographic key generation (FCS_CKM.1), cryptographic key distribution (FCS_CKM.2), cryptographic key access (e.g. backup, escrow, archive, recovery) (FCS_CKM.3) and cryptographic key destruction FCS_CKM.4).

*Analysis:* This family is required by FCS_COP. Algorithms and standards to be used have to be decided later.

*Management:* Changes to cryptographic key attributes as explained in Section 3.2.1.

#### Family FCS_COP: **Cryptographic operation**

*Description:* The FCS_COP family requires functions to perform cryptographic operations with a specified cryptographic algorithm and cryptographic key size that meets a standard.

*Analysis:* Cryptographic operations are required by *Use Case* "Signing a document" and to provide authentication modules. In order to perform secure external communication channels symmetric encryption and decryption operations are necessary and to distribute symmetric keys asymmetric encryption is required. Section 3.2 contains an extended discussion of required cryptographic operations.

### 3.1.5 Class FDP: **User data protection**

This class contains families specifying requirements for *TOE Security Functions* and *TOE Security Function Policies* related to protecting user data. User data to be protected is the private key used to sign documents, thus this class is required. All required security functions are listed in Appendix B.3.3. This class is composed of the following families:

#### Family FDP_ACC: **Access control policy**

*Description:* This family identifies the access control *SFP*(s) (by name) and defines the scope of control of the policies that form the identified access control portion of the *TSP*.

*Analysis:* I don't want to make special assumptions on the access control policies at this point of development. To be able to support all upcoming access control policies the *reference monitor* has to be designed to be able to cover all operations (FDP_ACC.2) and it should be possible to have more than one access control policy in parallel.
To evaluate the design of the *reference monitor* the prototype should implement a sample policy to show that it works. Some considerations on access and information flow control are separated into Section 3.4.2.

#### Family FDP_ACF: **Access control functions**

*Description:* This family describes the rules for the specific functions that can implement access control policy named in FDP_ACC which specifies the scope of the policy.

*Analysis:* Because no SFP is defined by family FDP_ACC, currently no rules can be described.

#### Family FDP_DAU: **Data authentication**

*Description:* This family provides methods to provide guarantees of the validity of a specific unit of data that can subsequently be used to verify that the information content has not been forged or fraudulently modified. In contrast to class FAU, this family is intended to be applied to "static" data rather than data that is being transferred.

*Analysis:* Although not mentioned explicitly I think that this family only deals with user data. If necessary this requirement can be satisfied by higher protocols like hash or signature functions.

*Management:* The assignment or modification of the objects for which data authentication may apply could be configurable to the system.

Family FDP_ETC: **Export to outside TSF control**

*Description:* This family provides functions for exporting user data from the *TOE* such that its security attributes and protection either can be explicitly preserved or can be ignored once it has been exported.

*Analysis:* As outlined in Use Case "Generation of a new key pair" (Section 2.2.2) public keys have to be exported outside the secure environment. It contains no security attributes, thus FDP_ETC.1 should suffice.

Family FDP_IFC: **Information flow control policy**

*Description:* Provides subset information flow control FDP_IFC.1 or complete information flow control FDP_IFC.2.

*Analysis:* Information flow, especially covered channels, is a very complex and security relevant problem and Section 3.4.3 contains a short discussion about this topic. The system should be designed to be able to satisfy every upcoming information flow control policy.

Family FDP_ITC: **Import from outside TSF control**

*Description:* This family defines mechanisms for introduction of user data into the *TOE* such that it has appropriate security attributes and is appropriately protected.

*Analysis:* Required to be able to install new applications. Associated security attributes could be derived from third party certificates. I decided to define installed applications as user data to make it possible that users can install applications locally.

*Management:* The modification of the additional control rules used for input by the security-admin role.

Family FDP_ITT: **Internal TOE transfer**

*Description:* This family provides requirements that address protection of user data when it is transferred between parts of the *TOE* across an internal channel.

*Analysis:* In contrast to the informal description of this family the requirements use the term "physically separated parts of the TSF". I define the address spaces and separated hardware components of the PDA to be non-separated.
Upcoming requirements of internal transfers can be satisfied by class FTP "Trusted Paths/Channels" (e.g. between smartcard and PDA) and by family FPT_PHP "TSF physical protection".

Family FDP_RIP: **Residual information protection**

*Description:* This family addresses the need to ensure that deleted data is no longer accessible, and that newly created objects not contain information that should not be accessible.

*Analysis:* FDP_RIP.1 is required to protect, e.g., deleted secret keys. It is not necessary to protect, e.g., installed applications, thus FDP_RIP.2 is not a must. It is important not only to consider object creation/deletion. This kind of problem occurs if, e.g., a virtual memory page is reallocated.

*Management:* The choice of when to perform residual information protection (i.e. upon allocation or deallocation could be made configurable within the TOE.

Family FDP_ROL: **Rollback**

*Description:* The rollback operation involves undoing the last operation or a series of operations, bounded by some limit, such as a period of time, and return to a previous known state.

*Analysis:* In general a rollback mechanism is required to provide consistency and resistance against software bugs and hardware failures (see FDP_SDI). To prevent data inconsistence the *TSF* shall rollback into a consistent state. This topic is discussed in Section 3.2.7.

*Management:*

1. The boundary limit to which rollback may be performed could be a configurable item within the *TOE*.
2. Permission to perform a rollback operation could be restricted to a well-defined role.

Family FPD_SDI: **Stored data integrity**

*Description:* This family provides requirements that address protection of user data while it is stored within the *TSC*.

*Analysis:* It is important to monitor stored data integrity to be able to detect internal errors, e.g. hardware glitches or errors. An action the system has to invoke if it detects an error could be the rollback mechanism (FDP_ROL).
$\Rightarrow$ FDP_SDI.2 is required.

Family FDP_UCT: **Inter-TSF user data confidentiality transfer protection**

*Description:* This family defines the requirements for ensuring the confidentiality of user data when it is transfered using an external channel between distinct *TOE's* or users on distinct *TOE's*.

*Analysis:* Not required by the first prototype, but may be necessary if secure applications share data with external databases or use external services (e.g. a stock broking application).

Family FDP_UIT: **Inter-TSF user data integrity transfer protection**

*Description:* This family defines the requirements for providing the integrity for user data in transit between the *TSF* and another trusted IT product and recovering from detectable errors.

*Analysis:* Similar to FDP_UCT; not required by the first prototype, but may be necessary if secure applications share data with external databases or use external services (e.g. a stock broking application).

### 3.1.6 Class FIA: **Identification and authentication**

Families in this class address the requirements for functions to establish and verify a claimed user identity. Required security functions are summarized in Appendix B.3.4.

Family FIA_AFL: **Authentication failures**

*Description:* This family defines values for some number of unsuccessful authentication attempts and *TSF* actions in cases of authentication attempt failures.

*Analysis:* The first version of the secure environment does not have to restrict the number of unsuccessful authentication attempts, thus FIA_AFL is not required.

*Management:*

1. Management of the threshold for unsuccessful authentication attempts.
2. Management of actions to be taken in the event of an authentication failure.

Family FIA_UAD: **User attribute definition**

*Description:* All authorized users may have a set of security attributes, other than the user's identity, that is used to enforce the *TSP*. This family defines requirements for associating user security attributes with users as needed to support *TSP*.

*Analysis:* The type of security attributes (capabilities, clearance, rights) rarely depends on the model of access control. Currently only role-based access control is used, therefore only roles can be assigned.

*Management:* If so indicated in the assignment, the authorized administrator might be able to define additional security attributes to users.

Family FIA_SOS: **Specification of secrets**

*Description:* This family defines requirements for mechanisms that enforce defined quality metrics on provided secrets and generate secrets to satisfy the defined metric.

*Analysis:* This family is currently not considered by this diploma thesis.

Family FIA_UAU: **User authentication**

*Description:* This family defines the types of user authentication mechanisms supported by the *TSF*.

*Analysis:* As demanded by Section 2.3.2, only authentication should be possible before the claimed identity of the user is authenticated (FIA_UAU.2 is required). A later version can additionally provide a help machanism belonging to the authentication procedure. Non-sharable authentication mechanisms are desirable, but currently not planned. SR 151 is only an example of FIA_UAU.5.2. Re-authentication (FIA_UAU.6) is required in two cases: First, if the user requests the TSF to perform actions of a higher security level. Also the *TSF* shall force the user to re-authenticate after a time interval of inactivity (login timeouts). Protection of authentication feedback (e.g. the number of characters typed) (FIA_UAU.7) is required but should be configurable to help disabled persons.

*Management:*

1. Management of the authentication data by the administrator.
2. Management of the authentication data by the user associated with this data.
3. Management of authentication mechanisms.
4. Management of the rules for authentication
5. If an authorized administrator could request re-authentication, the management includes a re-authentication request.
6. Management of authentication feedback.

#### Family FIA_UID: **User identification**

*Description:* This family defines the conditions under which users shall be required to identify themselves before performing any other actions that are to be mediated by the *TSF* and which require user authentication.

*Analysis:* As defined in family FIA_UAU users have to identify themself before performing any other action.

*Management:* Management of the user identities.

#### Family FIA_USB: **User-subject binding**

*Description:* An authenticated user, in order to use the *TOE*, typically activates a subject. The user's security attributes are associated (totally or partially) with this subject. This family defines requirements to create and maintain the association of the user's security attributes to subjects acting on the user's behalf.

*Analysis:* This family deals with the problem of transitivity of permissions. A server which acts on behalf of a user has to consider appropriate permissions. *Example:* An encryption service can be invoked by all users of the system, but only the owner of a specific secret key is allowed to use it. Therefore the encryption service has to inherit the user's permission to use that key.
The first prototype can bypass this problem by enforcing re-authentication whenever a user uses a secure service.

*Management:* An authorized administrator can define default subject security attributes.

### 3.1.7 Class FMT: **Security management**

This class is intended to specify the management of several aspects of the *TSF*: security attributes, *TSF* data and functions. The different management roles and their interaction, such as separation of capability, can be specified. All requirements have been listed in Appendix B.3.5.

#### Family FMT_MOF: **Management of functions**

*Description:* This family allows authorized users control over the management of functions in the *TSF*.

*Analysis:* The ability to change the behaviour of security related functions should be restricted to one or more administrator roles to prevent users from deleting system data. Obviously FMT_MOF.1 is required. Because it is currently not possible to list all system functions only general statements have been made.

*Management:* Managing the group of roles that can interact with the functions in the *TSF*.

### Family FMT_MSA: **Management of security attributes**

*Description:* This family allows authorized users control over the management of security attributes.

*Analysis:* FMT_MSA.1 is required to assign security attributes to individual users/subjects. FMT_MSA.2 is required by all families of class FCS. Because security attributes are currently not available, they have been separated into critical (e.g. default values) and non-critical (e.g. priorities) attributes.

*Management:* Managing the group of roles that can interact with the security attributes.

### Family FMT_MTD: **Management of *TSF* data**

*Description:* This family allows authorized users (roles) control over the management of the *TSF* data.

*Analysis:* No TSF data has been defined so far, therefore I divided them into security-related and system-wide data. Also actions to be taken are currently not defined because they are security-policy dependend.

*Management:*

1. Managing the group of roles that can interact with the *TSF* data.
2. Managing the group of roles that can interact with the limits on the *TSF* data.

### Family FMT_REV: **Revocation**

*Description:* This family addresses revocation of security attributes for a variety of entities within a *TOE*.

*Analysis:* Capabilities, assigned to users, have to be removed sometimes. Therefore the access control mechanism should provide a function to revoke security attributes. Revocation rules themself are security-policy dependend.

*Management:*

1. Managing the group of roles that can invoke revocation of security attributes.
2. Managing the list of users, subjects, objects and other resources for which revocation is possible.
3. Managing the revocation rules.

#### Family FMT_SAE: **Security attribute expiration**

*Description:* This family addresses the capability to enforce time limits for the validity of security attributes.

*Analysis:* Not required by the first prototype.

#### Family FMT_SMR: **Security management of roles**

*Description:* This family is intended to control the assignment of different roles to users.

*Analysis:* FMT_SMR.1 is required because roles are used by a lot of components. Restrictions on assignment of security roles are currently not required. FMT_SMR.3 is required to prevent assignment of any roles to the *Client OS*.

*Management:* Managing the group of users that are part of a role.

### 3.1.8 Class FPR: **Privacy**

This class contains privacy requirements. These requirements provide a user protection against discovery and misuse of identity by other users.

#### Family FPR_ANO: Anonymity

*Description:* This family ensures that a user may use a resource or service without disclosing the user's identity. Anonymity is not intended to protect the subject identity.

*Analysis:* I decided to ignore this requirements for the first prototype. But it may be required to prevent covered information flows (see Section 3.4.3).

#### Family FPR_PSE: Pseudonymity

*Description:* This family ensures that a user may use a resource or service whiteout disclosing its user identity, but can still be accountable for that use.

*Analysis:* Similar to the family discussed above it may be necessary to prevent unwished information flows, but it is currently not considered.

#### Family FPR_UNL: Unlinkability

*Description:* This family ensures that a user may make multiple uses of resources or services without others being able to link these uses together.

*Analysis:* May be required to prevent unwished information flows, but ignored by this work.

#### Family FPR_UNO: Unobservability

*Description:* This family ensures that a user may use a resource or service without others, especially third parties, being able to observe that the resource or service is being used.

*Analysis:* May be required to prevent unwished information flows, but ignored by this work.

### 3.1.9 Class FPT: **Protection of the TSF**

This class contains families of functional requirements that relate to the integrity and management of the mechanisms of the *TSF* and to the integrity of the *TSF* data (independent of the specific contents of the *TSF* data). The list of requirements necessary to build the secure environment is listed in Appendix B.3.6.

#### Family FPT_AMT: **Abstract machine test**

*Description:* This family defines requirements for the *TSF* to perform testing to demonstrate the security assumptions made about the underlying abstract machine upon which the *TSF* relies.

*Analysis:* "Underlying abstract machine" in this case means the hardware and the microkernel. This family is required but its contents are out of the scope of this document. This work assumes that the hardware and the kernel work correctly.

#### Family FPT_FLS: **Fail secure**

*Description:* The requirements of this family ensure that the *TOE* will not violate its *TSP* in the event of identified categories of failures in the *TSF*.

*Analysis:* FPT_FLS.1 is required to protect the *TSP* and the integrity of the *TSP* data. It is also required by FDP_SDI and FRU_FLT (fault tolerance). Example classes of failures are hardware failures (CPU, memory, disk) or software failures. Software can be separated into core components and other components. I decided only to consider software failures here.

#### Family FPT_ITA: **Availability of exported *TSF* data**

*Description:* This family defines the rules for the prevention of loss of availability of *TSF* data moving between the *TSF* and a remote trusted IT product.

*Analysis:* Not required because no *TSF* data has to be exported.

#### Family FPT_ITC: **Confidentiality of exported *TSF* data**

*Description:* This family defines the rules for the protection from unauthorized disclosure of *TSF* data during transmission between the *TSF* and a remote IT product.

*Analysis:* Not required because no TSF data has to be exported.

#### Family FPT_ITI: **Integrity of exported *TSF* data**

*Description:* This family defines the rules for the protection, from unauthorized modification, of *TSF* data during transmission between the *TSF* and a remote trusted IT product.

*Analysis:* Not required because no *TSF* data has to be exported.

Family FPT_ITT: **Internal *TOE* *TSF* data transfer**

*Description:* This family provides requirements that address protection of *TSF* data when it is transferred between separate parts of a *TOE* across an internal channel.

*Analysis:* In class FDP I assumed the *TOE* not to be separated. Thus this family is not required.

*Management:* -

Family FPT_PHP: ***TSF* physical protection**

*Description:* *TSF* physical protection components refer to restrictions on unauthorized physical access to the *TSF*, and to the deterrence of, and resistance to, unauthorized physical modification, or substitution of the TSF. Protection against non-physical attacks is examined, e.g., by the next family.

*Analysis:* Protecting the *TSF* from physical tampering and interference is required but out of the scope of this document. This family is required, e.g. by Section 3.3.4.

*Management:* -

Family FPT_RCV: **Trusted recovery**

*Description:* The requirements of this family ensure that the *TSF* can determine that the *TOE* is started up without protection compromise and can recover without protection compromise after discontinuity of operations.

*Analysis:* It is very important to guarantee a secure initial boot mechanism and a secure recovery boot mechanism after an error. At least a manual recovery mechanism (a defined role can recover the system into a secure state) is required which provides recovery to guarantee a secure state. Further discussions on this topic can be found in Section 3.3.4.

*Management:* Management of roles which have the permission to restore the system state.

Family FPT_RPL: **Replay detection**

*Description:* This family addresses detection of replay for various types of entities and subsequent actions to correct.

*Analysis:* Required to be able to provide atomic operations which may be required to ensure that a message is sent only once. This criteria has to be considered by all components which provide persistence.

*Management:*

1. Management of the list of identified entities for which the replay shall be detected.
2. Management of the list of actions that need to be taken in case of a replay.

### Family FPT_RVM: **Reference mediation**

*Description:* The requirements of this family address the "always invoked" aspect of a traditional reference monitor. The goal of this family is to ensure, with respect to a given *SFP*, that all actions requiring policy enforcement are validated by the *TSF* against the *SFP*.

*Analysis:* Required to provide a reference monitor as demanded by the *Common Criteria*, Part 2.

### Family FPT_SEP: **Domain separation**

*Description:* The components of this family ensure that at least one security domain is available for the *TSF's* own execution and that the *TSF* is protected from external interferences and tampering by untrusted subjects.

*Analysis:* FPT_SEP.3 is required to provide a reference monitor as defined in the *Common Criteria*, Part 2.

### Family FPT_SSP: **State synchrony protocol**

*Description:* This family ensures that two distributed parts of the *TOE* have synchronized their states after a security-relevant action.

*Analysis:* Not required because the secure environment is not distributed.

### Family FPT_STM: **Time stamps**

*Description:* This family addresses requirements for a reliable time stamp function within the *TOE.*

*Analysis:* Required for the purpose of security attribute expiration (expiration of a secret key).

*Management:* Management of the time.

### Family FPT_TDC: **Inter-TSF TSF data consistency**

*Description:* This family defines the requirements for sharing and consistent interpretation of security attributes between the *TSF* of the *TOE* and another trusted IT-product.

*Analysis:* Not required because no *TSF* internal data has to be shared with another trusted IT product.

### Family FPT_TRC: **Internal *TOE TSF* data replication consistency**

*Description:* The requirements of this family are needed to ensure the consistency of *TSF* data when such data is replicated internal to the *TOE.*

Analysis: This kind of problem occurs if more than one subsystem accesses *TSF* data and/or is able to modify it, or if *TSF* data is shared between different subsystems and has to be synchronized.

#### Family FPT_TST: **Self test**

Description: This family defines the requirements for the self-testing of the *TSF* with respect to some expected correct operation.

Analysis: It may be important to test the processor, memory and random bit generators, but self-testing of these components is out of the scope of this document.

### 3.1.10 Class FRU: **Resource utilization**

This class provides three families that support the availability of required resources such as processing capabilities and/or storage capacity. Hardware resources to be mediated are interrupts, tasks, BIOS, I/O ports and unique resources like memory, display, keyboard, harddisk(s), printer, etc. All required functions are listed in Appendix B.3.7.

#### Family FRU_FLT: **Fault tolerance**

*Description:* The requirements of this family ensure that the *TOE* will maintain correct operation even in the event of failures.

*Analysis:* At least FRU_FLT.1 is required to provide resistance against software bugs of non-core components. If hardware bugs occur, software cannot ensure correct operation. To provide correctness of additional devices, e.g. harddisks, is out of the scope of this document.

#### Family FRU_PRS: **Priority of services**

*Description:* The requirements of this family allow the *TSF* to control the use of resources within the *TSC* by users and subjects such that high priority activities within the *TSC* will always be accomplished without undue interference or delay caused by low priority activities.

*Analysis:* It is very important to guarantee the execution of core components of the system. Therefore at least all resources required by these core components have to be mediated by trusted components. This requires FRU_PRS.1.

*Management:* Assignment of priorities to each subject in the *TSF*.

#### Family FRU_RSA: **Resource allocation**

*Description:* The requirements of this family allow the *TSF* to control the use of resources by users and subjects such that denial of services will not occur because of unauthorized monopolization of resources.

*Analysis:* In contrast to family FRU_PRS this family required the mediation of all non-sharable resources by fully trusted core components. At least the definition of maximum quota (FRU_RSA.1) is required to prevent unauthorized monopolization of resources.

*Management:* Specifying maximum limits for a resource for groups and/or individual users and/or subjects by the administrator.

### 3.1.11 Class FTA: TOE access

This class specifies functional requirements for controlling the establishment of a user's session. Required functions have been listed in the Appendix B.3.8.

#### Family FTA_LSA: Limitation on scope of selectable attributes

*Description:* This family defines requirements to limit the scope of session security attributes that a user may select for a session.

*Analysis:* Limitations of session security attributes are currently not necessary.

#### Family FTA_MCS: Limitation on multiple concurrent sessions

*Description:* This family defines requirements to place limits on the number of concurrent sessions that belong to the same user.

*Analysis:* Currently not considered.

#### Family FTA_SSL: Session locking

*Description:* This family defines requirements for the *TSF* to provide the capability for *TSF*-initiated and user-initiated locking and unlocking of interactive sessions.

*Analysis:* Session locking after a specified period of inactivity should be supported by the secure environment, thus FTA_SSL.1 is required. Also the user should be able to lock a session. This requires FTA_SSL.2. A mechanism that should occur prior unlocking a locked session is defined by FIA_UAU.6 "re-authentication".

*Management:*

1. Specification of the time of user inactivity after which the lock-out occurs for an individual user.
2. Specification of the default time after which the lock out occurs.
3. Management of the events that should occur prior to unlocking the session.

#### Family FTA_TAB: TOE access banners

*Description:* This family defines requirements to display a configurable advisory warning message to users regarding the appropriate use of the TOE.

*Analysis:* Not required at the moment.

*Management:* Maintenance of the banner by the authorized administrator.

#### Family FTA_TAH: ***TOE*** access history

*Description:* This family defines requirements for the *TSF* to display to a user, upon successful session establishment, a history of successful and unsuccessful attempts to access the user's account.

*Analysis:* Currently ignored.

#### Family FTA_TSE: **TOE session establishment**

*Description:* This family defines requirements to deny a user permission to establish a session with the *TOE*.

*Analysis:* Currently ignored.

### 3.1.12 Class FTP: **Trusted path/channels**

Families in this class provide requirements for a trusted communication path between users and the *TSF*, and for a trusted communication channel between the *TSF* and other trusted IT products. Required functions are summarized in Appendix B.3.9.

#### Family FTP_ITC: **Inter-TSF trusted channel**

*Description:* This family defines requirements for the creation of a trusted channel between the *TSF* and other trusted IT products for the performance of security critical operations.

*Analysis:* Required by later versions to provide inter-*TSF* user data integrity (FDP_UIT) and confidentiality (FDP_UCT). Currently required, e.g. by *Use Case* "Download of new content", to initiate a (secure) communication to a trusted IT product of the content provider.

*Management:* Configuring the actions that require trusted channels, if supported.

#### Family FTP_TRP: **Trusted path**

*Description:* This family defines the requirements to establish and maintain trusted communication to or from user and the TSF.

*Analysis:* Required by FDP_UCT and FDP_UIT. A trusted path is required for any security-relevant interaction. The trusted path has to guarantee, e.g., that untrusted applications cannot pretend to be trusted ones. This topic is discussed by Section 3.4.8.

## 3.2 Analysis of Use Cases

The following subsections contain an analysis of the *use cases* introduced in Chapter 2. At the end of this section a complete list of required system functions is specified, listed in Appendix B.4, which may overlap with some security requirements defined by the last section. Extracted functions have not been numbered continuously to be able to insert new ones.

### 3.2.1 Signing an email

In order to sign documents created by *Client OS* applications (e.g. a mailtool) it is necessary that the *secure environment* provides a bidirectional communication path to the *Client OS* and that applications can use it. Figure 3.1 outlines in a simplified manner the data-flow of the sign function requested by *Use Case* "Signing an email" if a *Client OS* application has sent a document to the secure environment. The input parameters of the main function are a plain ASCII message and a unique secret-key identifier.

To sign documents and generate sendable messages, four functions and an underlying key management interface are required. Analysis of signature creation

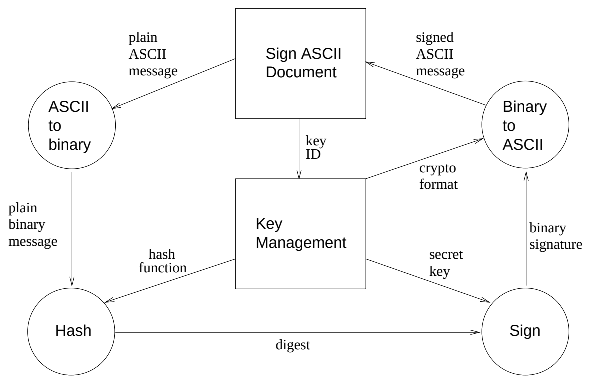

Figure 3.1: Data-flow while signing an ASCII text.

processes of crypto-formats OpenPGP [2] and S/MIME v.3 [1] in [30] have shown that nearly all functions depend on the underlying crypto-format which itself is defined by the secret key. This should be considered whenever interfaces are to be defined.

Another restriction concerning interfaces of signature services is given by smartcards. To make the use of cryptographic smartcard functions transparent to the user, the defined interface must not be higher granulated than the interface given by smartcards.

The next restriction is given by the answer of the question which function/data has to be protected by the secure environment and which not. Of course secret-key data has to be protected, and thereby operations which operate on this data, at least the secret key operation of the digest. But because the secure environment should be able to show documents to users for verification, also hash and convert functions have to be provided by the secure environment.[1]

In order to check that the *Client OS* has sent the correct document to the secure environment, it should be able to show them for verification. Therefore Section 3.4.8 discusses requirements for a general document viewer.

The result of this subsection is the establishment that the secure environment has to provide a bidirectional communication path between secure environment and *Client OS* applications [F 3] which can be used without modification of *Client OS* applications, a trusted sign function [F 5] which accepts a plaintext and a key-id as arguments and contains a crypto format independent key management service [F 10]. Internally the secure environment has to provide a hash function [F 60] which converts binary messages into digests, a signature function [F 65] which signs digests using a public key algorithm and two functions, [F 70] and [F 71], to convert an ASCII text into binary data and vice versa. All functions have to meet at least one public crypto standard.

## 3.2.2 Generation of a new key pair

Of course key-pairs have to be generated by the secure environment to protect it against malicious programs and a buggy *Client OS*. Nevertheless a lot of requirements have to be considered:

**Security Level.** In my opinion the *Client OS* must not translate abstract security levels into key generation parameters, because of the following reason: Abstract security levels are used to hide algorithm-specific details. If users decide to generate a "very secure" key, we cannot assume

[1]This opens the question, how binary data could be verified?

that they understand verification messages of the secure environment like *"Shall I create an X key with key-length Y?"*. Instead, the secure environment has to ask *"Shall I create a [very secure] key?"*. Of course, other parameters may be shown or even be changeable (optional). Thus I decided that the secure environment has to provide abstract security levels.

**Key Id.** Although *Client OS* and secure environment can use any internal data type to identify keys, the secure environment has to provide a unique key id locally in a user-readable format. This is necessary to prevent a verification message like *"Are you sure you want to sign this document using key 0xa4f6202c?"*.

**Random Generator.** Secure creation of cryptographic keys requires cryptographically strong random generators. A framework should provide an interface which returns crypttographic strong random bytes on demand.

**Public Key.** The user also needs access to the public key. Often (always?) the public key data can be extracted from secret key data, but in all cases some public key data is required.

**SmartCards.** In order to make it possible to use smartcards, their key-generation interfaces should also be considered.

New requirements to the secure environment's key management service: A key-generation function [F 15] with no arguments[2] which provides abstract security levels and key identifiers in a user-readable format, a service [F 20] which creates cryptographically strong random generators and a function [F 25] which creates random values from user inputs. Additionally a function [F 30] is required which returns public keys on demand.

### 3.2.3 Opening/Closing a session

Users who open new sessions have to be authenticated to ensure that security policies can be enforced. If sessions are closed it would be nice to be able to store their state to be able to load it if a new session is opened by the same subject (session management).

The secure environment has to provide a function to open a session which uses authentication mechanisms [F 85] and a function to close the current session [F 90]. The locking mechanism, demanded by [SR 340] - [SR 346], is equal to closing a session if a session management function is provided. Therefore an explicit locking mechanism is obsolete (but of course it can be implemented explicitly). To fulfill [SR 155] the system has to close the session after a specified time of user inactivity. This makes it necessary to have an internal timer [F 95]. To be user-friendly the system should warn users before the session is closed.

### 3.2.4 Activate/Deactivate System

If the system is activated it should reload into its latest consistent state, but then invoke an authentication mechanism to prevent misuse.

Therefore deactivating the system is similar to invoking function [F 80]. If the system is activated (turned on) the default (and protected!) behaviour should be to invoke function [F 81]. The secure environment has to enforce that the

[2]At this point of development I ignore cryptographic algorithms like fail-stop or group signatures which need extra data to create a new key pair.

activated system invokes an authentication mechanism (explicitly or by closing the current session before the system is deactivated). It should be possible to grant the permission to deactivate the system to a specific role.

### 3.2.5 Installation of new Software

To install new software the program data of the new application/service has to be copied into the secure environment by an internal function [F 35] which makes it possible to access external (e.g. ftp server) data. The permissions of the software to be installed depends on the rights of the subject which invokes the installation process and the trust in the correctness of the software to be installed.

If users shall trust the secure environment they have to make sure that used components work as expected. They may assign different trust levels to different services and define rules how trust levels have to be assigned to components. The trust level of components may depend on different things, e.g.:

1. Whether the source code is available or not.
2. The trust in the provider of the software.
3. Third party certificates which guarantee correctness of the software.
4. The trust in the third party which provides certificates.
5. The number of certificates available for the software.

Finally only users can assign levels of trust to contents, but the secure environment can help to derive suggestions [F 55]. Starting a subsystem heavily depends on the underlying system and development tools (e.g. linker) and has to be provided by a separate modular function [F 52]. External function are [F 40] to start a new subsystem and a function [F 45] to remove an existing subsystem. [F 40] should accept the destination address where the subsystem's code and further information to derive the trust level can be found, and [F 45] should accept a unique identifier of the subsystem.

### 3.2.6 Updating a Service

Extends *Use Case* "Installation of new Software". If a service is updated two things have to be considered:

1. The internal ID (address) of the service may change.
2. The updated service may provide a different interface (e.g. higher or lower version).

Because of 1.) both references to subsystems and the access control mechanism (see Section 3.4.2) must not depend on internal subsystem-IDs like task or thread IDs [SR 400]. The second point makes is necessary to integrate a versioning mechanism [F 78] to the interface description of a subsystem (see Section 3.4.1). Additionally a naming service [F 75] is required which maps external references into internal data types and vice versa.

### 3.2.7 Rolling back the System State

To be able to define granularity of rollback mechanisms, it has to be decided if it is possible to rollback only one subsystem or if the complete system state has to be rolled back if an error occurs. Also the design of the recovery mechanism heavily depends on the storage mechanism of persistent data discussed in Section 3.3.2. Section 4.5.2.1 introduces a model based on [26, page 5] which provides global persistence and can also be used as recovery mechanism. A new external function [F 95] is required to rollback the system state.

### 3.2.8 Enforcing Local Security Policies

Passive data of different security levels cannot interfere with each other, therefore it is not necessary to store them into separated protected domains, but of course security levels of these data must not be higher than the trust-level of the implementation of the management service (e.g. database). Services like databases or filesystems which manage large amounts of passive data should be able to enforce their own access control policy, as illustrated by *Use Case* 2.2.10, because system-wide security policies cannot understand semantics of operations provided by services. In contrast, the other (alternative) description of this use case is only a refinement of the system-wide access control rules and can be enforced by existing mechanisms, but this leads to frequent changes of these rules. Additionally this approach is very inefficient because the whole set of access control rules is used every time the access control checks permissions of every subsystem.

Because local subjects/objects should not be able to bypass system-wide access control policies only refinements can be allowed. This leads to a new security requirement [SR 299]: Subsystems should be able to refine global access control policies (by providing additional rules or enforcing their own security policy).

To be able to enforce local security policies or refine global ones, subsystems need some additional information to recognize and distinguish subjects. Thus a new internal system function [F 110] is required which makes security attributes assigned to subjects acting on behalf of users available to subsystems being accessed by these subjects.

## 3.3 Analysis of Security Requirements

This section contains additional analysis of security requirements of the *Common Criteria* presented in Section 3.1.

### 3.3.1 Client OS

The secure environment cannot grant capabilities to individual tasks of a *Client OS* without restricting security, because the *Client OS* is untrusted and a malicious client task could copy/modify capabilities. It is also impossible to re-identify a task, because the *Client OS* is able to modify the task's id. This results in two restrictions:

1. It is not possible to distinguish between different users of the *Client OS*, because they can only be identified by their tasks. Therefore all users of one *Client OS* instance have the same permissions concerning the secure environment. Of course is it possible to distinguish them if the secure environment authenticates the given identification. This leads to the second restriction:

2. Every time tasks of the *Client OS* (re-)connect to the secure environment, they have to be (re-)authenticated.

### 3.3.2 Persistent Storage

As demanded by [SR 296] the secure environment provides separated domains to enforce independence between subsystems. In order to ensure that these protection mechanisms cannot be bypassed (e.g. if the system is halted and internal data is written to another storage) is to use persistent memory. If this is not possible trusted components should provide a persistent memory behaviour. Persistence provided by the secure environment has some additional advantages:

1. Subsystems have to be started only once, which is more intuitive to the user and unifies the behaviour of personal digital assistants (PDAs) containing static memory and computers with dynamic memory (e.g. PCs).

2. Application developers are relieved from designing and implementing data input/output mechanism to another persistent storage, which is less error prone.

3. Implementation of session management function is obsolete because persistence also stores the current state of all subsystems.

To be able to provide a persistent system behaviour on hardware containing dynamic memory two functions are required: One which stores the complete system state on a persistent storage (e.g. harddisk) [F80], and the other [F81] which reloads the system into the state stored on the persistent storage. The following sections describe the most common approaches to store data persistently:

***A flat file system.*** The most common kind of persistent storage used by a lot of operating systems [41]. Only simple data operations are provided, which makes it necessary to subsystems to separate operations (the application) and data (the files), and to convert the internal data structures into a linear file format. Also file systems are very error prone, because they only provide more or less reliable persistence of data and they cannot provide consistency and integrity themselves. Building a database based on a file system is very inefficient, because file access is very inefficient.

***A database.*** In contrast to file systems databases provide data consistence, integrity and persistence. Additionally some databases contain version-control and recovery mechanisms [8]. Unfortunately development of a database is very complex and another disadvantage is that developers themselves have to decide wether a variable has to be persistent or not.

***A library.*** Further approaches use libraries which support functions (or base classes) to create persistent objects [42]. Using this kind of library is often very complicated and error prone, because application developers themselves have to take care that all persistent data is handled by the library.

***Persistent Memory Pages.*** To provide a persistent behaviour memory pagers can be implemented in such a way that they store memory pages to a persistent storage and reload them on demand. This approach is very similar to the virtual memory provided by most CPUs, except that the complete system state (including page table and states of additional devices) has to be stored. Because some hardware functions (e.g. virtual pages, segmentation) can be used, it can be implemented very efficiently [26] [15].

The approach to provide persistent memory pages by memory pagers seems to have the most advantages. Therefore it is evaluated in Section 4.5.2.1. Because Chapter 4 illustrates that integration of functions which provide persistence can easily be integrated into the OOD model, Section 4.5.2.2 contains a discussion of this topic.

### 3.3.3 Providing Independence Between Subsystems

As demanded by [SR 296], the secure environment has to enforce independence between subsystems. Consequently, direct access to unsharable hardware resources of untrusted subsystems, e.g. the *Client OS*, has to be prevented. Figure 3.2 outlines the traditional operating system behaviour: It has full control over the system and allocates all system resources. Device drivers and core components are able to access the hardware directly. Two basic solutions to bypass this unacceptable state are

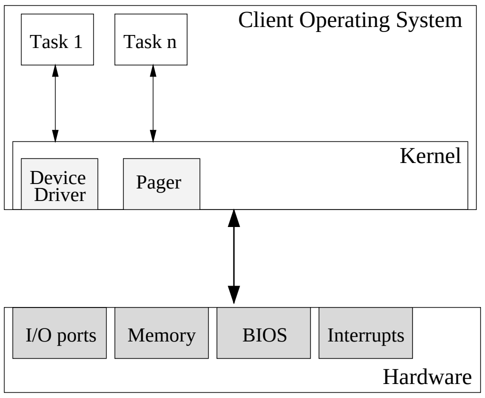

Figure 3.2: Abstract model of a traditional operating system.

possible:

1. Inserting a virtual hardware layer between microkernel and *Client OS* as demonstrated by Figure 3.3, like vmware[3] and its free clone freemware[4] do it. The virtual hardware layer catches all hardware accesses of the *Client OS* and remaps them into calls to trusted resource managers which consider security policies. This approach makes it possible to use every operating system designed for this hardware as *Client OS* without modifications. The loss of efficiency should be small, because only privileged system functions have to be emulated.

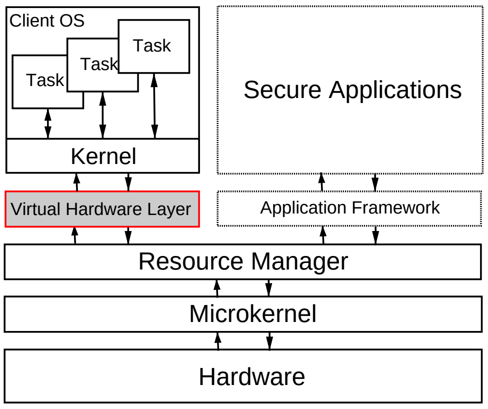

Figure 3.3: System design if a virtual hardware layer is inserted between microkernel and Client OS.

2. Replacing all components containing hardware access by stubs which access trustworthy drivers as illustrated by Figure 3.4. This decreases code size and increases speed, because every driver exists only once. On the other hand

the *Client OS* has to be modified and this is not possible in all cases, e.g. if commercial software products shall be used.

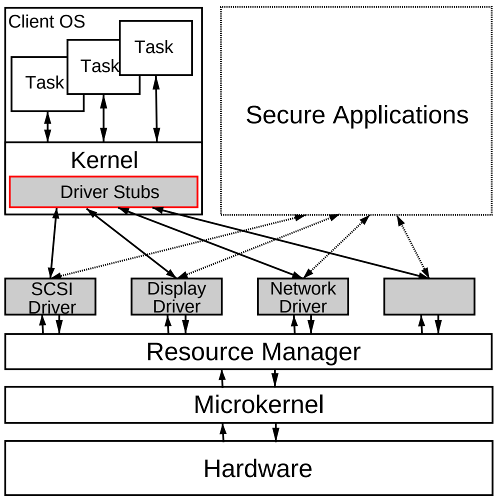

Figure 3.4: System design if the device drivers are separated into tasks of the microkernel.

3. A combination of solution one and two. E.g. some device drivers are replaced by stubs and the other accesses are emulated using a virtual layer.

The first solution seems to be the more flexible one, but the second is faster because privileged system functions do not have to have be emulated. Section 4.5.1 presents an example how a virtual hardware layer could be implemented, but because this is a too complex task for this diploma thesis and developers of the DROPS[5] project have experience in writing separated microkernel device drivers ([43], [31] and [32]) the second or third solution is preferred. The next subsections contain short discussions of resources which have to be protected.

#### 3.3.3.1 Limiting I/O Port Access

Only one trusted device driver should be able to use the same set of I/O ports at the same time. Two ways of I/O port addressing are commonly used by CPUs:

1. Separate I/O address spaces besides physical memory (e.g. Intel i386). Special CPU I/O instructions are used to access the I/O port.

2. Memory-mapped I/O (e.g. Motorola PowerPC). I/O ports appear in the address space of physical memory.

If the hardware supports separate I/O address space, (hardware-) access control with a high granularity is required to make sure that every I/O port can be accessed only by one subsystem. If memory mapped I/O is used, paging/segmentation mechanisms need a high granularity.

[3]http://www.vmware.com
[4]http://www.freemware.org
[5]http://os.inf.tu-dresden.de/drops

#### 3.3.3.2 Harddisk Driver

It is important to separate the *Client OS* and the harddisk driver(s) to prevent the *Client OS* from accessing and modifying disks/partitions of the secure environment. The *Client OS* should only be able to “see” virtual harddisks.

#### 3.3.3.3 Display Driver

The general way to share a display would be to provide virtual displays (windows) to all subsystems which need display access. The interface provided by the virtual displays has to be defined and it has to be decided how subsystems can access them.

#### 3.3.3.4 DMA channels

Even the use of DMA channels, if provided by the underlying hardware, has to be managed by a trusted component.

#### 3.3.3.5 Interrupts

A trusted component has to control reservation of interrupts and to manage interrupt-sharing.

#### 3.3.3.6 Tasks

If the number of active tasks/subsystems is limited by underlying components, permissions to start a new task has to be managed by a trusted component.

#### 3.3.3.7 Events

Some events, e.g. mouse clicks or keystrokes, have to be forwarded to the subsystem which currently has the focus (gets user input). Events are a more abstract view of a subset of other resources to be shared (e.g. interrupts). A trusted instance is required which synchronizes forwarding of events and the subsystem which uses the DisplayDriver.

### 3.3.4 Secure Booting

A secure booting mechanism has to provide the integrity of code (protection agains replacement/changes of components/configurations) and the integrity and confidentiality (protection against spoofing) of data.

Let us assume a bootloader is used which uses an encrypted configuration file and asks for a password (or a smartcard) on startup to be able to decrypt configuration files. To prevent attackers from replacing the bootloader to bypass its security mechanisms the whole persistent storage has to be protected (encrypted/signed) and the bootloader has to initialize appropriate device drivers with keys (which themselves are stored encrypted using a password entered on startup by the ADMIN) to decrypt/test stored data. This approach relies on the fact that the startup password is secure and that it is impossible to access the password. The second restriction could be guaranteed by deleting the password (or the whole bootloader) after decryption of the keys of the device drivers.

Another approach could prevent that attackers can access the persistent storage (e.g. to replace the bootloader). This makes it necessary to use some kind of hardware protection like BIOS passwords. I fear that we have to encrypt/sign the whole data in any case to protect information stored on the persistent storage, except if we use some kind of self-destroying harddisk or e.g. a device which uses (internally) dynamic memory and turns the power off if it detects external attacks.

Section 4.5.2 contains discussions about memory managers which use cryptographic functions to provide confidentiality and integrity.

## 3.4 Packages

*Packages* are used by *UML* to summarize other elements (e.g. classes). The concept of packages is required to build meaningful groups of elements to describe the system structure on a higher level.

### 3.4.1 Package: **Subsystem**

The package Subsystem contains elements to describe and identify subsystems (see Figure 3.5). In order to be able to store dependencies between subsystems general definitions of provided services are necessary. These descriptions have to be implementation-independent, therefore the class Service describes the Interface and the Version which is implemented by a subsystem. A subsystem can implement different services and/or different interface versions. The versioning scheme has to consider two types of modifications of interfaces:

1. changes (incompatible interface), hence called *major interface number.*
2. extensions (compatible interface), hence called *minor interface number.*

Consideration of patchlevel or similar features is not necessary because it is interface-independent.

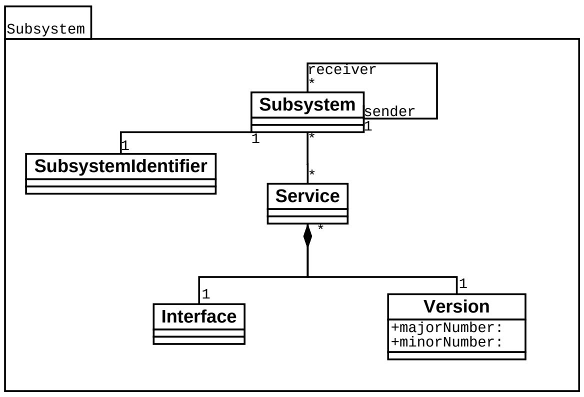

Figure 3.5: Components of the Subsystem package.

Figure 3.6 outlines an example scenario: Subsystem $x$ provides two services *s1* and *s2* which implement different versions *v1* and *v2* of the same interface *i2*. Subsystem $y$ provides only service *s2*, therefore it implements version *v2* of interface *i2*. Subsystem $z$ provides two services, it implements version *v1* of interface *i2* and version *v3* of interface *i1*.

### 3.4.2 Package: **AccessControl**

Access control can be logically divided into three different parts:

1. The *policies* (e.g. Bell-LaPadula [7]) are high level rules which determine how accesses are controlled and access decisions determined.

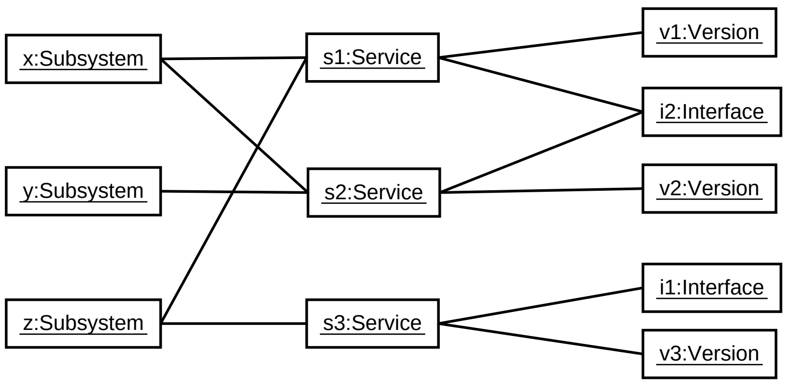

Figure 3.6: Example scenario describing relations between subsystems, services, interfaces and versions.

2. The *model* (e.g. RBAC [40]) defines the syntax of rules to express one or more policies.
3. The *mechanism*, also known as reference monitor, is low-level software and hardware functions which enforce the policies and guarantee that it cannot be bypassed.

Because access control policies depend on security policies and user wishes, it is out of the scope of this document to define access control policies or to decide which access control model should be used to define these policies. Therefore the goal of this section is to provide an access control *mechanism* which can be used to enforce different security policies. Demands on the behaviour of the *reference monitor* are:

1. It cannot be bypassed by other subsystems [SR 60].
2. A subsystem should only be able to use indirect [SR 400] and locally valid [SR 405] references to access other subsystems.
3. Global security policy to control subsystems, local policies to individually refine global policies and protect individual subsystems [SR 299].
4. Possibility for two subsystems to synchronize access to a third subsystem (bypass [SR 405]) if permitted [F 115] (an example is given in Section 4.3.2).
5. Local references of incoming and outgoing messages have to be synchronized [F 120] to be able to enforce local access control policies.

The general way to define an access control policy within an object-oriented analysis model would be to define an access control matrix $M_{AC}$:

$$S \times O \times M_o \rightarrow \{true, false\}$$

with the set of all subjects $S$, the set of all objects $O$ and the set $M_o$ of all methods accepted by object $o \in O$. Because security is a primary goal of a secure environment this section provides a more efficient model which is based on results of Section 2.3.1:

The global access control policy grants access permissions to subsystems which can be refined locally by access control policies, e.g. access control lists (ACL). ACLs can be enforced by further trusted components and/or by the subsystem themself. Figure 3.7 proposes an access control mechanism which corresponds to this model.

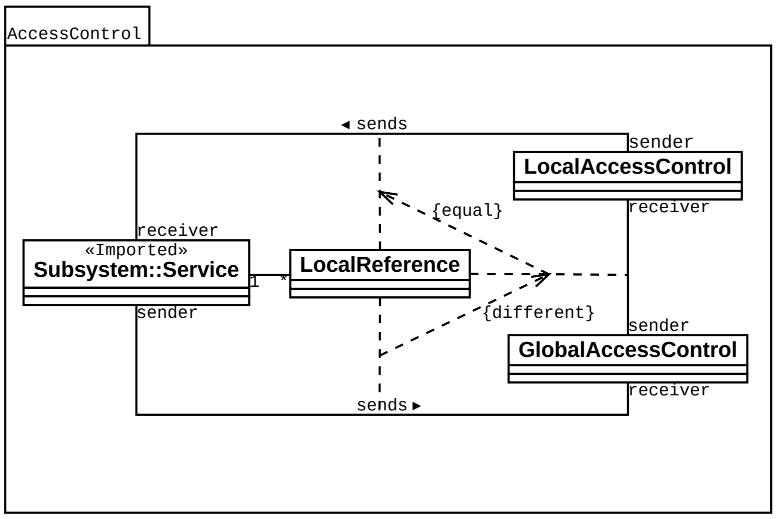

Figure 3.7: Components of the AccessControl package.

Subsystems can only access other subsystems via a GlobalAccessControl which enforces a global access control policy and maps LocalReferences of the sender into LocalReferences of the receiver. Messages are forwarded to the LocalAccessControl of the receiver's subsystem which is able to refine the global policy. Sender and receiver references are remapped before the LocalAccessControl is invoked to prevent information flow of global references if subsystems enforce local access control themselves. To fulfill the "always invoked" requirement [SR 60] an abstract *message redirection mechanism* enforces that all messages sent by a subsystem are forwarded to the GlobalAccessControl. Because it has to be a trusted component, users can trust it to forward messages to the appropriated LocalAccessControl.

### 3.4.3 Package: **Information Flow Control**

According to [9] information flows from sender $S$ through a channel to receiver $R$. A simple form of information flow is a message $M$, sent by $S$ to $R$. To be able to enforce security policies (e.g. to provide confidentiality), information flows between subsystems have to be observed by an information flow control mechanism.

Detecting information flows in "real life" is more complex: The decision, if a message sent from $S$ to $R$ leads to an information flow depends on the message and the knowledge of $R$. For example, if $R$ already knows the contents of $M$, no information flow has happended. On the other hand, the fact that $S$ has sent a message to $R$ can lead to an information flow. Also the fact that $S$ sends no message or to know that a subsystem is available or not can be an information flow. These kinds of information flows are called covert information flows. As you can see controlling flows of information is a very complex matter and different approaches exist how control rules can be enforced.

To avoid using a monitor which detects real information flow it would be possible to consider only theoretical flows. That means that the policy assumes that an information flow has taken place if it could. This approach can be enforced by access control policies if all possible flows are considered when permissions are granted to the subsystem. For example, to guarantee assurance levels of services or applications, information flows from highly trusted components to components with a lower trust level can be prevented if write access to subsystems with a lower trust level is not allowed. To be able to do that, the installation process can deduce assurance or trust levels and assign them to every component (this is discussed in Section 3.4.5). If assurance levels are assigned to every subsystem, the information flow mentioned above can be controlled by enforcing e.g. a security policy according to BELL and LAPADULA [7].

A more flexible (and more complex) approach could analyse the source code of subsystems (e.g. according to DENNING [14]) to detect possible data flows within the subsystem to consider them by the derivation service while permissions are granted. The results of the sourcecode analysis can be made public, e.g. by trusted parties (see Section 4.9.3.1).

The most complex approach would be to provide an information flow monitor which checks every message sent between subsystems and aborts them if it detects a forbidden information flow. The monitor could also consider evaluation results of information flow analysis of subsystems.

Information flow control is a very interesting topic and should be considered by the design of security mechanisms, but I decided not to consider information flows within the first prototype. Thus the package InformationFlowControl is left empty.

### 3.4.4 Package: **ResourceManagement**

This package contains interface definitions of services to manage unshareable resources as demanded by Section 3.1 and Subsection 3.3.3. Figure 3.8 shows dependencies between different types of resource managers.

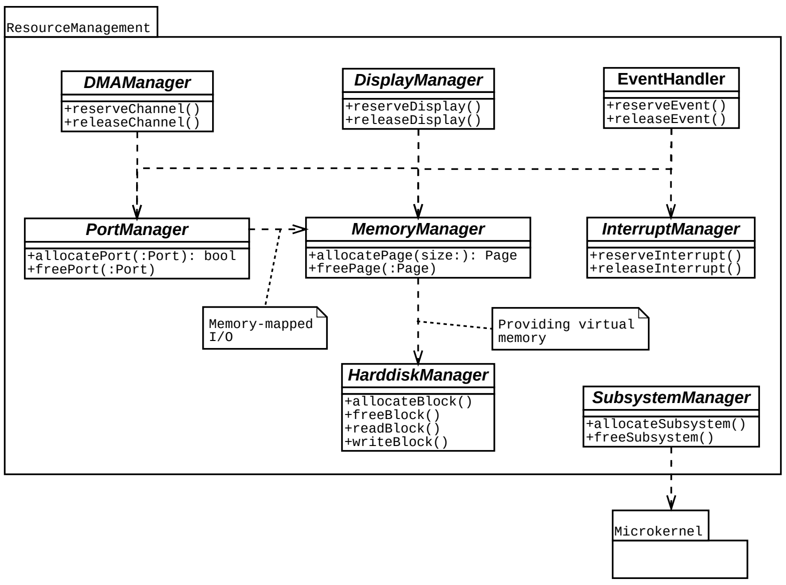

Figure 3.8: Components and their dependencies of the ResourceManagement package.

### 3.4.5 Package: **SubsystemManagement**

If users want to *update/delete* subsystems they have to be sure not to decrease the security of the secure environment. Also the *installation* of new subsystems is a very security-related procedure, because installation of malicious software can produce security holes. The following facts have to be considered:

1. Subsystems need a way to access subsystems which provide required services (Section 3.4.5.1).

2. The subsystem's code has to be downloaded (Section 3.4.5.2).
3. The permissions/capabilities of the new subsystem have to be derived (Section 3.4.5.3).
4. When updating/removing subsystems, dependencies between subsystems have to be considered (Section 3.4.5.4).
5. To be able to start the new subsystem the linker format has to be interpreted correctly (see Section 3.4.5.5).

The next subsection contain discussions of these topics.

#### 3.4.5.1 Service Definitions

As defined in Section 3.4.2 ACEFs have to provide only local valid references to encapsulated subsystems [SR 405]. The OOA model assumes that all subsystems use local references which have been assigned to exactly one service-version and which are mapped by a function [F 75], called naming service, into global references of the system environment.

#### 3.4.5.2 Loading New Content

To ensure that the security of the system is not reduced the secure environment has to provide integrity of the downloaded content. To install new subsystems, a system has to move data into memory. In most cases the subsystem's data is copied by a system function from a filesystem into a reserved memory area, but this is not a must. It is also possible to get the data via FTP, NFS, or HTTP. Thus, in general the data has to be copied into the memory via an (undefined) protocol. As explained above the interface provided to download contents should be independent of the data transfer protocol to make it easy to add new protocols. It is common to select the protocol implicitly or explicitly by the destination address of the content, e.g. "`http://www.semper.org`", or "`ftp://ftp.linux.org`". The interface should consider this. The class Loader contains all related operations.

#### 3.4.5.3 Permissions of New Subsystems

Before new subsystems can be started their permissions are derived by a DerivationService. Rules which define permissions of new subsystems are user and security policy dependent, thus the implementation should be policy independent. Users have to define rules wether and how components of different trust levels are allowed to interact. It is the access control's job to enforce these rules.

#### 3.4.5.4 Dependencies Between Subsystems

To make sure new content can be installed, the installation process should be able to check if dependencies between subsystems are fulfilled, e.g., if all required services (and compatible versions) are available. Also it should be possible to install different versions of the same service in parallel (if two subsystems depend on different versions of another subsystem, see *Use Case* 2.2.4). It should only be possible to replace an existing subsystem by another one if it provides the same functionality (e.g. a higher/lower patchlevel or higher minor interface number). Removing of subsystems should only be allowed if no other component depends on them (provide reliability). Dependencies are stored by the DependencyDB [F50] which is updated by the SubsystemManager.

#### 3.4.5.5 Starting a new subsystem

To start a new subsystem the secure environment has to execute the following list of actions:

- Check if the loaded data is an executable subsystem for this system and processor. This information is provided by a data-header created by the linker of the binary. Widespread formats used by many Unixes are the old a.out, the newer ELF or the multiboot-header-format used by FIASCO and the GRUB bootloader. To be able to use existing development tools the system should know at least one of those formats.
- If the data is recognized to be executable, different sections of the binary, e.g. the code-, data- and stack section, have to be identified.
- A new address space has to be created and the system has to copy the different sections into the new address space depending on some general rules (e.g. the code-segment is only accessed read-only, the stack and data segment read-write).
- Some special addresses of the code-segment have to be identified. For example every C program is started by the `main()` function, but if, e.g., C++ is used, static data has to be initialized before.
- The last step is to create a new thread, to initialize its registers with references to the code, stack and data segment and to execute it.

The class Installer provides all related operations to start a new subsystem. The class SubsystemManager provides an abstract interface to install/update subsystems. This class also provides the interface SubsystemManager of the ResourceManagement package, discussed in Section 3.8, to manage/grant rights to start new subsystems.

#### 3.4.5.6 Conclusion

Figure 3.9 summarizes all classes of the SubsystemManagement package.

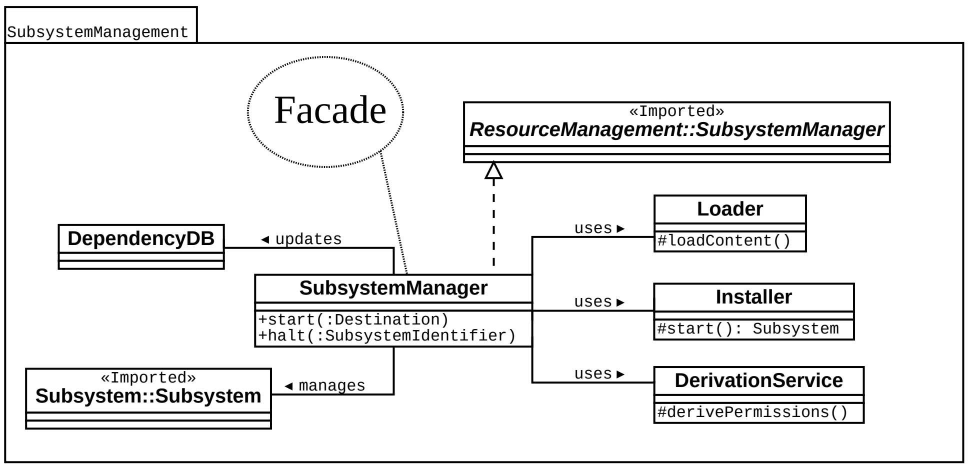

Figure 3.9: Contents of the SubsystemManagement package.

### 3.4.6 Package: **Crypto**

This package contains cryptography-related classes required by *Use Case* “Signing an Email” and “Generation of a new Key Pair”. The implementation of the class Converter depends on provided crypto standards.

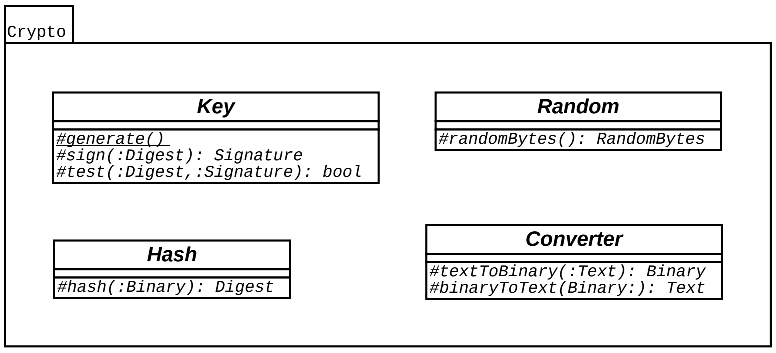
Crypto
Key
#generate()
#sign(:Digest): Signature
#test(:Digest,:Signature): bool
Random
#randomBytes(): RandomBytes
Hash
#hash(:Binary): Digest
Converter
#textToBinary(:Text): Binary
#binaryToText(Binary:): Text

Figure 3.10: Components of the Crypto package.

### 3.4.7 Package: **KeyManagement**

Of course secret key data have to be stored by the secure environment and must not be accessible by the *Client OS* except through well-defined interfaces (subsystems). But also public keys have to be stored to be able to verify a path of certificates. They have to provide a crypto-standard independent mechanism to identify owners and to verify authenticity. [47] contains a detailed discussion of this topic.

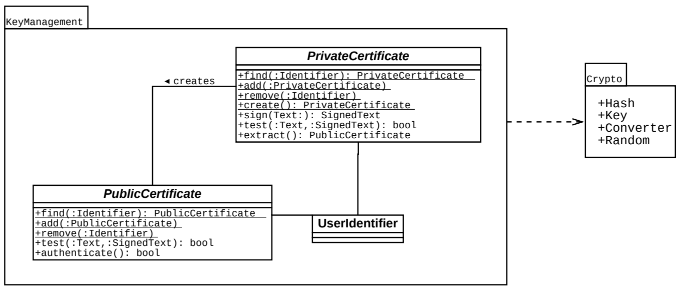
KeyManagement
PrivateCertificate
+find(:Identifier): PrivateCertificate
+add(:PrivateCertificate)
+remove(:Identifier)
+create(): PrivateCertificate
+sign(Text:): SignedText
+test(:Text,:SignedText): bool
+extract(): PublicCertificate
◂ creates
PublicCertificate
+find(:Identifier): PublicCertificate
+add(:PublicCertificate)
+remove(:Identifier)
+test(:Text,:SignedText): bool
+authenticate(): bool
UserIdentifier
Crypto
+Hash
+Key
+Converter
+Random

Figure 3.11: Components of the KeyManagement package.

### 3.4.8 Package: **TrustedPath**

To be able to provide a trusted path as demanded by [SR 255-357], all software layers between subsystem and output device have to be trusted or cryptography has to be used to provide confidentiality and integrity. These topics are discussed in Section 3.4.8.2 and Section 3.4.8.3.

Additionally both ends of the trusted path, the subsystem and the user, have to authenticate each other. Authentication of the subsystem is discussed in Section 3.4.8.1, user authentication is examined in Section 3.4.8.4.

#### 3.4.8.1 Subsystem Authentication

It is important that users know exactly which application they are communicating with, to prevent malicious subsystems from pretending to be another one. This information *always* has to be provided by the secure environment outside control of the subsystem itself. Three different approaches can fulfill this requirement:

1. Additional hardware (LED or an additional display) can be used to provide this information to the user.
2. The second approach uses a reserved display region to display this information. This requires the secure environment to guarantee that no other subsystem is able to access this region.
3. The third approach is used e.g. by Java which adds a reserved yellow region to every applet window, to prevent that malicious applets pretend to be a system window. Certified applets do not have this warning region.

Using extra hardware to show the trust level makes it possible for existing applications to use the whole display as they are used to, but requires to modify the hardware. Because this is not possible for me, this solution is not examined. Using a reserved region of the display to show the trust level to the user restricts the size of the display, therefore some applications may not work without modifications[6]. Another disadvantage is that only the security level of one application can be shown at the same time.

The usefulness of the second and third approach heavily depends on the size and general use of the display. If the display is small and usually used by only one fixed-sized application (PDA), both solutions are equal and the second solution seems to be simpler in this case. Some design decisions for small displays are discussed in Section 4.5.4.1.

If, in contrast, the display size is large and more than one application is visible at the same time (PC), the third solution should be preferred. To provide a trusted path at least the area which contains the reserved region has to be drawn by trusted components which have to be protected from malicious attacks by a separated protected domain. Section 4.5.4.2 contains a discussion of this topic.

#### 3.4.8.2 Package: TrustedGUI

To be able to provide a trusted communication path to local users as demanded by [SR 355-357] the secure environment has to access the display device itself to prevent modifications made by the *Client OS*. In order to provide confidentiality and integrity for information displayed by GUI widgets secure applications should use a trustful GUI toolkit implementation [F 100] provided by the secure environment. To keep the probability of errors small and make verification easier, only a small subset of GUI classes should be provided by the trusted GUI. Table 3.1 lists all GUI widgets the trusted GUI has to provide. Of course it is possible for each subsystem

| | |
|---|---|
| window | = virtual display |
| message box | (ok, yes/no, yes/no/cancel) |
| push button | to make yes/no/cancel buttons |
| combo box | for 1 of n selections |
| line edit field | type names, IDs, passwords |
| label | to display static text or graphic |
| scrollbar | to show large documents |

Table 3.1: List of widgets the trusted GUI has to provide.

to provide its own GUI implementation, but this is very complex, error prone and

[6]This is not valid e.g. for X, because it can be configured to have any size.

makes it necessary to check and evaluate the whole GUI implementation to certify a subsystem. Another advantage of separation between GUI toolkit and subsystem is that users can decide themselves which GUI implementation they want to use. Three approaches to separate GUI interfaces and implementations which depend on the functionalities provided by the underlying system exist:

***Static* Libraries.** If the system provides only static libraries users have to compile subsystems themselves to link them with their own GUI implementation or they have to trust the content provider to use the correct library. Alternatively, developers who want to distribute their software only in binary format could provide the object-file which users have to link against other libraries themselves.

***Shared Libraries*.** Using another GUI implementation is much easier if the system provides shared libraries, because this makes it possible to exchange the library-implementation at any time.

***Separated address spaces*.** It could be a disadvantage of the two above-mentioned approaches that the GUI library and the subsystem share the same address space (e.g. malicious subsystems can access also private and protected C++ data). If the GUI toolkit is implemented using its own address space (service), this can be prevented.

#### 3.4.8.3 Trusted Document Viewer

As demanded by Section 3.2 a document viewer is required to make it possible for users to verify documents [F 105]. The data-format used by the viewer should be system-independent (e.g. HTML or XML) and publicly available. The viewer has to use a trusted GUI toolkit to provide confidentiality and integrity.

#### 3.4.8.4 Package: UserAuthentication

This package contains classes which relate to user authentication. At least two authentication mechanisms, PasswordBased and SmardcardBased, should be provided by the system.

#### 3.4.8.5 Conclusion

Classes required to provide a trusted path between subsystem and user are contained in the TrustedPath package. Components of the trusted GUI got their own subpackage TrustedGUI and components of the user authentication are separated into the package UserAuthentication.

### 3.4.9 Package: CommunicationPath

To prevent that every *Client OS* application which uses components of the secure environment has to be modified, a communication path [F 3] between *Client OS* and secure environment should exist. It should be as flexible as possible to fulfill as many demands of different scenarios as possible and an existing protocol should be selected to make it easier to adapt existing *Client OS* applications. The following protocols are well-known and often used by Unix systems:

1. A network connection (TCP/IP).

2. Remote Procedure Call (RPC).

3. CORBA.

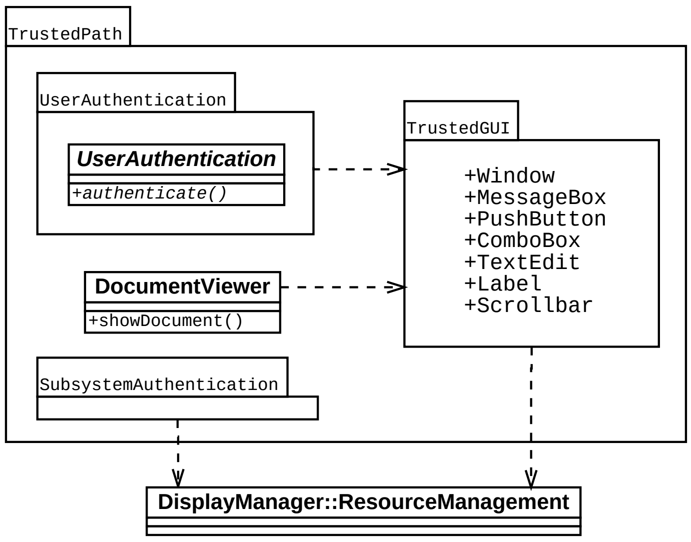

Figure 3.12: Components and subpackages of the TrustedPath package.

Because both protocols, RPC and CORBA, use TCP/IP as transport layer and a lot of services (smtp, pop3, *XWindows* etc.) also use TCP/IP it should be considered if it is possible to establish a virtual network connection which internally uses $\mu$-kernel IPC as communication path between secure environment and *Client OS*. This makes it possible to develop higher protocols later. Another advantage of this approach is that nearly all modern operating systems have an abstract interface supporting TCP/IP which hides the data transport layer (ethernet, serial line, IrdDA etc.). This simplifies creation of a communication path because only a new driver implementing the TCP/IP interface has to be written and applications do not have to be modified. More advantages are mentioned in the next paragraph. Alternatively a new protocol or interface (e.g. provided by a library) based on $\mu$-kernel IPC can be used. This should be faster, but requires modification of every application accessing the secure environment. The advantage of the TCP/IP approach can be illustrated using the *Use Case* "Signing an Email":

> If the secure environment provides SMTP/POP3 services, a mail client of the *Client OS* can easily be adapted by configuring it in such a way that it sends all outgoing messages to the SE's service and receives all incoming messages from the SE's POP3 server. If the *Client OS* is Unix-like, adapting is even simpler, because only the sendmail service of the *Client OS* has to be reconfigured to send/receive to/from the secure environment. If the SE controls the network device (Figure 3.13 (b)) it can enforce using its services by deleting (filter) all messages sent to other services.

Two different approaches are conceivable, using TCP/IP as communication path, outlined by Figure 3.13:

1. The *Client OS* keeps control over the network adapter and can establish a point-to-point connection to the *secure environment* (a).

2. Control over the network adapter is moved to the secure environment which acts as a default gateway to the *Client OS* (b) which forwards all network traffic to the *secure environment*.

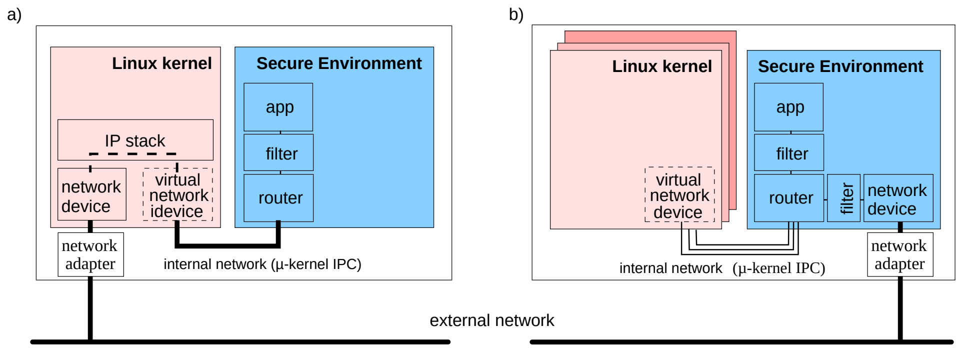

Figure 3.13: Internal communication path if a) Linux and b) the *secure environment* controls the network device.

As you can see the second solution has the advantage that more than one *Client OS* could be used if every instance gets its own IP address. Further the secure environment can act as a firewall to the *Client OS* instances, because it can control all network traffic from/to the *Client OS*'s. To avoid implementing a network device driver for the secure environment I prefer (for this prototype) the first solution, although the second seems to be the more secure (and flexible) one.

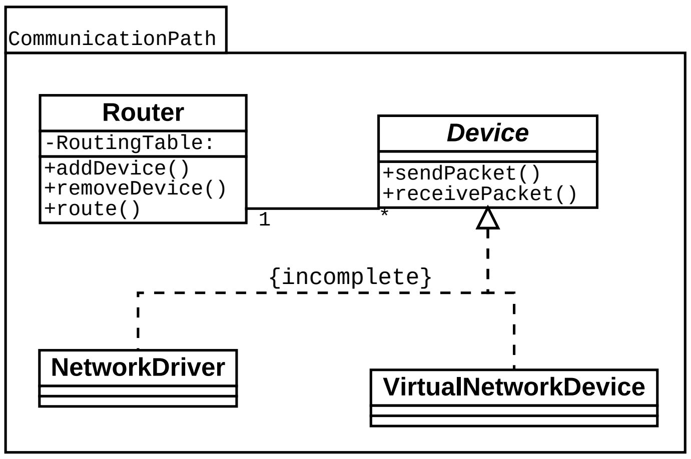

Figure 3.14: Classes of the CommunicationPath package between subsystems.

The *UML* diagram 3.14 describes required classes and interfaces of the package CommunicationPath. The Router is a service which routes packages received from devices (IP numbers) which can register or unregister themselves. The VirtualNetworkDevice driver of the *Client OS* and every subsystem which represents a network device (e.g. a trusted network driver) has to implement the Device interface. Packet filters shown by Figure 3.13 can be implemented in three different ways:

1. Directly implemented by the Router subsystem.
2. Extend the Router interface by methods to insert external filters.
3. Using a Device instance which acts as a default gateway, receives all network packets, filters them and sends them back.

In order to avoid implementing a new TCP/IP packet filter (and if demanded security-levels are not too high) it would be possible to use the filter and firewall mechanisms provided by another minimalized LINUX kernel, protected by the secure environment.

## 3.5 OOA model

Figure 3.15 outlines layers and their components of the current system model. Below the red line we have the *hardware* and the *$\mu$-kernel* which hides hardware-dependent concepts. Based on the $\mu$-kernel (between red and yellow line) the *secure platform* provides security-related core components which provide separation and protection of subsystems. These are an *access control mechanism*, *resource manager* and *subsystem management functions*. The *application framework* above the yellow line contains services which can be accessed by the application layer (above the green line). Dependencies between services of the application framework are not repre-

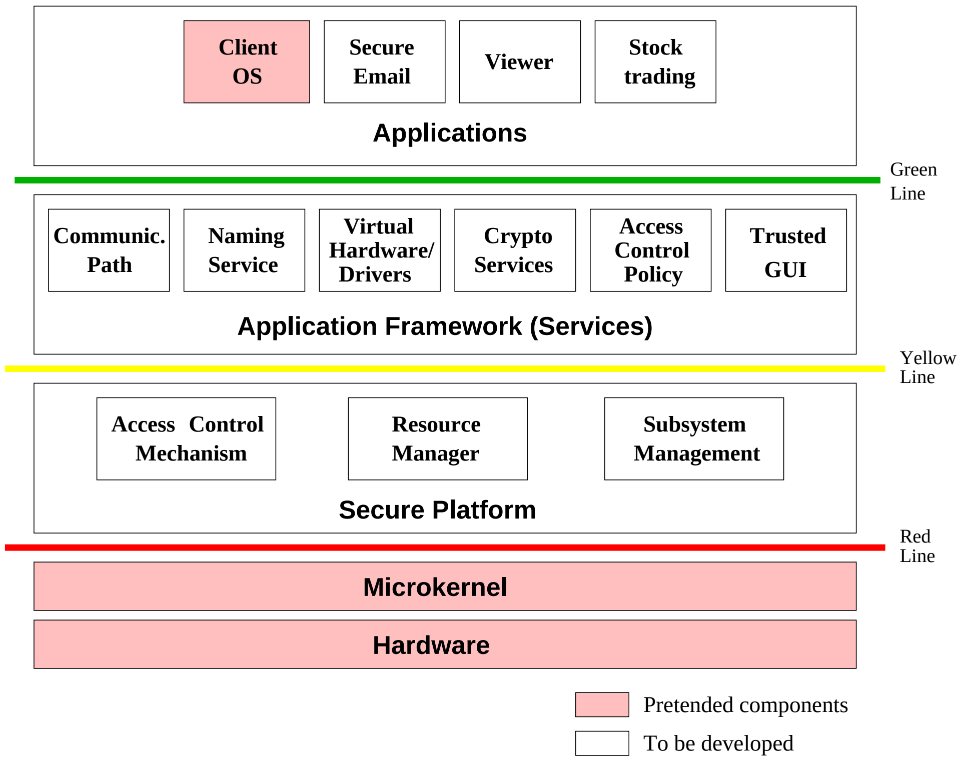

Figure 3.15: System model after analysis phase.

sented by this figure, but they are outlined by the UML diagram (Figure 3.16) which contains all components mentioned in the analysis phase.

The main difference between this model and the model suggested by the PERSEUS proposal is that the *secure platform* only contains elementary components and that the *Client OS* is only one application besides others, which is able to access services of the *application framework*. This is possible because the assurance level of applications is defined by the access control policy and enforced by the access control mechanism.

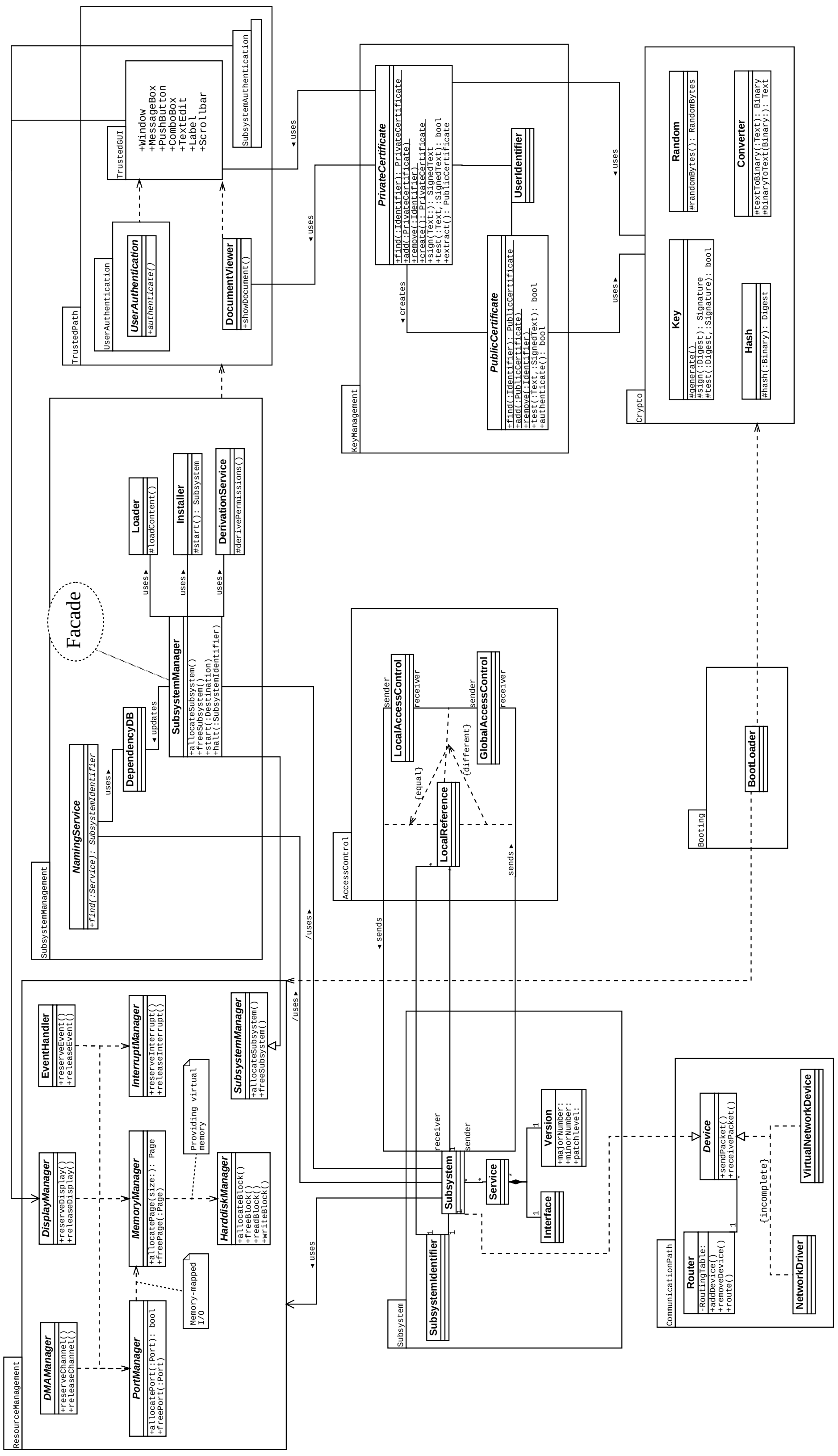

Figure 3.16: The OOA model.

# Chapter 4

# Design

The preliminary PERSEUS architecture [21] suggests to use FIASCO as microkernel and L4-LINUX as *Client OS*. I decided to use them, too.

## 4.1 General Assumptions on the Underlying Components

This section describes assumptions and demands on components below the red line of Figure 3.15 to be able to fulfill some general security requirements of Chapter 3.

I assume that a mechanism is provided which protects different subsystems from each other. Such kind of protection is required by [SR 295], [SR 296] and [SR 297] to protect subsystems, especially the access control components, from interference and tampering by untrusted subsystems. Separated address spaces provided by the hardware is a usual way to fulfill this requirement.

Additionally it is assumed that the microkernel provides an internal communication channel between subsystems that provides confidentiality and integrity to protect internally transferred data as demanded by [SR 355] and [SR 260].

To be able to build a monitoring component that is "always invoked" as described in the *Common Criteria*, subsystems must not be able to initiate a communication to other subsystems without recognition by the kernel.

## 4.2 The Fiasco Microkernel

This section gives a short overview of the FIASCO microkernel. It contains excerpts from [19], [27] and [25].

The **FIASCO** microkernel is a new implementation of the L4 microkernel interface for the x86 architecture. L4 is a microkernel interface defined by JOCHEN LIEDKE, and there exist implementations for the x86-, the MIPS- and the Alpha CPU. L4/x86 is implemented in highly-optimized assembly language and it is not freely redistributable. In contrast to the L4 microkernel, FIASCO is completely implemented in C++ and distributed under a freeware (open-source) license.

***Address Space.*** At the hardware level, an *address space* is a mapping which associates each virtual page with a physical page frame or marks it non-accessible. The microkernel hides the implementation of virtual memory pages by providing the concept of address spaces, which protect a subsystem's code and data. The basic idea is to support recursive construction of address spaces outside the kernel. Three operations, implemented by IPC, are provided by FIASCO:

1. The owner of an address space can *grant* any of its pages to another space if the recipient agrees. The granted page is removed from the granter's address space.

2. The owner of an address space can *map* any of its pages into another address space if the recipient agrees. The granted page can be accessed in both address spaces.

3. The owner of an address space can *flush* any of its pages. The flushed page remains in the flusher's address space, but is removed from all other address spaces which had received the page directly or indirectly by a map or grant operation.

***I/O.*** Although documented in [19, page 14] the FIASCO and the L4 microkernel do not have a concept to mediate I/O ports. In Section 4.5.3 a basic concept to securely control and mediate I/O port access is presented.

***Threads.*** A thread is the basic execution abstraction. A thread has an address space (shared with other threads), a unique thread-id (TID), a register set, a page fault handler (pager) and an exception handler. The pager and the exception handler are threads running within the same or another task. IPC operations are addressed to threads via their TIDs.

***Task.*** A task contains an address space and all threads belonging to the address space. A task itself is no active entity, only threads are able to act.

***Inter Process Communication (IPC).*** It is the only[1] communication mechanism between threads of different address spaces provided by the FIASCO microkernel. CPU aligned words and bytestrings can be copied by reference and by value. IPC is used by FIASCO for interrupt-handling, pagefault-handling, exception-handling, wakeup-calls and grant/map/flushing of pages. The independence between subsystems is guaranteed, because FIASCO's IPC mechanism enforces a certain agreement between sender and receiver and it guarantees confidentiality and integrity of the message.

***Clans and Chiefs.*** Clans and chiefs are a basic mechanism to implement arbitrary security policies. They allow controlling IPC and thus information flow. A task's creator is that task's chief, all tasks created by a chief are that chief's clan. Tasks created by a chief can create subtasks and subclans on their own. If a message is sent to a thread outside the clan it is delivered to the sender's chief and vice versa, if a task gets a message from outside the clan the message is sent to the receiver's chief. This is FIASCO's kind of message redirection.

***Resource Allocation.*** Each resource is allocated on a first-come-first-serve basis. Initial services have the chance to allocate resources and can then mediate them depending on their philosophy.

## 4.3 General Design Decisions

This section discusses some global design decisions applicable to nearly all subsystems.

[1]Also shared pages can be used as a communication channel. But because pages can only be shared using IPC, controlling IPC means controlling the existence of information channels between subsystems. The difference is that it is not possible to control (on demand) the data flow between subsystems which share memory pages. To ensure that no covered data flow exists it is necessary to evaluate the subsystem's source code (see Section 3.4.3) and to restrict permissions to share pages. To be able to revoke the permission to share pages (see FMT_REV) a trusted component has to manage all pages shared between untrusted subsystems (currently not possible).

### 4.3.1 Methods and Interfaces

The model proposed in Chapter 3 assumes that underlying components, which provide protected domains, support object-oriented concepts. FIASCO uses the tasks to protect subsystems against each other, but the concept of methods is not supported. Therefore I use the following mechanism (which is also used by the RMGR of the FIASCO package) to emulate it by the IPC mechanism of the FIASCO microkernel: The first inlined dword_t argument, hence called MessageID and provided by all IPC functions, is used to select the method to be invoked.

To control access to different interfaces which are implemented by one task, different threads of the same task provide these interfaces which can then be controlled separately. The *UML* notation is modified in such a way that the alternative symbol of interfaces, which uses a circle, defines those interfaces which have to be implemented by different threads. This modification is required because *UML* only uses the three visibilities `public`, `protected` and `private`. Figure 4.1 illustrates an example of a subsystem which uses two threads to provide three inherited interfaces.

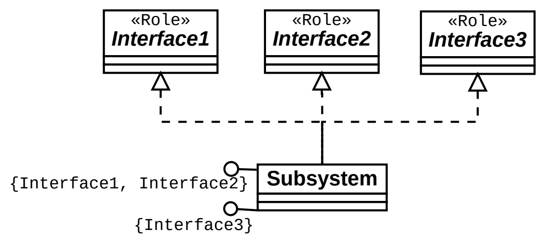

Figure 4.1: Illustration of the modified UML syntax: Two threads are used to provide three inherited interfaces.

The stereotype "Role" is used to remark that permission to access the interface is bound to a role. Developers have to consider that MessageIds of roles which may be implemented by one thread (Interface1 and Interface2 in this case) do not overlap.

### 4.3.2 Separate interfaces and protocol implementations

A lot of system services are required which shall be easily extendable by new protocols. To be able to install/delete protocols without global changes (to prevent re-evaluation), *services* have been separated into different subsystems which are managed by a subsystem which acts as a Proxy. Figure 4.2 illustrates the general design model: A ServiceManager (which acts as a proxy) implements the Service interface and additionally an interface Manager to install/delete new services. The ServiceManager registers with the *naming service* and forwards incoming requests to appropriate service implementations. Therefore the implementation of the ServiceManager instance can be kept small and simple. Because external subjects do not know how many protocols a service implements, the method `register()` invokes Service instances to register themselves to a given Manager implementation[2]. Another advantage of this approach is that protocol implementations of different providers can be used in parallel and cannot disturb each other. The Proxy pattern is also used by services to fulfill a requirement of Section 2.1, which demands that subsystems are able to refine or extend the default behaviour of services (see Figure 4.3): If the system provides a global Service (or ServiceManager) $x$, subsystems can

[2]This is an example scenario which makes it necessary to synchronise local references between two or more subsystems: To invoke a Service to register to a manager the Manager argument of the register() method has to be synchronised.

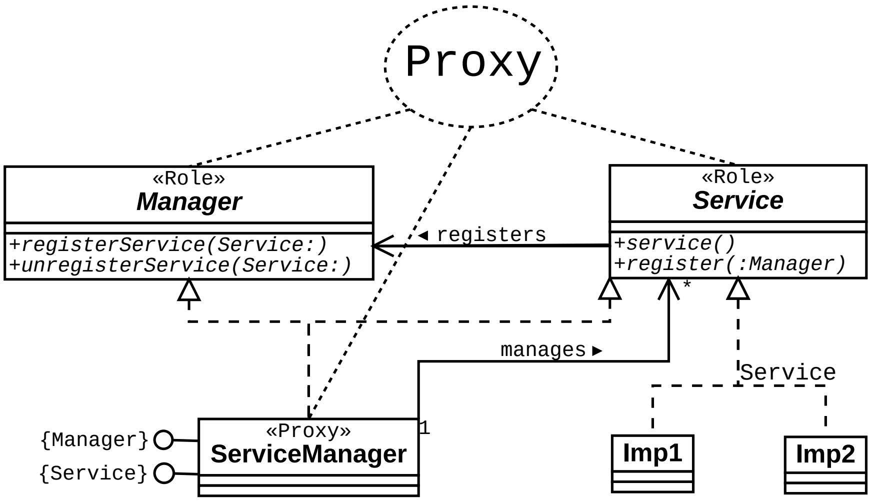

Figure 4.2: Design of separation of interface and protocol.

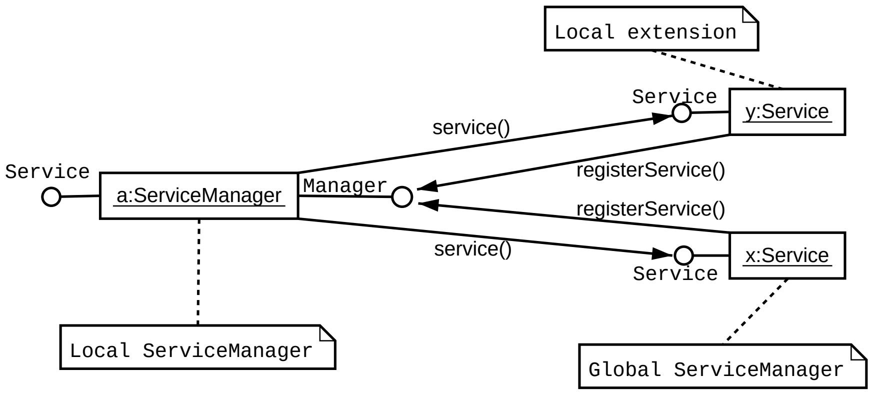

Figure 4.3: Dependencies between Manager and Services.

initiate their own SeviceManager $a$ and register further (or only other) services $y$. Two different implementations have to be distinguished if ServiceManagers themselves are registered to other ones:

1. If ServiceManager $a$ registers itself with another ServiceManager $b$ access control only checks if $a$ has permission to access ServiceManager methods of $b$.

2. If ServiceManager $a$ calls its registered services to register to another ServiceManager $b$, access control separately checks if these services have permissions to register to ServiceManager $b$.

The first solution should be preferred to ease changes and extensions, e.g. of global services. The Proxy pattern given by Figure 4.2 has to be refined by concrete implementations and the design of the Loader package (Section 4.10) is an example implementation.

## 4.4 Package: **Access Control**

To get a flexible, less error prone and policy-independent reference monitor, access control should be separated into policy-dependent and policy-independent components [9] [38]. According to [SR 298] the policy-independent part, the *Access Control Enforcement Facility* (ACEF), has to be *always invoked* [SR 60] whenever a subsystem accesses another subsystem. The policy-dependent component, the *Access Control Decision Facility* (ACDF), decides if the demanded access is allowed or not; it encapsulates the security policy. This approach keeps the ACEF policy-independent, prevents changes of the ACEF if the policy is changed and leaves all other subsystems access control independent. Because the ACEF provides only a small amount of functions it can be kept small and (hopefully) error-free. The ACDF contains policy-dependent rules which define access control policy. Because the ACDF can be accessed by the ACEF only by a fixed set of messages, it is possible to change the rules of the ACDF, or to exchange the ACDF on demand.

The general model has been discussed in Section 3.4.2, but because FIASCO does not provide a *message redirection mechanism*, its chief & clan concept is used to enforce the *always invoked* requirement:

A trusted ACEF implementation is used as a subsystem's chief which catches and forwards all upcoming messages. In order to synchronize incoming and outcoming messages the ACEF internally maps local references into global ones. The ACEF of the receiver maps global references into local ones of its encapsulated subsystem. Thus communication between ACEFs uses global references. Figure 4.4 outlines an abstract model which illustrates virtual and real data flows.

Thread-based IPC mechanisms influence the access control decision facility in that way that it has to decide if a given task has permission to access a demanded thread. This restriction is required, because information of threads which run within one address space cannot be protected from each other (except information flow analysis proves independence of threads, but this is currently ignored). Thus the access control policy has to be defined by an access control matrix $\mathrm{M}_{AC}$:

$$S \times R \times M_m \rightarrow \{true, false\}$$

with the set of all sending tasks $S$, the set of all receiving threads $R$ and the set $M_m$ of all methods accepted by the receiving thread $R$.

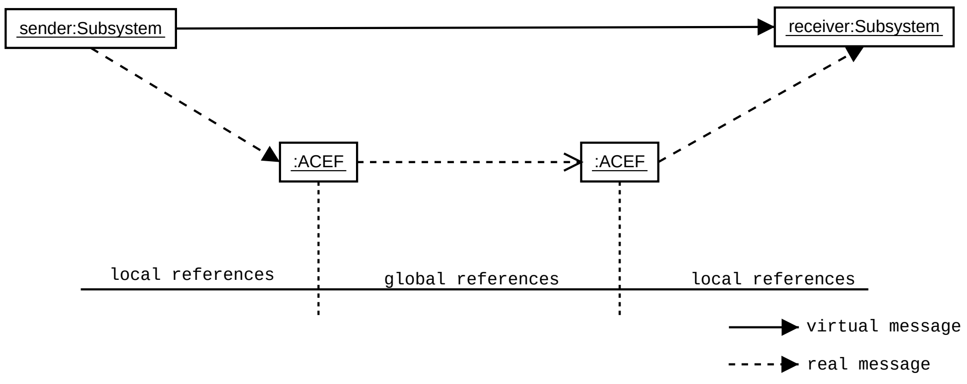

Figure 4.4: Real and virtual data flow between sender and receiver.

## 4.4.1 Global Security Policy

In order to make it possible to use more than one access control policy in parallel, the access control decision facility is designed using the Proxy pattern described in Section 4.3.2. The ACDFManager only returns true (access allowed) if all managed ACDFs (all policies) agree. To enforce global access control only one ACDF instance (a Singleton) can be used (to simplify changes of access control policies), or individual ACDFs can be assigned to ACEFs on installation. Figure 4.5 outlines ACDF-related classes.

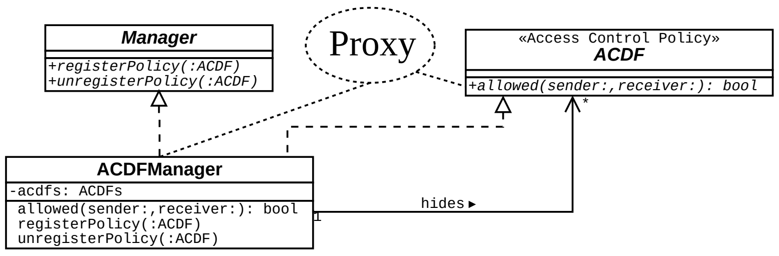

Figure 4.5: ACDF related classes of the AccessControl package.

## 4.4.2 Access Control Lists

Permitted subjects are able to assign an ACDF which acts as an ACL to the ACEF of a subsystem. Because an ACDFManager (see above) can be used, it is possible to consider more than one ACL in parallel. Users should be able to define a default ACL, assigned to every created ACEF by the SubsystemManagement package.

## 4.4.3 Reference Mapping

As described above the ACEF implementation has to map local references into global ones and vice versa, and it has to synchronize references of incoming and outgoing messages. A fixed mapping between services and local references as proposed by the OOA model (Section 3.4.5.1) has some important disadvantages:

1. It is difficult to synchronize unique references of all existing services. At least one party has to support a database containing all local reference to service mappings.

2. The datatype of the local reference has to be very large to prevent shortage of available local references. This produces more overhead and uses system resources.

3. Range-checking is not possible. Also efficient mapping of references using an array is not possible, because this requires a continuouse range of local references.

Thus a *naming service* which maps service names into global references is provided by an interface NamingService. The input parameter of the NamingService is a String type which describes the required service. The subsystem's view of the type of local references is ThreadId to make no differences between internal and external services; the ACEF converts them into an integer type to be able to uses them as indices of an array.

The NamingService interface is implemented by the ACEF which can return the global ThreadId if the service is available in the same clan (faster access). Else a new entry is added to the array of ThreadIds. Using the naming service does not decrease efficiency, because it has to be invoked only if the service is accessed the first time. The persistence of underlying components (see Section 4.5.2.1) stores mappings until the subsystem is removed.

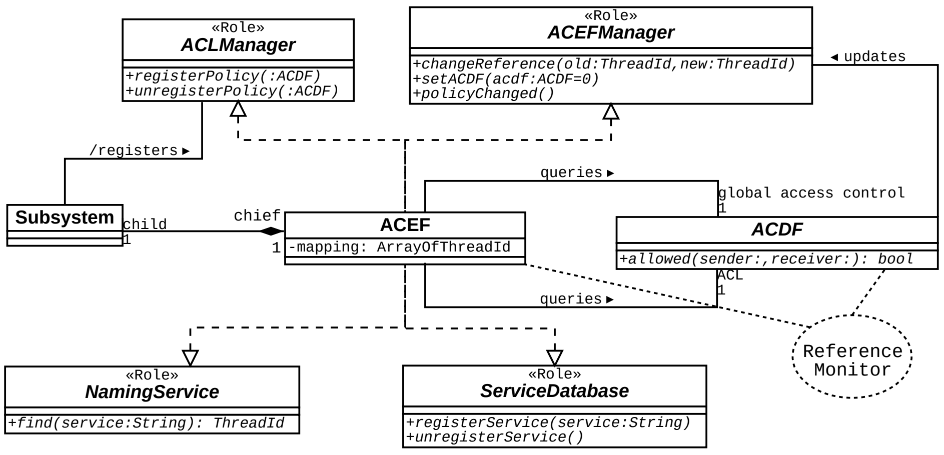

Figure 4.6: Design of the class ACEF.

The interface ACEFManager provides management methods to be used by trusted components, e.g. security management modules. If global references of services change (e.g. updated or restarted) the method `changeReference()` can be used to update the ACEF's internal mapping table. To revoke the capability to access another subsystem the new ThreadId can be made invalid. The method `setACDF()` replaces the ACDF used by the ACEF and the method `policyChanged()` has to clear the mapping table and enforce the ACEF to re-check permissions if the subsystem use them the next time. It has to be considered that also shared pages have to be flushed to be able to re-check the permission to share pages. Therefore some management functions are required to store a list of shared pages. For example the subsystem's ACEF could internally update a list of pages of its subsystem's shared pages and flush them if the method `policyChanged()` is invoked.

The interface ACLManager can be used to set/change/remove (locally) access control lists. It is available to the encapsulated subsystem to enable enforcement of local policies by the subsystem itself. The interface and behaviour of ACLManager and ACDFManager are very similar, therefore the same interfaces have been used.

The interface ServiceDatabase provides methods to register/unregister new services. It is the security policy's task to decide if it is registered locally or system-wide.

### 4.4.4 Global Names

As demanded by [F 115] subsystems sometimes have to synchronize their local references (an example is given in Section 4.3.2). Because the ACEF does not know the semantics of arguments of messages, local references cannot be remapped automatically. Different approaches are possible:

1. Using global names which are independent of global references. Permitted subsystems can use a GlobalNameService to map a local reference into a global one which can be remapped into a valid local reference by the receiver.

2. A reserved parameter is used to invoke ACEFs to remap reference arguments. Large overhead.

3. Global names are used as suggested by the first approach, but the sender receives the global name from the ACEF of the appropriate subsystem or the subsystem itself.

4. If this problem is only service-related the *naming service* can be used.

At the moment I cannot decide which solution would be the most flexible one, because every approach has its advantages and disadvantages.

### 4.4.5 Increase Granularity of Access Control

The access control mechanism proposed so far provides thread-based granularity, because Fiasco's IPC mechanism is thread-based. Currently L4 tasks can execute "only" 128 threads in parallel, therefore access control with a granularity of 128 separate sets of messages is possible within every address space. To be able to enforce different access control policies, developers of subsystems have to separate their messages into meaningful sets, implemented by different threads. Section 4.7 provides another approach which is more powerful but also slower. In order to control also arguments of messages (e.g. to enforce information flow control or to increase the granularity of access control), two different approaches could be used:

1. To enforce access control policies which also consider message arguments, proxy-tasks can be used: The *naming service* does not return the address of the service; instead it returns the address of a proxy-task which first enforces the policy and then forwards the message.

2. To enforce access control policies which also consider message arguments, the ACEF can be replaced by a new version which queries the ACDF every time a message occurs.

Currently the proposed solution (without control of arguments) should suffice.

### 4.4.6 Alternative Approach to Control Access

The approach discussed in this section can be used to increase the granularity of concepts suggested above or as an alternative model.

If services handle all messages by one thread, access can be restricted to a defined set of messages by using proxy threads. The advantage of this approach is that ADMINS can define the provided granularity themselves, but because another thread is involved whenever two subsystems communicate, it may be slower. Figure 4.7 illustrates this approach. All role proxies have to be trusted components and

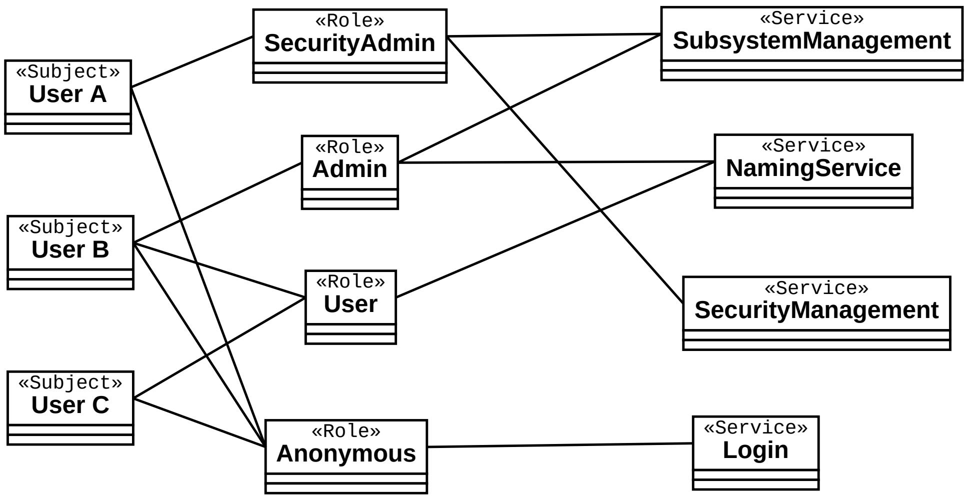

Figure 4.7: Alternative access control model using role proxies.

provide all messages assigned to the appropriate role. Because they are trusted they can access all services without restrictions. The ACEF has to enforce that the encapsulated subsystem can only access allowed roles. 128 roles should suffice within a usual sized system, thus all roles can be provided as threads of one subsystem.

### 4.4.7 Hierarchical Access Control

This section proposes an approach to use security domains which refine or redefine access control policies. It acts as an extension of the access control concepts explained in Section 4.4.1 to 4.4.3 and simplifies controlling sets of subsystems (necessary, e.g. to enforce that users cannot have two roles at the same time) and can reduce overhead produced by suggested concepts.

Instead of encapsulating exactly one subsystem, FIASCO's chief & clan concept allows to control all subsystems of one user by one ACEF. It acts as a root of further subsystems started by this user (similar to home directories used in many Unixes). If users start new subsystems, they are executed as children of the user's ACEF and therefore automatically controlled by it. Users are free to use the same policy and/or mechanisms within their own collection of subsystems, but if the global security allows it, they can enforce their own access control policy or use no local policy. To refine access from outside they can add ACLs to their ACEF; to control their own subsystems (e.g. debugging, logging) they can replace the default ACDF of their local subsystems by another one. Figure 4.8 illustrates the basic idea:

Two global services are controlled by their ACEF instances. Also the *Client OS*, which internally manages its own tasks and users and enforces its own access control policy, is controlled by an ACEF. User *A* protects and controls his/her local subsystems using separate ACEF instances, User *B* does not protect local subsystems at all. Figure 4.9 describes this scenario with respect to ACDFs and ACLs.

This design makes it possible to use individual ACDFs for every user, because only one ACDF has to be defined to control all subsystems of one user. Also changes of user-dependent permissions get easier: To deny access to the display or printer, only the user's root subsystem's ACDF has to be changed.

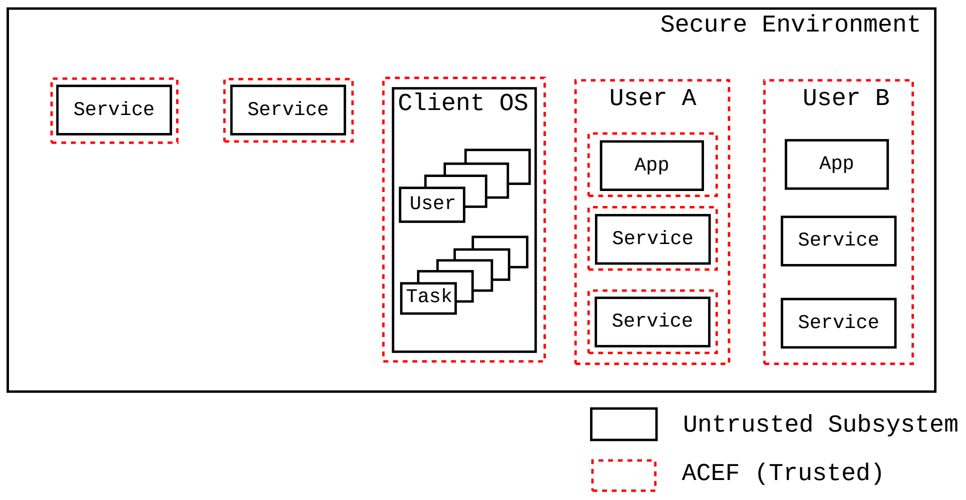

Figure 4.8: Hierarchical access control using clans & chiefs.

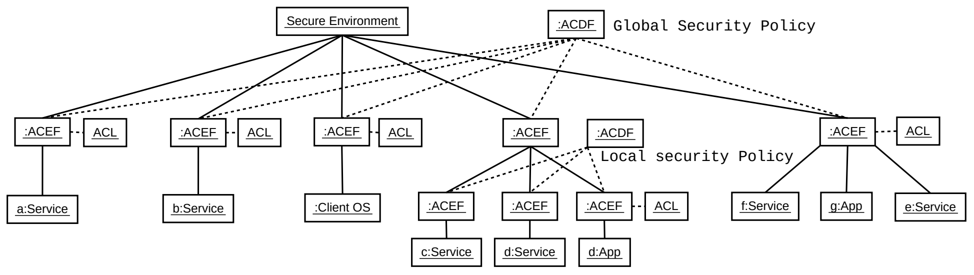

Figure 4.9: Task hierarchy of the scenario of Figure 4.8.

Generally speaking, the user's root ACEF created by the ADMIN provides its encapsulated subsystems a virtual view of the whole system, because locally installed subsystems cannot distinguish if they are installed globally or locally. So does the main ACEF, which provides globally installed services a virtual system view, dependent on their permissions. In my opinion this extention of the access control mechanism provides a mechanism to unify single user (PDA) and multi user system (PC). A single user system uses only one main ACEF and multi user systems use sub-ACEFs to separate users. Because ACEFs provide a virtual one-user-system view, subsystems can be used within both system types.

If subsystems need fast access to services (e.g. the naming service or a global database), it is possible to install trusted proxy services within the same clan which themselves query the global service to cache results. (see Figure 4.10).

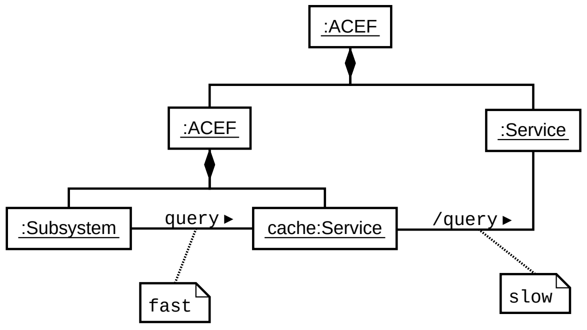

Figure 4.10: A caching service increases performance.

## 4.5 Package: **ResourceManagement**

This section contains refinements of the ResourceManagement package presented in Section 3.4.4. The first subsection explains a basic idea how a virtual hardware layer could be provided. Because implementation would use a lot of time, this approach is not further considered by this work. The other subsections discuss the design of subpackages of the "separated drivers" approach (Figure 3.4).

### 4.5.1 How to Provide a Virtual Hardware Layer?

One approach could use FIASCO's exception handler concept:

> Access to unsharable hardware resources (e.g. I/O ports) is globally disabled and trusted exception handlers are defined which are notified by a processor trap signal if the *Client OS* tries to access the hardware. If access is permitted (e.g. if no other subsystem has reserved this port) the exception handler can carry out the command which raised the exception and then return the control back to the *Client OS*. In all other cases the exception handler can do a defined action (e.g., try to allocate the port or abort the task).

This approach should be practicable because FIASCO allows to assign an exception handler to every subsystem and the exception message contains the address of the instruction which raises the exception. A detailed discussion of approaches how virtual hardware could be provided can be found at the homepage of the freemware project[3].

[3]http://www.freemware.org

### 4.5.2 Package: **MemoryPager**

MemoryPagers are accessed by subsystems implicitly if they produce page faults or explicitly if they demand new memory pages. The MemoryPager replies by granting a new page into the address space of the invoking subsystem or by an error message (e.g. if no more memory is available). To fulfill [SR_100] all MemoryPagers have to clear pages (using random numbers or NULL bytes) before they are granted to a new subsystem.

The general way to provide more memory than physically available is to swap out some pages to another storage (disk). As mentioned in Section 3.3.4 subsystems have different demands on allocated pagers, listed below:

- providing integrity
- providing confidentiality
- providing integrity and confidentiality
- providing fast access

I assume that dynamic or static memory is secure, thus security-demands to memory pagers only have to be considered if pages are swapped to another storage media. Therefore all these requirements can be fulfilled if pages are not swapped out, but this is not possible in all cases, e.g. if the system is shut down (see Section 4.5.2.1) or if more pages are required than available. On the other side, MemoryPagers could be used which themselves use secure media like protected disks or memory cards. Therefore the way how these requirements are fulfilled is an implementation detail. This section suggests a set of five types of pagers which support integrity, confidentiality and fast (locked in memory) pages (Figure 4.11).

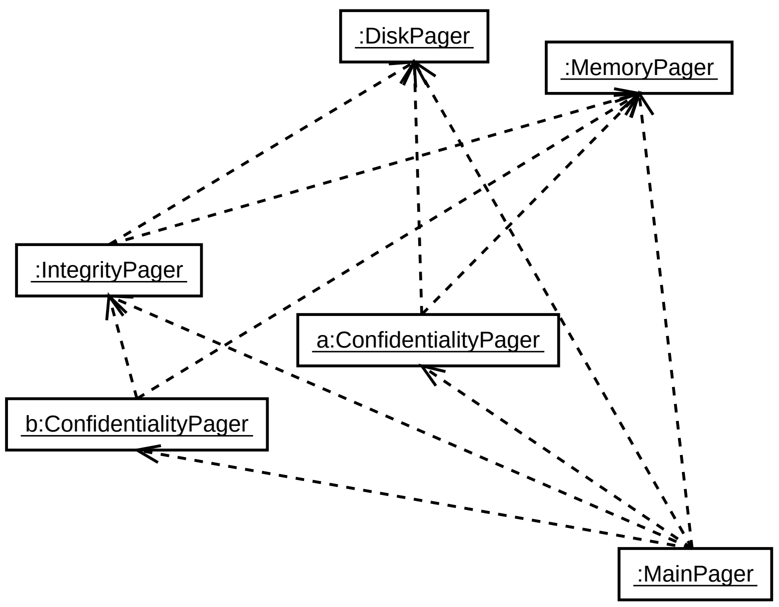

Figure 4.11: Dependencies between different pagers.

The first pager is the MainPager which supports all types of pages and provides a general interface to other subsystems. In general the MainPager uses the MemoryPager which handles memory pages. If not enough memory pages are available any more, the MainPager can swap out some page: To swap out pages without attributes the DiskPager can be used which stores pages on a persistent storage. Pages providing integrity are stored using the IntegrityPager, which hashes pages before sending them to the DiskPager and uses memory pages to store the hash-values. If the page is reloaded the hash-value is verified. To swap out pages providing confidentiality the MainPager uses the ConfidentialityPager *a* which encrypts pages before they are sent to the DiskPager. Pages providing integrity and confidentiality are sent to another ConfidentialityPager *b* which itself sends the encrypted page to an IntegrityPager. The ConfidentialityPagers use (locked) memory pages to store the symmetric key used for encryption, the pager providing integrity can use them, e.g. to store hash values.

It is important that on startup and shutdown memory pages internally used by IntegrityPager and ConfidentialityPager are stored securely by using operations which provide integrity and confidentiality. The master key given by the ADMIN on startup (see Section 3.3.4) can then be used to encrypt the last page which contains the encryption key. Also the behaviour of all pagers has to be adapted to the persistence model. This topics are discussed in the next two subsections.

#### 4.5.2.1 A Persistent System behaviour

This section presents a mechanism which provides global persistence and allows frequent backups of dynamic memory. It is based on ideas of [26], [15], [17] and [18].

One principle of this model is that the persistent storage (e.g. a harddisk) is divided up into persistent pages (ppages) and that the processor directly acts on these ppages. To understand the concept, *dynamic memory* can be ignored or considered as cache of the ppages. If it is possible to enforce that the manipulation of ppages is an *atomic operation*, we get a persistent system which can be switched off at any time (at this point of abstraction restoring of processor and device states is ignored).

A model of persistence which uses *dynamic memory* as cache has to ensure that

1. modified memory pages (mpages) are frequently written back to the persistent storage to keep loss of data small if an error occurs.

2. the consistency of the data is not destroyed.

A simple solution to fulfill the second requirement would be to make storing of all mpages to ppages an *atomic operation*, as it is done e.g. by BIOS functions of some notebooks which store all memory pages to a reserved disk partition (suspend-to-disk). Because it takes a long time to store for example 128 MB to a disk it is unreasonable to do this frequently as demanded by the first requirement, also interrupting the system is not possible, because users will not accept this behaviour and some devices like serial ports and ethernet cards have to be read frequently to prevent buffer overflows.

As a first step to decrease time of system inactivity, the atomic backup operation can copy the content of modified mpages to other (free) mpages. Copying these mpages to ppages can then be done by a background process. If all modified pages have been stored the next backup can be made. Thus the frequency of backups only depends on the priority of the backup process and the speed of the persistent storage.

But also copying 128 MB of RAM may interrupt the system for a (too) long time interval. The atomic mpage copy operation can be optimized by using virtual pages and a page-wise copy-on-write mechanism: At the beginning of the backup procedure the mpage is copied by creating a new virtual page which points to the same mpage which has to be marked read-only. Only if the mpage is modified by a write access, which occurs slower and one after the other, it is physically copied to

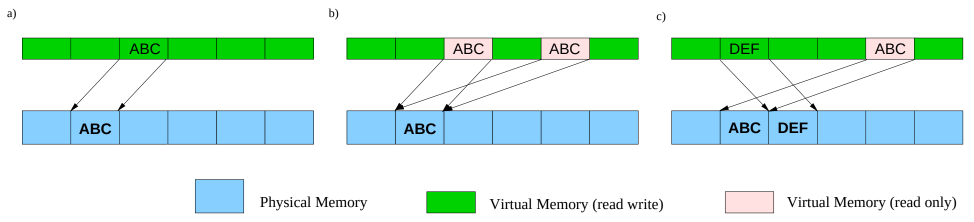

Figure 4.12: Address space (a) before and (b) after copy-on-write, and (c) after overwriting the original mpage.

another mpage. Figure 4.12 illustrates the behaviour of a copy-on-write mechanism. Copying a virtual page is identical to copying a reference to the mpage, thus the atomic part of the backup operation only has to copy references of all mpages. This approach can be optimized if the backup operation copies only references of mpages which have been modified since the last backup operation. Most processors have a pagetable indicator which is automatically updated if a page is modified.

To protect the system consistency against errors while writing the mpages to the persistent storage two regions of ppages should be used which are alternately written. Figure 4.13 outlines the complete set of required virtual, physical and persistent pages.

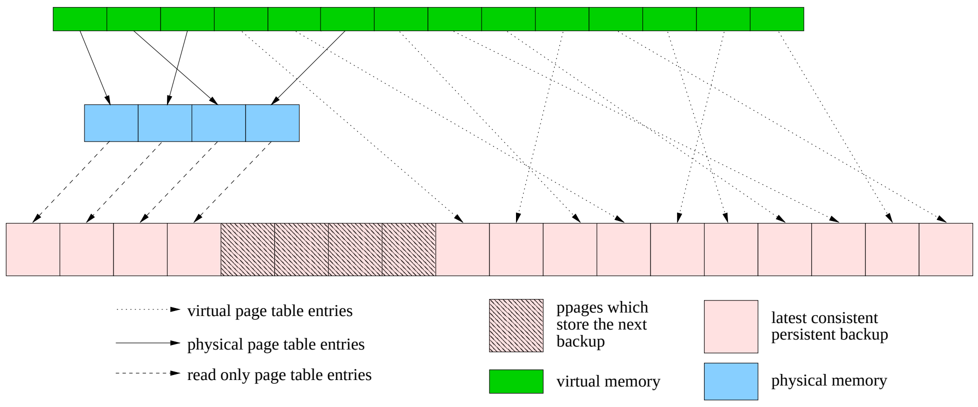
virtual page table entries
physical page table entries
read only page table entries
ppages which store the next backup
virtual memory
latest consistent persistent backup
physical memory

Figure 4.13: Required mpages and ppages to build a persistent system.

In order to keep the system state consistent it is important to store the pagetable of a consistent state together with its pages. A stored pagetable is valid until the next pagetable, which contains the next consistent state, has been stored completely and marked valid. To be able to reload the latest consistent state a protocol has to be used which guarantees that the latest valid pagetable (and referenced pages) can be found and loaded. It is not necessary to reload all mpages to restore the latest consistent state. Only the pagetable itself has to be loaded[4] and all mpages have to be marked as invalid. Then the system itself loads the pages one after the other if page-faults occur. Figure 4.14 describes the control structure of the operations of the `load_page()` operation after a page fault and the `backup()` operation. If the *Client OS* uses MemoryPagers of the secure environment, it should be possible to store its state, too.

To fulfill requirements of subsystems which need to know if their internal state has been stored (e.g. to guarantee atomic operations) the system should send a

[4]In general a pagetable is divided into hierarchically-ordered page-aligned subpages, therefore only the root pagetable has to be loaded into the memory.

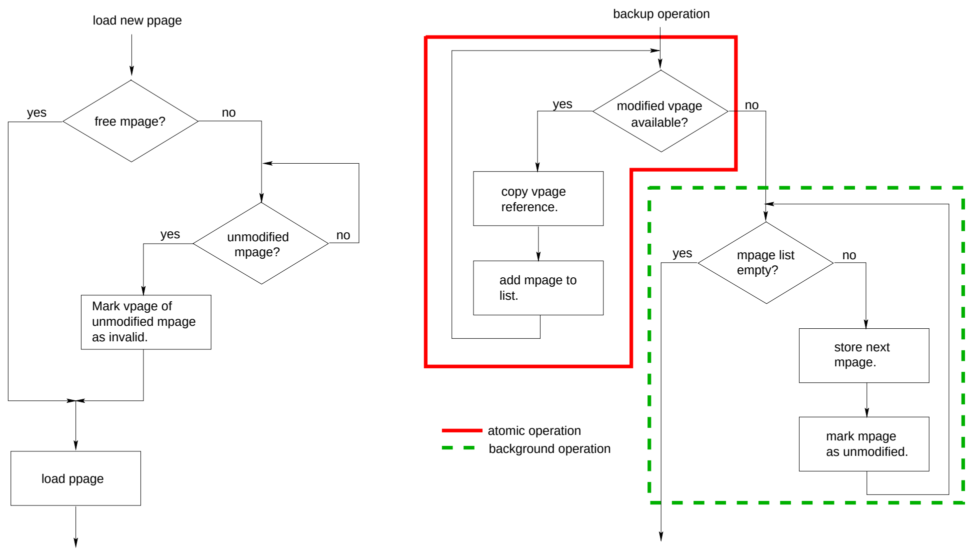

Figure 4.14: Program control structures of the `load_page()` and `backup()` operation.

message to all subsystems if a backup operation has finished. An interface which allows subsystems to invoke the system to start the next backup allows the administrator to select a backup interval between permanent (started after the last backup) and never (only subsystems invoke the system).

#### 4.5.2.2 2nd Approach to Provide Persistent Memory

Although Section 4.5.2.1 introduced a design to provide persistence, this section proposes another, simpler approach.

As explained in Section 4.2 the Fiasco $\mu$-kernel provides an abstraction of memory regions, called *address spaces*. The granularity of address spaces is processor-page aligned, therefore *dynamic memory* (*heap memory*) cannot be provided by the microkernel. Because C and C++ developers expect to be able to use dynamic memory, this has to be provided by additional functions. Section 4.12.1 suggests a simple dynamic memory model, but to explain the persistency model I have to anticipate some results here: The dynamic memory model manages small memory regions and requests new pages from the MemoryPager if no more memory is locally available. If the system would contain another MemoryPager (or pages with another attribute, see Section 4.5.2) providing persistent memory pages the dynamic memory manager can be extended in such a way that it provides additional functions to manage also persistent memory regions which have to be stored/loaded on shutdown/startup as described in Section 4.5.2.

The advantages of this approach are that implementation is easy, because only a new memory attribute is required, and that specific regions of memory can be stored to a persistent storage explicitely. Disadvantages are that developers have to decide themselves whether objects have to be persistent or not, and that management of dynamic memory structures (e.g. trees) produces additional overhead. Potential sources of errors are mixtures of persistent and transient objects (e.g. a transient object has to become persistent if it is appended to a persistent data structure).

### 4.5.3 Package: **PortManager**

The current version of FIASCO and its resource manager does not provide mechanisms to limit I/O port access. Therefore a new interface called PortManager has to be defined to handle access to I/O ports. To be compatible to the behaviour of existing servers, the implementation should grab all existing ports during its initializing phase and handle them on a first-come first-serve basis. Because x86 processors support highly granulated access control of I/O ports, workarounds to increase the granularity are not required:

The Intel 80x86 processor family since i386 supports four different privilege levels, also known as rings of protection ([3], [22], [11]). Every task runs in one privilege level, stored in the CPL (Current Privilege Level) of the task's EFLAGS register. It is possible to specify privilege levels (Input/Output Privilege Level) which have permission to access the I/O ports by setting the IOPL flag of the processor. This means, it is possible to forbid I/O access completely, or to create privileged tasks, which are allowed to access I/O ports in principle. If unprivileged tasks access I/O ports, an exception is raised. If privileged tasks access I/O ports, the I/O Permission Bit Map (IOPBM) inside the Task State Segment (TSS) is checked to see if access to the selected port is allowed. This makes it possible to control I/O port access to every port separately.

Table 4.1 describes two required messages the PortManager has to provide, one message to reserve port access and one message to release ports. The first parameter

| Message | par1(16 bits) | par2 (16 bits) | result |
|---|---|---|---|
| reservePort | first_port_nr | nr_of_ports | ok/error |
| releasePort | first_port_nr | nr_of_ports | void |

Table 4.1: Two new IPC Messages used by the RMGR to assign/release I/O port access to other tasks.

contains the port number, the second parameter contains the number of ports to be reserved. This makes it possible to reserve a continuous range of ports (devices often need more than one port). Both parameters have a length of 16 (65536 I/O ports) bits, thus they fit into one 32 bit register. The PortManager interface can be implemented by the existing resource manager RMGR of the FIASCO package or by any a separate service. Because granting/removing of permissions to access I/O ports requires modifications of the IOPBM and thus of the task's TSSs, two different approaches are possible:

1. The FIASCO $\mu$-kernel is the only instance which is allowed to modify a task's TSS register and therefore change the I/O port access rights and the PortManager reserves all ports during initialization. This approach is also used by FIASCO to grant permissions to start new tasks, but I fear it requires to modify the microkernel.

2. Alternatively we can leave the microkernel untouched and move the right to change the task's TSS register to the PortManager. This requires the PortManager implementation to be a higher privileged task. This is not compatible to the microkernel philosophy which demands all servers to be user-level tasks.

I prefer the first solution which corresponds to the FIASCO and microkernel philosophy, but because both approaches change only the interface between PortManager and $\mu$-kernel, this can be changed later. Although FIASCO does currently not support controlling of port access at all, later prototypes should contain a PortManager to prevent changes of other components if it is supported.

### 4.5.4 Package: DisplayManager

As explained in Section 3.4.8.1 the design of the DisplayManager depends on the size of the display. A display is "small" if usually only one application is shown (PDAs), thus it is defined to be "large" if more than one application can be shown at the same time (PCs). Section 4.5.4.1 contains a short discussion on small displays and the following sections suggest a design for big displays. I think that a lot of design decisions of the "Big Display" approach can also be used by the "Small Display" approach.

#### 4.5.4.1 Small Displays (PDAs)

General properties of small displays are: Usually only one application uses the whole display at the same time, therefore applications expect a fixed display size. Because the reserved region reduces the usable size of the display and applications should not (or cannot) be changed, a trusted component DisplayManager has to control access to the display hardware, write to the reserved region and provide a virtual display to applications which has the same resolution as the hardware display. The interface of the virtual display depends on the used *Client OS*: If it cannot be modified and a virtual hardware layer (see Section 4.5.1) is used the DisplayManager's interface should be similar to the interface of the hardware to make redirection of calls fast. If the *Client OS* can be modified, the DisplayManager's interface should be similar to the *Client OS*'s interface. However, a more abstract interface produces faster output, because calls between different layers occur less frequently.

To be able to see the whole application, the reserved region should contain two buttons to scroll the application's window to the top or bottom. It should be considered that also input data (mouse or pen positions) have to be adapted to the relative position of the region of the window.

#### 4.5.4.2 Large Displays (PCs)

On large displays in general more than one application is visible at the same time and they do not expect a fixed display size, because virtual displays (windows) are used. The most common approach to encapsulate display hardware in Unix systems is the *XWindows* system[5]. The heart of the *XWindows* system consists of a program called *XServer* which runs on a machine with a display, keyboard and a mouse. It contains device drivers and provides a common interface to all other application programs called *XClient*. The clients can be running on the same machine as the *XServer* or on another machine connected to the *XServer*. The *XClients* communicate to the *XServer* via a protocol language which is common across machine types. To simplify the communication between *XClients* and *XServer* the XLib library provides a large set of subroutines.

Many clients can simultaneously use the same *XServer*, because the server provides a virtual screen (window) to every client. One client which should always run is the *windowmanager*, which draws the window borders and makes it possible e.g. to resize and move windows. The *XWindow* system does not provide high-level support like buttons, menus, scroll-bars etc.; to use them additional GUI toolkits like Motif[6], Qt[7] or GTK[8] have to be used. Figure 4.15 outlines the interfaces between *XClient* and the video hardware.

If both the secure environment and the *Client OS* share the same display hardware using the *XWindows* system we get two (maybe virtual) protocol stacks which

[5]http://www.x.org
[6]http://www.motif.com
[7]http://www.troll.no
[8]http://www.gtk.org

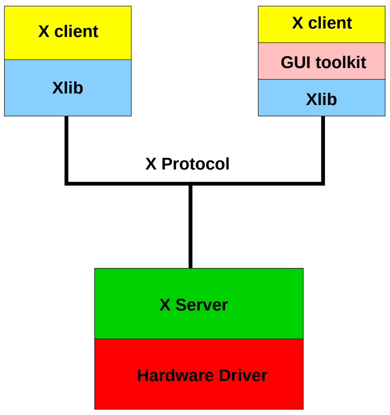

Figure 4.15: Components of the X Windows System.

have to share the same hardware driver (red box), as outlined in Figure 4.16. In general I prefer to use existing interfaces, thus the three colored lines show the possible interfaces where different implementations can be put together[9].

If the functionality below the blue line shall be moved into the secure environment the client OS XLib has to be replaced by a new XLib stub which communicates with the secure environment. This is possible because the XLib contains a fixed interface, but statically linked *XClients* have to be rebuilt.

Using the X protocol layer (green line) to separate untrusted and trusted components may be better, because it is a network protocol which provides redirection. Redirecting output to the *XServer* of the secure environment would leave *Client OS* applications untouched. If it is too difficult to establish a network connection between SE and *Client OS*, a client *XServer* stub could be used to forward X protocol messages.

The third approach uses a *Client OS XServer* which acts as a client of the secure environment *XServer*. Thereby all *Client OS* windows are summarized in one *XWindow* of the secure environment. This makes it necessary to use two windowmanagers: One for the *Client OS* and one for the SE.

I prefer the second solution because it seems to be the most flexible one and *Client OS* applications can be left untouched, but currently I cannot decide if it is possible to implement it.

## 4.6 Package: **SubsystemAuthentication**

If the secure environment provides an abstraction of the display hardware, as described in Section 4.5.4.2, two ways to inform users about subsystem trust are possible:

1. A reserved region of the *XServer* can be used to show the trust level of the application which currently gets user input (has the focus).

2. A trusted windowmanager implementation can show the trust level using a reserved region of the window (e.g. the headline).

The second approach has the advantage that it shows trust levels of all applications and that the whole display resolution can be used. The disadvantage is that

[9]It is meaningless to put the GUI toolkits together, because too many toolkits with different interfaces exist and they have to fullfil different demands. The client OS toolkits have to be user friendly, the toolkit of the Secure Environment has to be secure.

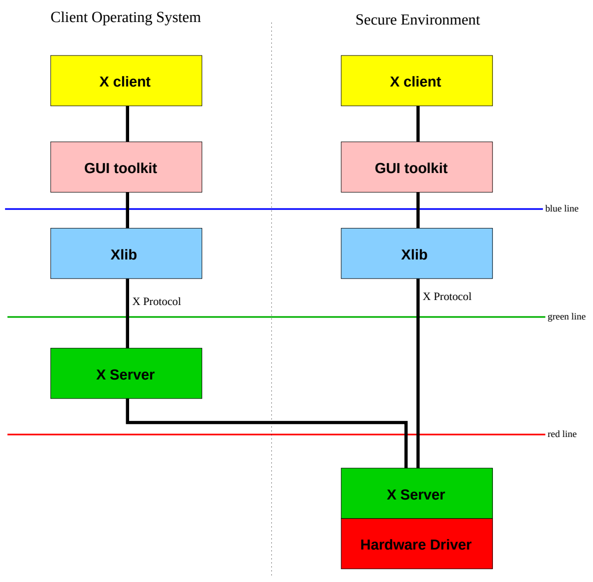

Figure 4.16: The three colored lines describe possible interfaces between untrusted and trusted components.

a trusted windowmanager implementation is necessary. On small displays both approaches are equal because only one application window is visible at the same time.

## 4.7 Package: **TrustedGUI**

Possible approaches how to provide a GUI toolkit were described in Section 3.4.8.2. The FIASCO microkernel itself currently does not provide a shared library concept, thus the second approach cannot be used. I decided to use the first one (static library) because of the following reasons:

Separation of application and GUI toolkit is not necessary, because the display hardware is protected by the DisplayManager. Also waste of memory and complexity of evaluation are not primary problems, because they are obsolete as soon as shared libraries or similar concepts are available.

## 4.8 Package: **CommunicationPath**

To establish a virtual network connection the *Client OS* has to be extended by a virtual network device and the definition of the router of the secure environment has to be refined. These topics are discussed in the next two subsections.

### 4.8.1 Router

As decided in Section 3.4.9 the Router routes all network packets between local (software and hardware) devices. To make the implementation fast and efficient, the packet size is page-aligned[10]. This makes it possible to send/receive packets via page mappings. Figure 4.17 outlines the interface of the Router subsystem. In order

[10]This is possible because the IP protocol is designed to use different underlying packet sizes while transferring an IP packet. If a packet is transmitted from a larger size to a smaller size, IP packets are divided into subpackets.

to simplify extensions device proxies are used which forward packets to registered protocol proxies (TCP, UDP, IPX) which themselves forward them to registered services (ping, telnet, ftp). Keep in mind that these are only interface definitions which provide high flexibility, implementations only have to provide the Device interface (e.g., the LINUX kernel only has to implement the Device interface because it does all other tasks by its own TCP/IP stack). But if certified implementations of ProtocolManager and ServiceManager exist, providing new Services (for the secure environment) becomes very easy.

## 4.8.2 Virtual Network Device

This section explains how the LINUX 2.2 kernel can be extended by a virtual network device (see Figure 3.13). Most information about LINUX network device drivers are extracted from [39] and [12].

A network interface represents a thing which sends and receives data packets and has to be distinguished from network drivers which access network adapters. Therefore a network interface can be a hardware device like an ethernet card or a software device like the loopback device[11], in any case it represents a unique IP number.

LINUX network drivers are, in contrast to other Unixes, neither block nor character devices. Instead they are directly connected to kernel structures. Each device is created by filling in a protocol-independent struct `device` and registering them via a kernel function `register_netdev()` call. Every device has a unique name which traditionally indicates its type, thus ethernet devices are named "eth0", "eth1", etc. A device driver which receives new data packets in a device-type dependent way passes them up to the kernel protocol layer using the kernel function `netif_rx()`. If the kernel wants a device to send packets it invokes the `hard_start_xmit()` function of the device driver.

The virtual network device has to do three tasks to adapt it to the Router (Figure 4.17):

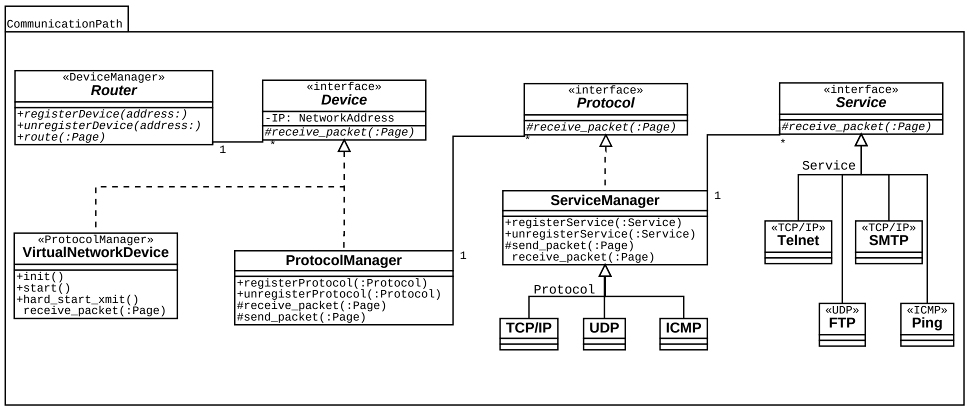

Figure 4.17: Components of the CommunicationPath package.

***1. Register with IP number to the Router.*** The device's network address is set by network administration tools (`ifconfig`) on initialization, but the device is opened (activated) by another command ("`ifconfig <> up`"). Therefore the

[11]A network device used by many Unix'es to test the TCP/IP stack and which provides local network functions without network adapters. Usually the loopback device ("localhost") uses the IP address 127.x.x.x.

virtual network device should register the thread which receives incoming messages when the device is opened and unregister it when the device is closed.

*2.* ***Forward packets received from the*** **Linux** ***kernel to the Router.*** The device has to provide a `hard_start_xmit()` function which can be invoked by the Linux kernel, which sends the received packet via $\mu$-kernel IPC to the router.

*3.* ***Forward packets received from the Router to the*** **Linux** ***kernel.*** The receiver thread which receives network packets forwards them via `netif_rx()` kernel call to Linux' TCP/IP stack.

The sequence diagram outlined by Figure 4.18 shows the control structure if Linux pings a device of the secure environment.

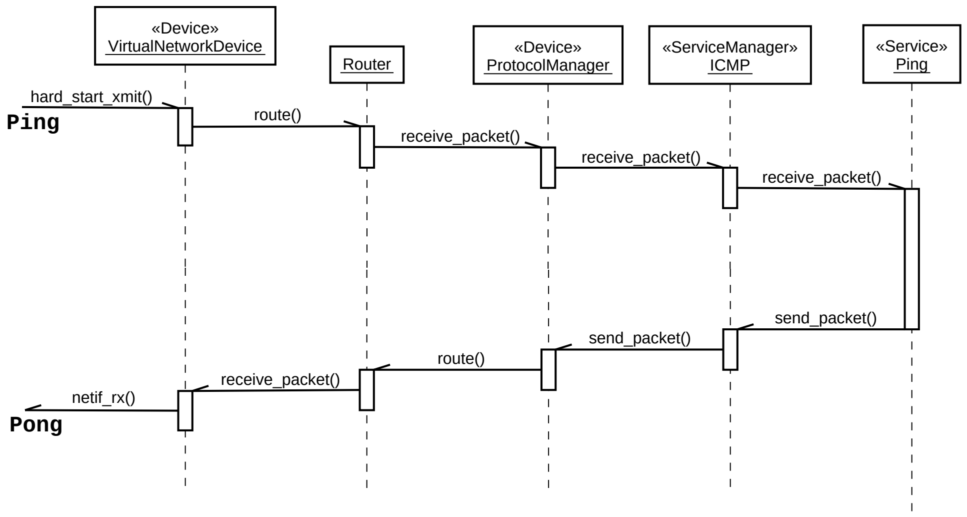

Figure 4.18: Control structure of a ping command invoked by Linux.

## 4.9 Package: SubsystemManagement

The package SubsystemManagement provides components to grant/manage permissions to start new subsystems (tasks) and related classes which are necessary to install/update/delete subsystems. The class Loader has been removed from this package and separated into the package Loader, because it can also be used for other tasks than loading new subsystems.

### 4.9.1 TaskManager

This service acts as a facade to all other components of this section, because it provides a simple interface to start/halt tasks (external functions [F 40] and [F 45]). Figure 4.19 illustrates available methods provided by this service. A new subsystem can be started by defining the destination where the content and optionally the *content-certificates* can be found. The TaskManager invokes the Loader to load the new software and a content-description provided by the content provider. Among other things (see Section 4.9.1.1) the content-description contains a hash value of the content and a signature of the content-provider. Both have to be tested by the TaskManager. The content description also contains a list of related content-certificates which have to be loaded. Then the DerivationService is invoked to derive the permissions of the new content. Next the TaskManager invokes the Installer to parse the content and extract starting points. At least it updates the naming service and dependency database.

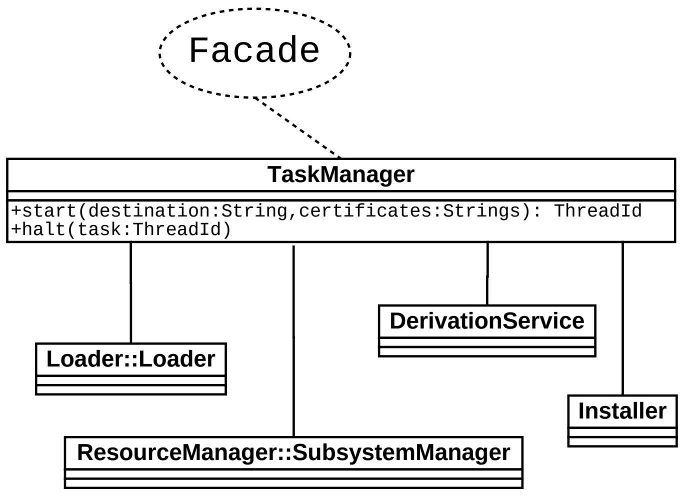
Facade
TaskManager
+start(destination:String,certificates:Strings): ThreadId
+halt(task:ThreadId)
Loader::Loader
DerivationService
Installer
ResourceManager::SubsystemManager

Figure 4.19: Design of the class TaskManager.

Fiasco does not support starting of subsystems within a clan of another subsystem, only granting the right to start a specific task is supported. Thus to control who can start new tasks the security policy has to control these rights. If users have permission to start new subsystems themselves the security-policy cannot control which subsystem is installed (e.g. enforce that ACEFs are used). Therefore the permission to access the TaskManager has to be restricted by the security-policy to the user's ACEF. Four different scenarios should be considered:

1. Users have permission to start new subsystems in their clan without restrictions.
2. The security policy has to control which subsystems are installed by users and an ACEF has to be used to control every subsystem.
3. Installed subsystems may want to create their own subtasks.
4. If the security policy allowes users to install subsystems which have permission to start subtasks on their own, users are able to install every subsystem whithin their clan, because they could have installed a TaskManager.

I am not sure, but I fear that it increases the security of the system if users can install any subsystem, even if they are controlled by the user's ACEF. If this is true, it should be possible for the security policy to control the right to start new subsystems. One approach could be to restrict the right to start new tasks to ACEF instances. Nearly all requirements of the scenarios mentioned above can be fulfilled:

- To permit users to start any subsystem within their clan the user's ACEF does not have to restrict rights to start new subsystems.
- In order to control which subsystems are installed the ACEF can check which subsystem the user wants to install.
- To permit subsystems to start tasks on their own, the ACEF can grant the right to start a new task to a child task. This is only useful if the ACEF checks which subsystems are installed.

This approach seems to be flexible enough to fulfill requirements of all security policies.

#### 4.9.1.1 Content-description

To make installation/updates of software more comfortable and less error prone the content provider has to provide a *content-description* which contains all necessary information for the TaskManager to install new content and to check if the system provides all required services. The content-description shall at least contain the contents summarized by Figure 4.20. Later versions of the content-description could

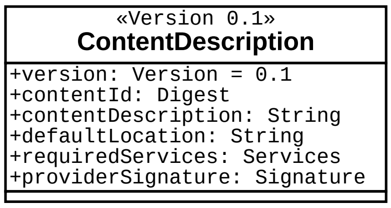

Figure 4.20: Contents of the content description.

provide suggestions how roles should be assigned to provided methods or assign security-levels to provided methods.

### 4.9.2 Package: **Installer**

The installer provides function [F 52] which parses linker-format dependent object-files to start them if they are valid executables. The proxy pattern is used to make management of new linker formats easier (see Figure 4.21). This design assumes

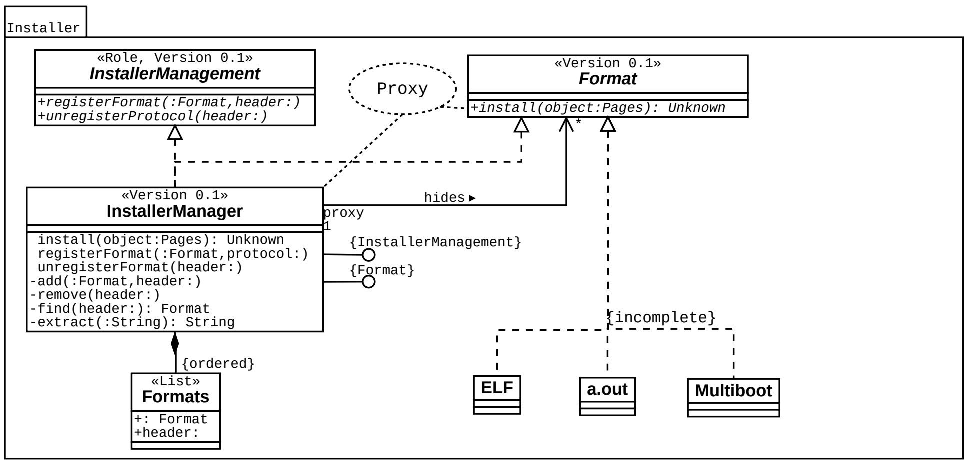

Figure 4.21: Design of the Installer package.

that linker formats can be distinguished by some kind of headers which is correct for ELF, a.out and multiboot headers but perhaps this has to be changed. The return value of the `install()` method is currently unknown.

### 4.9.3 Package: **DerivationService**

The DerivationService helps users to derive permissions of subsystems to be installed and this section suggests a sample implementation.

If new software is installed two opposed interests have to be solved: The USER or ADMIN who installs the software wants to grant the new software as few permissions as possible to keep the possibility of security holes small. In contrast the software to be installed needs some services to perform actions.

The services needed by software components are listed by content-descriptions presented in Section 4.9.1.1. Thus the DerivationService's task is to derive the smallest set of permissions which are required to access these services and to decide if these permissions can be granted to the new software component.

To perform the first task the DerivationService needs access to the NamingService and the access control database. The decision if derived permissions can be granted to the component depends on the trust in the correctness of the implementation and depends on the security-policy and user-defined values (Section 3.2.5 lists some possible criteria). I decided to use *content-certificates* (presented in Section 4.9.3.1) to derive the level of trust. Users have to define trust into content-certificate providers, and rules how many certificates are necessary to match a specific trust-level. Then security-policies decide if the derived trust level suffices to grant permissions to the software. This is very similar to trust levels of public key infrastructures provided by OpenPGP certificates [2].

It is not necessary to be able to enforce global installation restrictions locally, to ensure, e.g., that only subsystems which match requirements defined by the SECURITY-ADMIN can be installed. Instead, the permission to start new subsystems, provided by the interface SubsystemManager of the ResourceManagement package, can be limited to a trusted subsystem (e.g. a global TaskManager). Then all users are only able to install subsystems if they have permissions to access the global TaskManager and the global security policy is enforced. Thus the decision whether global installation restrictions are enforced, when users install applications locally, becomes security-policy dependent.

#### 4.9.3.1 Content-certificate

*Content-certificates* provided by trusted parties (e.g. the content provider or an independent instance) are used to derive the trust level as explained, e.g., in [23]. The certificate has to unequivocally identify the content description (e.g. using a hash function) and the provider of the certificate has to sign it to ensure integrity. The SECURITY-ADMIN has to define rules which describe how the DerivationService has to deduce evaluation results of the content-certificate provider into the local security policy. In order to test signatures of content-certificates a public key infrastructure is required. If the content is provided by source code the certificate should additionally contain a list of compatible compilers to be used. Figure 4.22 illustrates required contents of content-certificates.

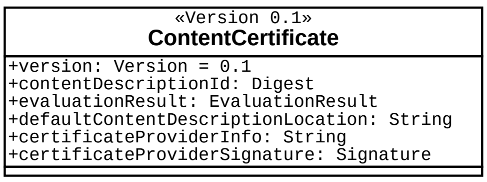
«Version 0.1»
ContentCertificate
+version: Version = 0.1
+contentDescriptionId: Digest
+evaluationResult: EvaluationResult
+defaultContentDescriptionLocation: String
+certificateProviderInfo: String
+certificateProviderSignature: Signature

Figure 4.22: Contents of the content-certificate.

The attribute `evaluationResult` describes the trustlevel (or similar contents) the certificate provider suggests using this certificate. The transformation into access permissions is a task of the security policy.

### 4.9.4 Conclusion

Figure 4.9 outlines the complete design model of the SubsystemManagement package.

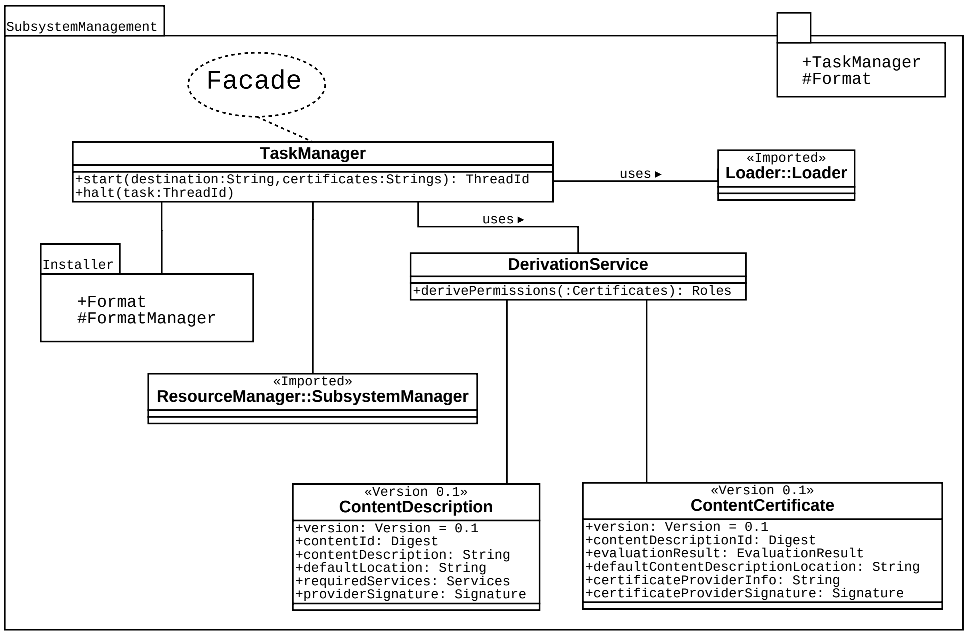
SubsystemManagement
Facade
+TaskManager
#Format
TaskManager
+start(destination:String,certificates:Strings): ThreadId
+halt(task:ThreadId)
uses ▸
«Imported»
Loader::Loader
uses ▸
Installer
+Format
#FormatManager
DerivationService
+derivePermissions(:Certificates): Roles
«Imported»
ResourceManager::SubsystemManager
«Version 0.1»
ContentDescription
+version: Version = 0.1
+contentId: Digest
+contentDescription: String
+defaultLocation: String
+requiredServices: Services
+providerSignature: Signature
«Version 0.1»
ContentCertificate
+version: Version = 0.1
+contentDescriptionId: Digest
+evaluationResult: EvaluationResult
+defaultContentDescriptionLocation: String
+certificateProviderInfo: String
+certificateProviderSignature: Signature

Figure 4.23: Overview of components of the SubsystemManagement package.

## 4.10 Package: **Loader**

The Loader provides the requested function [F 35] and has to consider that different protocols exists which can be modified and/or provided by different providers. To use only one interface to access different protocols and to make installation and deletion of single protocols easier the proxy pattern (presented in Section 4.3.2) is used. To define the destination address of the content to be loaded a String type is used and the protocol has to be defined explicitly by these address, e.g. "`http://www.linux.org`"[12]. The manager internally uses an ordered list of protocols to manage protocol implementations. The return value of the `load()` operation are memory pages allocated by Loader implementations. Figure 4.24 outlines contents of this package.

Figure 4.25 outlines two sequence diagrams which illustrate dynamic behaviour of manager and protocols if a) a Loader implementation adds itself to a LoaderManager and b) an external subject invokes the LoaderManager to load new contents. Multi-threaded behaviour of the manager and protocols is possible but ignored here to keep the diagrams simple.

## 4.11 Package: **Random**

Classes related to random bit generators have been extracted into the package Random. It contains only one public instance of the class RandomManager which is accessible by other components and provides function [F 20] (see Figure 4.26). The design of this package is similar to the Proxy design explained above, but instead of delegating messages to a random Source, it combines data of different implementations using an `xor()` function. The operation `random()` returns Pages which can easily be mapped into another address space. The class RandomManager caches a predefined set of random pages for efficiency. To increase the user's trust

[12]Later versions can be extended in that way that, if no protocol is selected explicitly, the LoadingManager asks every protocol if it is responsible.

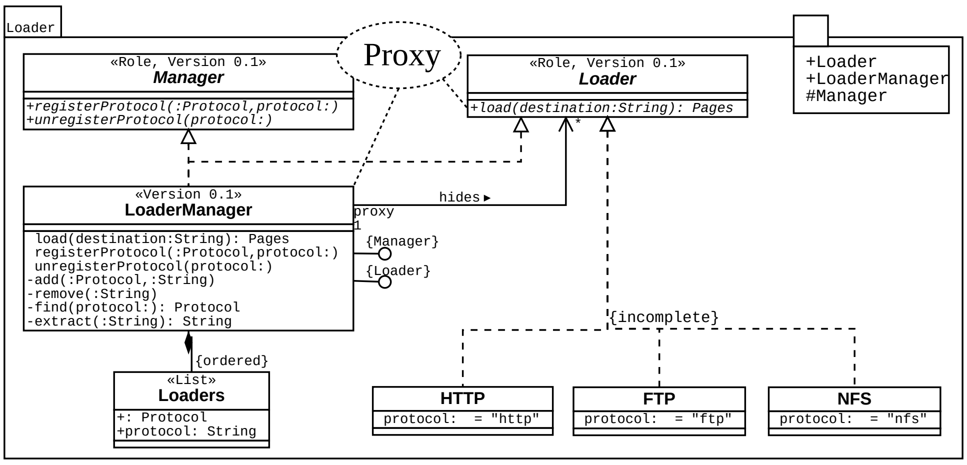

Figure 4.24: Design model of the package Loader.

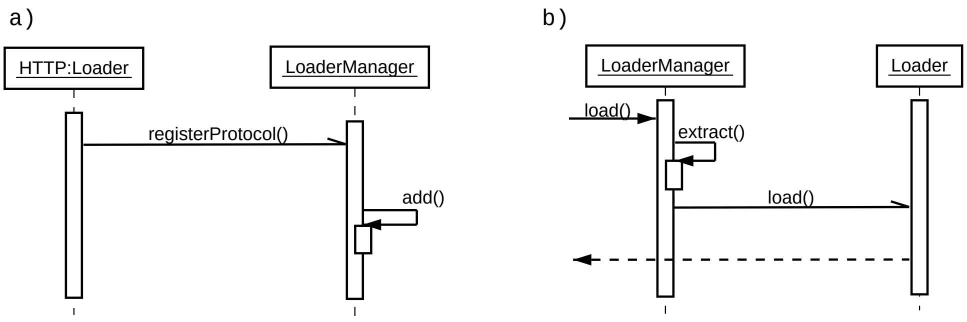

Figure 4.25: Sequence diagrams if a) a new protocol is added and b) the Loader-Manager is invoked to load new data.

into the correctness of created random bytes, the class Keyboard creates random pages from keystrokes or similar user-dependent input and implements [F 25].

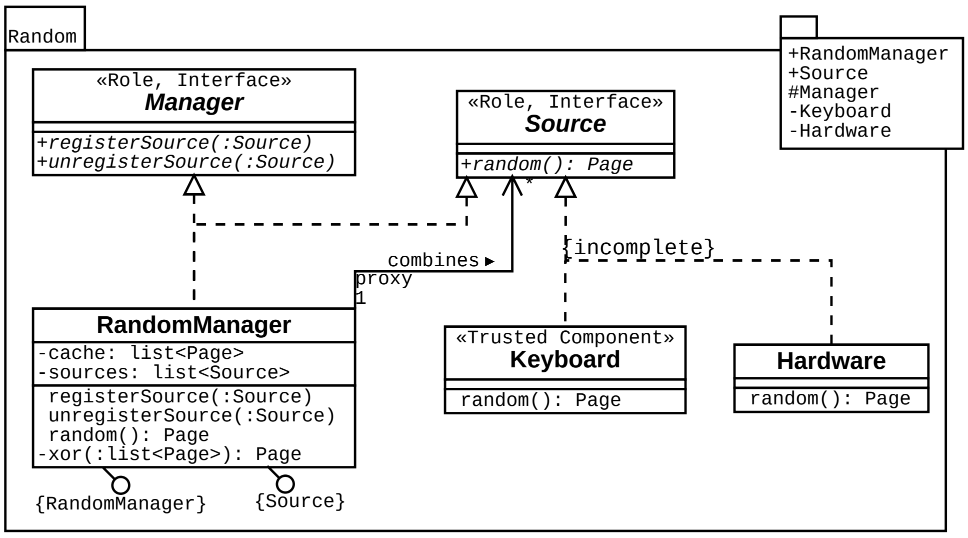

Figure 4.26: Design of the Random package.

If the RandomManager implementation is correct and at least one Source provides strong random numbers, the output produces strong random numbers. Therefore users are free to create their own RandomManager instance which uses a system-wide RandomManager and further user-dependent Sources (Figure 4.27).

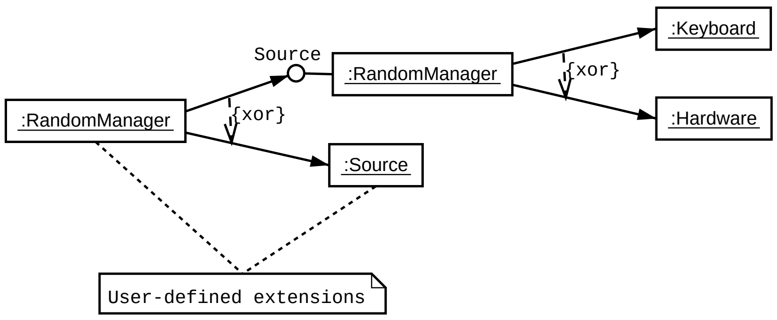

Figure 4.27: Scenario if a user extends the system-wide random service by his/her own Source.

## 4.12 Package: **Utilities**

This package describes designs of classes which build a framework to help developers developing secure applications and services. Implementations of this package are not security-critical, because they are running within protected address spaces, but errors can of course decrease the reliability of services which use these classes.

### 4.12.1 Class: **Heap**

The class Heap hides implementation details of dynamic memory allocation which has to be provided by additional functions.

If the $\mu$-kernel executes a new thread the program data is mapped read-only into the task's address space and a pointer to the stack is defined. Automatic memory is stored on the stack by writing to the stack's address. If the thread tries to write to an address which does not contain a physical page, a page fault message is sent to the thread's page fault handler which can request a new page and map it into the thread's address space.

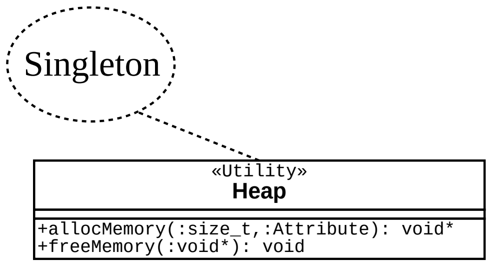
Singleton
«Utility»
Heap
+allocMemory(:size_t,:Attribute): void*
+freeMemory(:void*): void

Figure 4.28: Design of the class Heap.

Because dynamic memory has to be valid until it is deleted explicitely it cannot be stored on the stack. It is the Heap class' task to manage allocation and deletion of dynamic memory regions and to ensure that neither dynamic memory and stack memory, nor two dynamic memory regions overlap. To guarantee this requirement it should only be possible to instantiate only one Heap within one address space, therefore it is designed using a Singleton pattern[13] (Figure 4.28).

A more complex environment providing dataspaces and dynamic memory with different attributes can be found in [34]. I did not use this implementation because the current version did not compile.

### 4.12.2 Class: **ThreadManager**

The class ThreadManager helps creating/deleting/looking for threads within an address space. It is used by the class Thread, presented in subsection 4.12.4.

### 4.12.3 Class: **Pager**

This class acts as the default pager of all threads within one address space. It is used to be able to debug page faults of other threads and to be able to use virtual pages, because it maps any free page to the faulting address.

### 4.12.4 Class: **Thread**

The class Thread hides implementation details of the creation of threads and provides some basic functions which should be available to all applications. Similar to the Java class Thread of the java.lang package a new class of threads can be defined by overwriting the abstract `run()` method. The new thread is started by invoking the `start()` method. Methods or data which should only be instantiated once within an address space can be defined as static members of the class Thread. Example data members are the class ThreadManager (Section 4.12.2), Heap (Section 4.12.1) and Pager (Section 4.12.3).

[13]If the class Thread is used this is enforced because the Heap is a static data member of the class Thread.

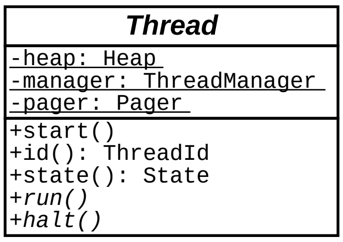

Figure 4.29: Methods of the class Thread.

## 4.13 OOD Model

Figure 4.31 summarizes the complete OOD model. The packages KeyManagement and Crypto are taken from Chapter 3.

Figure 4.30 illustrates layers and their components of the design model. The difference between this model and the model proposed in Section 3.15 is an additional layer "Subsystem Framework" between red line and secure platform. It provides utility components to be used by all subsystems.

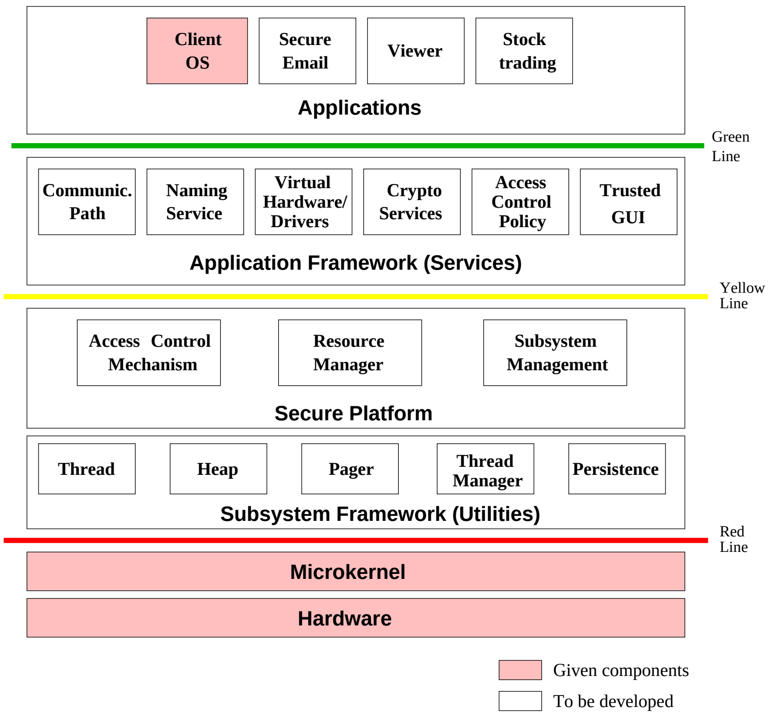

Figure 4.30: Layers of the design model.

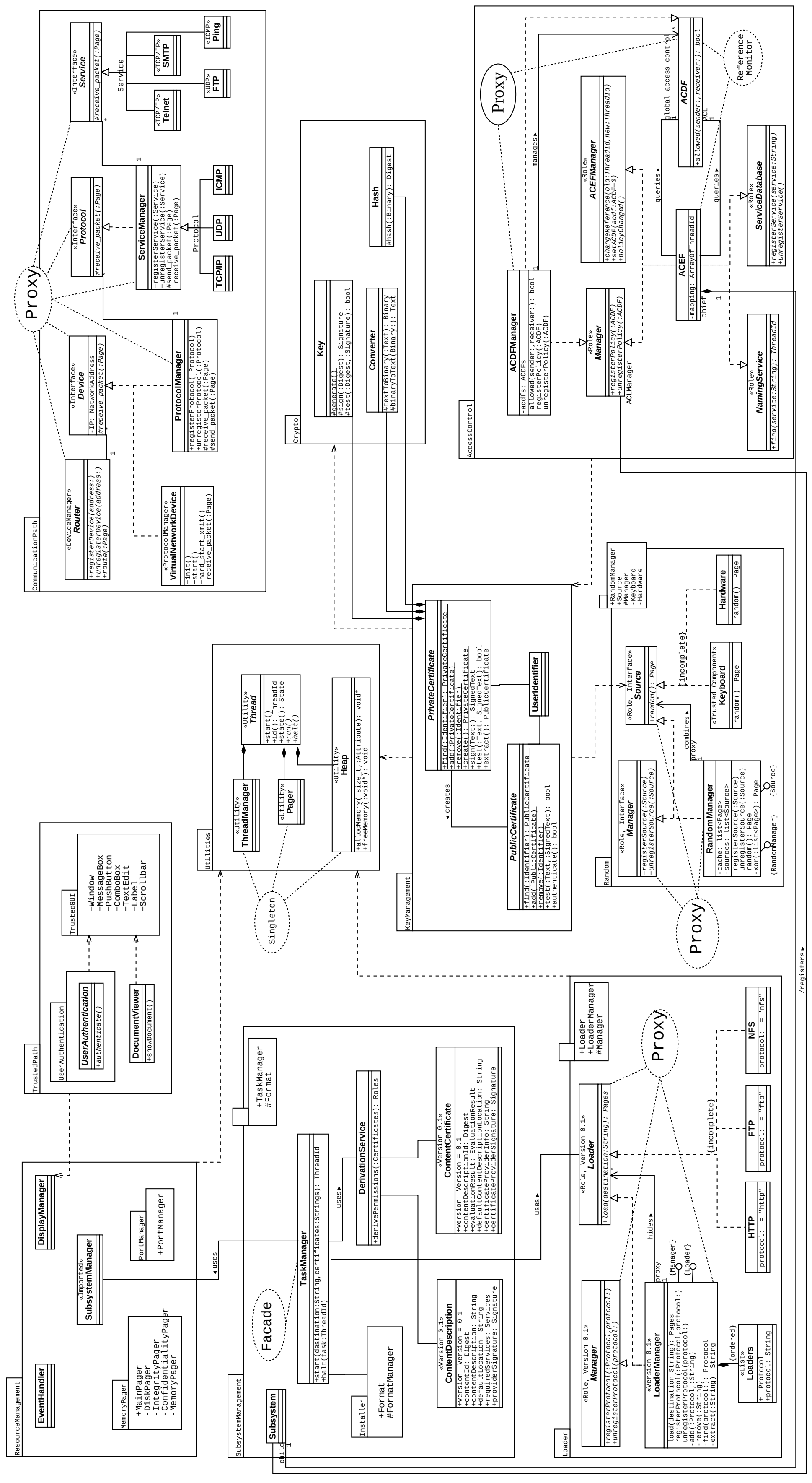

Figure 4.31: Components of the OOD model.

# Chapter 5

# Implementation

This chapter contains information about the prototype developed in parallel to this diploma thesis. The next section contains some general hints on development of the prototype and related components and short explanations of utility classes necessary to write nearly all other components. It also discusses the design of secure services implemented so far and describes required LINUX kernel modifications. Section 5.2 contains information about download, installation and use of the prototype.

## 5.1 Instead of a Developer's Manual

This section contains developer-specific information which should later become a separate DEVELOPERMANUAL.

### 5.1.1 Some Hints on Development

In this section some information about differences and peculiarities of programming and compiling applications using FIASCO are discussed and it is explained how I tried to solve some of these problems.

***Programming Language***. Usually any programming language can be used to develop PERSEUS applications if it is possible to create wrappers which provide Fiasco's system calls. In my opinion using object-oriented languages *can* lead to better implementation design, because it makes it easier to hide implementation details and it has a better data type control. Thus my first task was to make it possible to use C++ to implement subsystems. Another important point which reduces the count of errors is to use existing data structures whenever possible.

***Start Files.*** In general, the `g++` compiler links some operating system dependent start files to the compiled object which initialize some data and then invoke the `main()` function. The start file provided by the compiler cannot be used if FIASCO applications are created, thus linking against it has to be prevented using the `--nostartfiles` option. Instead a small assembler file `crt0.S` is used which does some initialization and reserves stack space. The current version does not initialize global (and static) data. Until now I got no information how to do that, thus using global objects can lead to errors.

***Load Address.*** The compiler option "`-Wl,-N,-Ttext,<address>`" defines the address where the RMGR should load the application to. This is necessary, because the RMGR cannot load subsystems to a virtual memory address. The developer has to ensure that all applications are loaded to different addresses and that they do not overlap.

***Task Creation.*** Another problem occurs if a task (chief) starts a new task (child) which uses objects containing static data members. In general, the chief starts a pager which maps the child's code into its address space read-only on demand. Automatic data is created on the child's stack, but static memory is not created again, thus the child tries to access the chief's static data. Because every address space should contain its own instance of static memory, the pager has to *copy* the demanded page in this case, but the problem is that the copied data is not re-initialized (e.g. imagine a task's thread-counter, which should be reinitialized to zero on a new task). Solutions could be to avoid using static data members, or to implement shared libraries or something like that. [34] could be a good starting point.

***Dynamic Memory.*** FIASCO does not provide dynamic memory management (except page allocation/deallocation), thus only automatic or static data can be used, or the application has to provide its own memory management functions.

***STL.*** The probability of errors can be reduced if developers can use existing data structures which are well-testes or evaluated. C++ provides a powerful library containing standard data containers called *Standard Template Library*, short STL. Development of new applications would be much easier if a library providing the well-known STL interface could be used. The free *CORBA* implementation MICO[1] contains a separated implementation called `mini-stl` which is much smaller than e.g. the STL provided by the GNU gcc compiler.

### 5.1.2 Directory Structure

According to the GNU standard, sources are summarized in the subdirectory `src`, additional documentation can be found in the subdirectory `doc` and the directory `tests` contains some test applications. The directory hierarchy of the `src` directory is similar to the respective package names of the last chapter. The main directory contains configuration files of automake/autoconf, explained by the next section.

### 5.1.3 Automake/Autoconf

The GNU autoconf info pages explain: "Autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of UNIX-like systems. The configuration scripts produced by Autoconf are independent of Autoconf when they are run, so their users do not need to have Autoconf.".

To generate autoconf-dependend makefiles `Makefile.in` the tool GNU automake can be used. In this case developers only have to write very simple `Makefile.am` files.

The most important file of autoconf is `configure.in` which contains all tests, to be done by the configuration script, and general makefile definitions. Automake/autoconf info pages contain more information.

### 5.1.4 Creating the Reference Manual

To generate a REFERENCEMANUAL (html and latex version) the tool `doxygen`[2] is used which parsed C/C++ header files to extract information out of JAVADOC, DOC++ and kdoc[3] compatible sourcecode comments. I decided to use this tool, because it is more powerful and stable than the other tools mentioned above.

[1]http://www.mico.org
[2]http://www.stack.nl/~dimitri/doxygen
[3]http://developer.kde.org

### 5.1.5 Writing Secure Applications

According to the description of this diploma thesis later evaluation of the implemented components can be ignored, thus security-related implementation restrictions or rules have neither been drawn up, nor considered.

### 5.1.6 Utility Classes

This section contains an overview of important classes which make development of secure applications easier. The following subsections do not contain detailed interface definitions or descriptions of the behaviour, because this is the REFERENCE-MANUAL's task. Instead, short descriptions of design decisions and implementation details are explained.

#### 5.1.6.1 Class: **Heap**

The class Heap is a very simple heap implementation to evaluate the design of the secure environment and to be able to use dynamic memory. It allocates an increasing

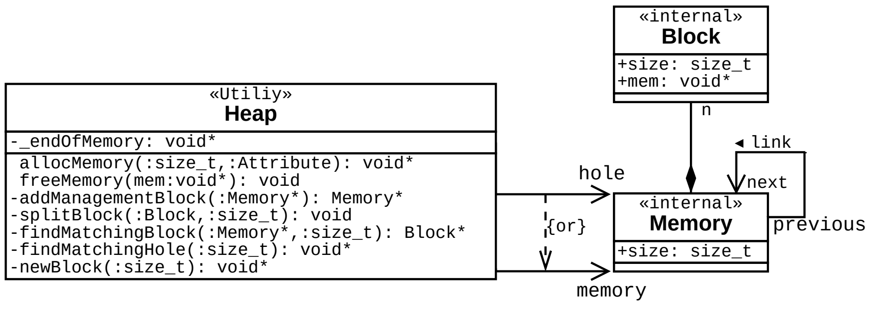
«Utiliy»
Heap
-_endOfMemory: void*
allocMemory(:size_t,:Attribute): void*
freeMemory(mem:void*): void
-addManagementBlock(:Memory*): Memory*
-splitBlock(:Block,:size_t): void
-findMatchingBlock(:Memory*,:size_t): Block*
-findMatchingHole(:size_t): void*
-newBlock(:size_t): void*
«internal»
Block
+size: size_t
+mem: void*
n
◂ link
next
previous
hole
{or}
memory
«internal»
Memory
+size: size_t

Figure 5.1: Classes and their relations of the Heap implementation.

amount of memory pages aligned above statically mapped data. To manage memory blocks it uses two lists of the class Block, `hole` and `memory`. The list `memory` contains pointers to allocated memory blocks in use, the list `hole` contains pointers to deleted memory blocks. In order to be able to dynamically enlarge the list of memory blocks, arrays of Blocks are stored by Memory data structures which themselves are stored within blocks. Nearly all private methods of the class Heap are used to find/create/resize Block and Memory structures. The member `_endOfMemory` is used to store the end address of used memory. To be able to provide error checking, new memory pages are requested from the Heap's pager, but this it not a must. Figure 5.2 describes the hierarchical dependency between methods of class Heap by a function tree.

A more complex environment providing dataspaces and dynamic memory with different properties can be found in [34]. I did not use this implementation because the version I have did not compile.

#### 5.1.6.2 Class: **Interrupt**

This class hides details of registering and receiving hardware interrupts. It reserves the interrupt number, given as constructor argument, and starts a new thread which receives the $\mu$-kernel's interrupt messages. The second constructor argument expects an implementation of the InterruptHandler interface. If interrupts occur, the receiving thread invokes the `irq()` method of the InterruptHandler. This class is used by the KeyboardManager.

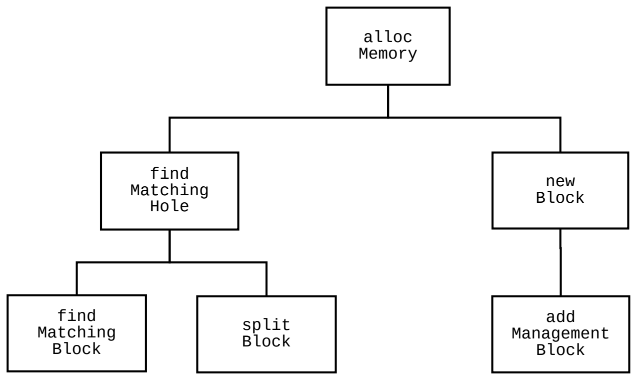

Figure 5.2: Hierarchical dependency of the public method `allocMemory()` of the class Heap.

#### 5.1.6.3 Class: **KeyboardStub**

This class translates scancodes sent by the KeyboardManager into ASCII codes. Currently only a subset of all possible codes are supported, other scancodes are converted into spaces. To use it create a new instance and initialize it with the ThreadId of the KeyboardManager. The method `next()` returns the next valid ASCII code. It is used by the AuthenticationManager.

#### 5.1.6.4 Class: **Thread**

The functionality provided by this class has been discussed in Section 4.12.4. Because static data members are currently not initialized by the init file a small workaround is used. The static data members Heap and Pager are returned by static methods `heap()` and `pager()` which themselves contain appropriate static data members. A ThreadManager is currently not implemented, instead the static method `newThreadId()` returns a ThreadId of an unused thread.

### 5.1.7 Packet: **AccessControl**

I decided not to implement ACEFs and ACDFs as described by Chapter 4 because of the following reasons:

1. The design of the access control packet has not finished yet. New *use cases* may lead to new requirements which change the API of the ACEF or ACDF. Additionally it has to be examined whether the design fulfills all security-related requirements.

2. Core developers of the microkernel to be used should take part of the design of the ACEF and ACDF, because they have more experiences about contrains and abilities of the $\mu$-kernel.

3. Because ACEFs have to start encapsulated subsystems as child tasks, implementations of the SubsystemManager, Loader and Installer are required. This is (in my opinion) very complex and would use a lot of time. A prototype which has a powerful access control implementation but no subsystems to control is useless. I think it is better to have some implemented subsystems which can then be used to examine new access control design models and implementations.

Thus the AccessControl packet contains only a (very) simple implementation of the interfaces NamingService and ServiceDB which may be implemented by the ACEF later. Because the NamingService has to be accessed by all subsystems (and no ACEF is available which can redirect the messages), the file `globals.h` of the directory `Interfaces` contains a static ThreadId called NS_TID which currently addresses the first thread of the first task started after the FIASCO-dependend boot modules. Later modifications (e.g. multi-threaded) should consider this. The headerfile also contains a definition of the maximum length of the interface name, named MAX_INTERFACE_LEN, currently fixed to 128 bytes.

If subsystems demand non-existent services, the current implementation returns a null address. This can cause problems if fast clients access the NamingService before slow services are able to register themselves. Thus I inserted loops into the start sequence of all clients (including LINUX) which requery the naming service if a query fails.

### 5.1.8 Packet: **ResourceManagement**

The subdirectory `DisplayManager` contains classes related to resources which provide a user interface.

#### 5.1.8.1 Class: **DisplayManager**

The class DisplayManager of the ResourceManagement/DisplayManager directory implements the TextDisplay interface by emulating a textmode VGA framebuffer. I decided to implement only the textmode, to keep the implementation of the prototype simple. It is the display manager's task to protect a reserved region of the display to show security-related information. Two different approaches are possible how clients can modify the video memory:

1. The DisplayManager provides an interface to be used by clients to modify the framebuffer, e.g. `putChar()`, `putString()`, `fillRegion()`, etc. This approach is similar to the *XServer*-related suggestion of Chapter 4.

2. The DisplayManager reserves the whole video memory but maps them into the address space of clients on demand.

Although the first approach should be desired (this makes it possible to provide the same interface even if the DisplayManager uses the graphic mode), I decided to implement the second approach because it can be implemented very easily. Additionally the first approach is slower because every letter has to be transmitted via IPC.

To protect a region of the display the DisplayManager moves the first line of the displayed memory area in such a way that the first line is on one memory page and the other 24 lines are on the next one. Because the DisplayManager maps only the second page into the address space of the client, the first line cannot be accessed by the client subsystem itself. A disadvantage of this approach is that now two 4kB video pages are required, but this should not be a problem because VGA video adapters have enough memory.

Subsystems should use the utility class TextConsoleStub (see Section 5.1.9.3), which hides details of accessing the DisplayManager. This makes it possible to change the DisplayManager in such a way that it uses graphic modes without modifications of subsystems. A good starting point to implement a graphic mode DisplayManager would be the abstract framebuffer interface provided by the LINUX kernel since version 2.2.

In order to be able to reserve and therefore protect video memory the DisplayManager has to be started before other untrusted subsystems can reserve these memory regions. Of course only direct access of the memory can be prevented, the DisplayManager cannot prevent that subsystems use the BIOS or the graphic controller (by using I/O) to modify the video memory. This has to be enforced by other components.

#### 5.1.8.2 Class: **KeyboardManager**

As the class DisplayManager provides mechanisms to manage user output, the class KeyboardManager, which implements the public interface Keyboard, hides details of the keyboard hardware. On startup the PC keyboard interrupt (1) is reserved and the keyboard controller is initialized. The Keyboard interface provides operations to register and unregister subsystems which want to receive keyboard data. If an interrupt occurs (a key is pressed), the appropriated message is forwarded to the class ConsoleManager (see below) which decides which subsystem gets the input.

The messages of the KeyboardManager contain only scancodes (a key number), sent by the keyboard controller, and status bytes. Subsystems have to translate scancodes into ASCII numbers themselves (LINUX does it itself, trusted subsystems can use the utility class KeyboardStub, see Section 5.1.6.3).

#### 5.1.8.3 Class: **ConsoleManager**

It is the ConsoleManager's task to synchronize user input and output by controlling the KeyboardManager and the DisplayManager and to enforce user authentication. It implements the TextConsole interface which provides methods to redirect user input and output to another task and to unlock the display:

If LINUX boots it demands access to the display and keyboard by accessing the Keyboard and TextConsole interface. These calls are forwarded to the ConsoleManager which stores the ID of the Linux thread which receives the keyboard data and switches to the LINUX screen (for debugging purposes). In contrast keyboard messages are forwarded to a trusted subsystem which implements the Authentication interface (see Section 5.1.9.4). If authentication was successful the console is unlocked and input and output is redirected to the LINUX kernel.

For debugging purposes hotkeys (`<Ctrl>+<Shift>+[<F1>-<F4>]`) are provided to switch to another virtual console. This makes it possible to view debugging output which is redirected by the microkernel to the first console. Switching to another console is only possible after authentication.

If trusted subsystems demand access, the ConsoleManager first copies the current screen and switches to console four. This is done to prevent that LINUX applications overwrite the output of the trusted subsystem and to be able to restore the screen's state after the trusted subsystem releases the console.

Because of dependencies between KeyboardManager, DisplayManager and ConsoleManager all three threads are provided by one subsystem UserInterface[4]. Figure 5.3 illustrates the static model of the UserInterface subsystem; untrusted applications access the Keyboard and TextConsole interface, trusted components are able to access the Console interface.

[4]The current design of these three threads/interfaces/classes seems to be too complicated. The reason for this is that dependencies between keyboard input and console output depend on each other which was not expected when I started to implement these classes. Because it works for my purposes I decided not to modify them.

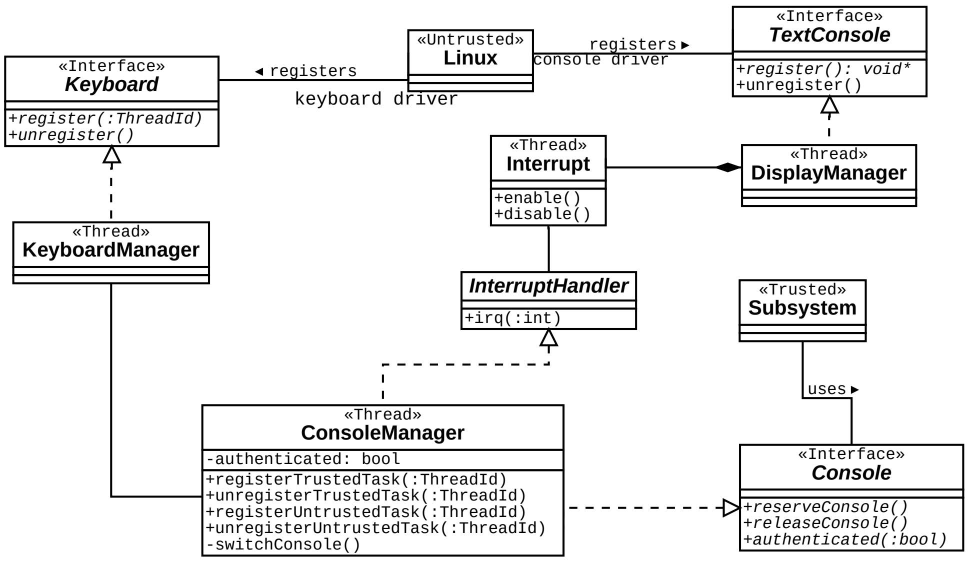

Figure 5.3: Static model of the UserInterface subsystem.

### 5.1.9 Package: **TrustedPath**

#### 5.1.9.1 A Virtual Linux Console

The file `vcon.c` (virtual console) works as a replacement of the i386/LINUX standard console implementation `vgacon.c`. The file distributed with the source package can only be loaded as a module (for test purposes); to ensure that LINUX uses the new console driver on startup, some further patches are necessary and the file has to be copied into the LINUX source tree.

Information about programming of the VGA video adapter have been extracted from [46], unfortunately I found no documentation about the boot procedure of LINUX and especially about initialization of the console drivers. Table 5.1 contains a list of related files of the LINUX kernel.

| File | Modified |
|---|---|
| `include/linux/console.h` | yes |
| `drivers/video/vcon.c` | new |
| `drivers/video/vgacon.c` | no |
| `drivers/char/console.c` | no |
| `drivers/char/tty_io.c` | no |
| `arch/l4-i386/kernel/setup.c` | yes |
| `init/main.c` | no |

Table 5.1: Important LINUX kernel files which initialize the console driver(s).

#### 5.1.9.2 A Virtual Linux Keyboard Driver

To adapt LINUX to the Keyboard interface the file `vkeyb.c` contains appropriate wrapper functions which replace the default ones. Additionally the file `include/asm-l4-i386/kernel/keyboard.h` has to be modified to force the keyboard driver (`drivers/char/keyboard.c`) to use the new functions.

Keyboard status LEDs, which are usually modified by a bottom half[5] of the keyboard interrupt handler, are currently not updated, because doing IPC on a bottom half seems to disturb LINUX (it produces only error messages “Scheduling on Interrupt”).

#### 5.1.9.3 Package: **TrustedGUI**

To make is possible to trusted subsystems to interact with users directly a simple GUI has to be implemented. Results of Section 5.1.8 are refined and a general GUI interface is presented.

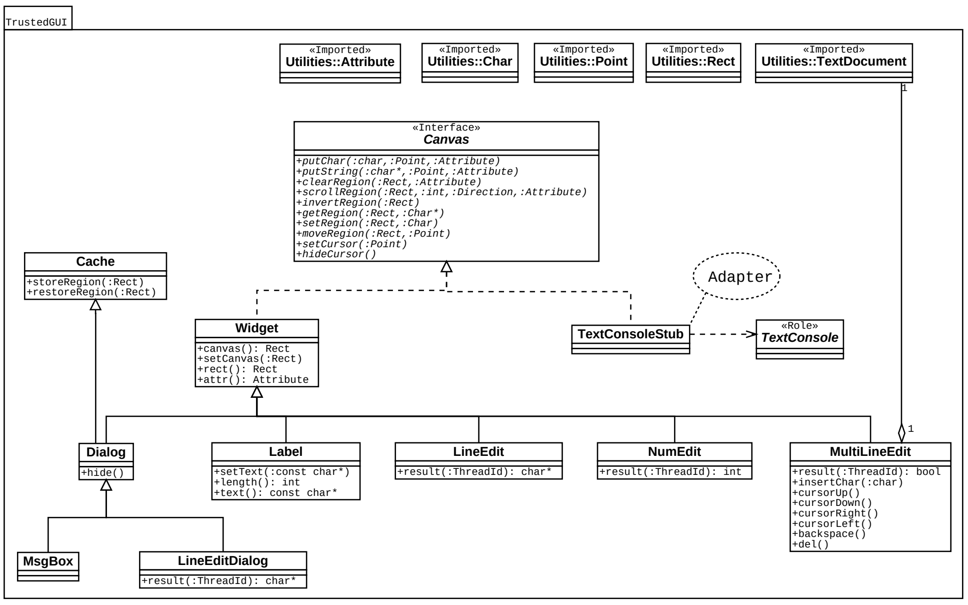

Figure 5.4: Components of the TrustedGUI package.

To keep it independent of the UserInterface implementation, an abstract interface to textmode output devices is provided by the abstract class Canvas (see Figure 5.4). The provided interface is very similar to the interface between text console driver and video driver used by the LINUX kernel. The class TextConsoleStub implements this interface and adapts them to the interface provided by the TextConsole interface.

Another class which implements the Canvas interface is the class Widget which acts as the base class of all GUI widgets. Therefore all widgets act as a canvas of their children and the TextConsoleStub acts as the root canvas. The class Widget extents the interface Canvas in such a way that it clears the background using a user-defined color and provides methods to resize and move the canvas it provides to its children. This feature is used, e.g. by the class Dialog which is derived from Widget and draws a frame and a headline. To prevent that children overwrite the frame, the canvas is moved and its size is decreased. Additionally the class Dialog is derived from the class Cache which stores its background to an internal buffer on construction and restores it on destruction. So all dialogs restore the region they have overwritten explicitly if the method `hide()` is invoked, or implicitly on destruction.

The class Label is derived from Widget and draws a static text which can be aligned horizontally and vertically. LineEdit draws a one-lined edit field and provides

[5]Linux divides interrupt handler into top halves, which are executed immediately after an interrupt and register appropriated bootom halves, which are executed as soon as possible.

simple editor functions to type a line of text. Very similar is the class NumEdit which accepts only numeric values and converts them into an integer value.

The class MultiLineEdit acts as a simple editor which uses the imported class TextDocument to store its content. If the shown document is larger than the MultiLineEdit's size, it is possible to scroll its content horizontally and vertically. The widget is activated by invoking the method `result()` and two return values are currently supported:

`True` is returned if the user leaves the method using the <TAB> key.

`False` is returned if the user leaves the method using the <ESC> key.

Three classes are derived from Dialog. The class MsgBox provides an easy way to show simple messages. The class LineEditDialog draws a dialog box which contains a static label and a line edit dialog. It is possible to define whether the dialog shows the letters typed by the user, or if only a placeholder is shown. This function is used, e.g. to create a password dialog box.

#### 5.1.9.4 Package: **UserAuthentication**

This subsystem implements the Authentication interface. After initialization it waits for a keyboard messages forwarded by the ConsoleManager. If such kind of message occurs it locks the display by invoking the ConsoleManager. Then a login dialog box and a password dialog box is shown and received values are compared with internal values. At least it unlocks the display and notices the ConsoleManager to forward keyboard messages to the LINUX kernel if authentication was successfull. Else it waits for the next keyboard message to restart the authentication process.

### 5.1.10 Package: **CommunicationPath**

The directory `CommunicationPath/Router` contains a simple router implementation according to Section 4.8.1. The presented version only accepts registration of new network devices and sends back received IP packets by modifying the header in such a way that `ping` packets are accepted by the sender as replies.

The subdirectory `VirtualNetworkDevice` contains a LINUX module which registers a new network device `vnd` (virtual network device) to the LINUX IP stack and to the Router. To be able to use the device, it has to be opened and registered. Example 5.1.10.1 loads the module, assigns IP address 192.168.96.6 to it and invokes LINUX to forward packets which have the destination address 192.168.97.1 to it. If you ping to this device you should get a reply.

#### 5.1.10.1 Example: **Opening and registering the vnd network device**

The following LINUX network commands load and initialise the virtual network module vnd.o.

```
# insmod vnd.o
# ifconfig vnd 192.168.96.6
# route add -host 192.168.97.1 vnd
# ping 192.168.97.1
```

### 5.1.11 Package: **KeyManagement**

The directory `KeyManagement` contains a subsystem PGP_Certificate which provides the interface Certificate (`Interfaces/Certificate.h`). It is similar to the requirements defined in Section 3.2.1. Additionally the key generation function, analyzed in Section 3.2.2, is supported internally. An instance of the subsystem represents exactly one key which can be created [F 15] and used to sign [F 65] an ASCII document. To prevent that internal data has to be sent to another subsystem it contains all internal functions demanded by Section 3.2.1. Also access control is enforced by the subsystem which forces users to enter a password given by the creator of the key. A key management service [F 10] is not provided because the naming service can be used to delegate access to different Certificate instances.

The subdirectory `Client` contains a LINUX application `seccert` (explained in Section 5.2.2.3) which makes it possible to use this subsystem.

Cryptographic functions [F 15] and [F 65] are currently only dummy implementations which show a simple GUI, but it should not be a problem to replace them by other ones.

### 5.1.12 OOP Model

Figure 5.5 gives an overview about components which have been implemented so far. Because the RMGR provided by the FIASCO distribution does not support virtual memory, all modules loaded on startup have to be loaded to another physical address (see Section 5.1.1). Table 5.2 lists hardcoded start addresses and the length of all subsystems started by the RMGR.

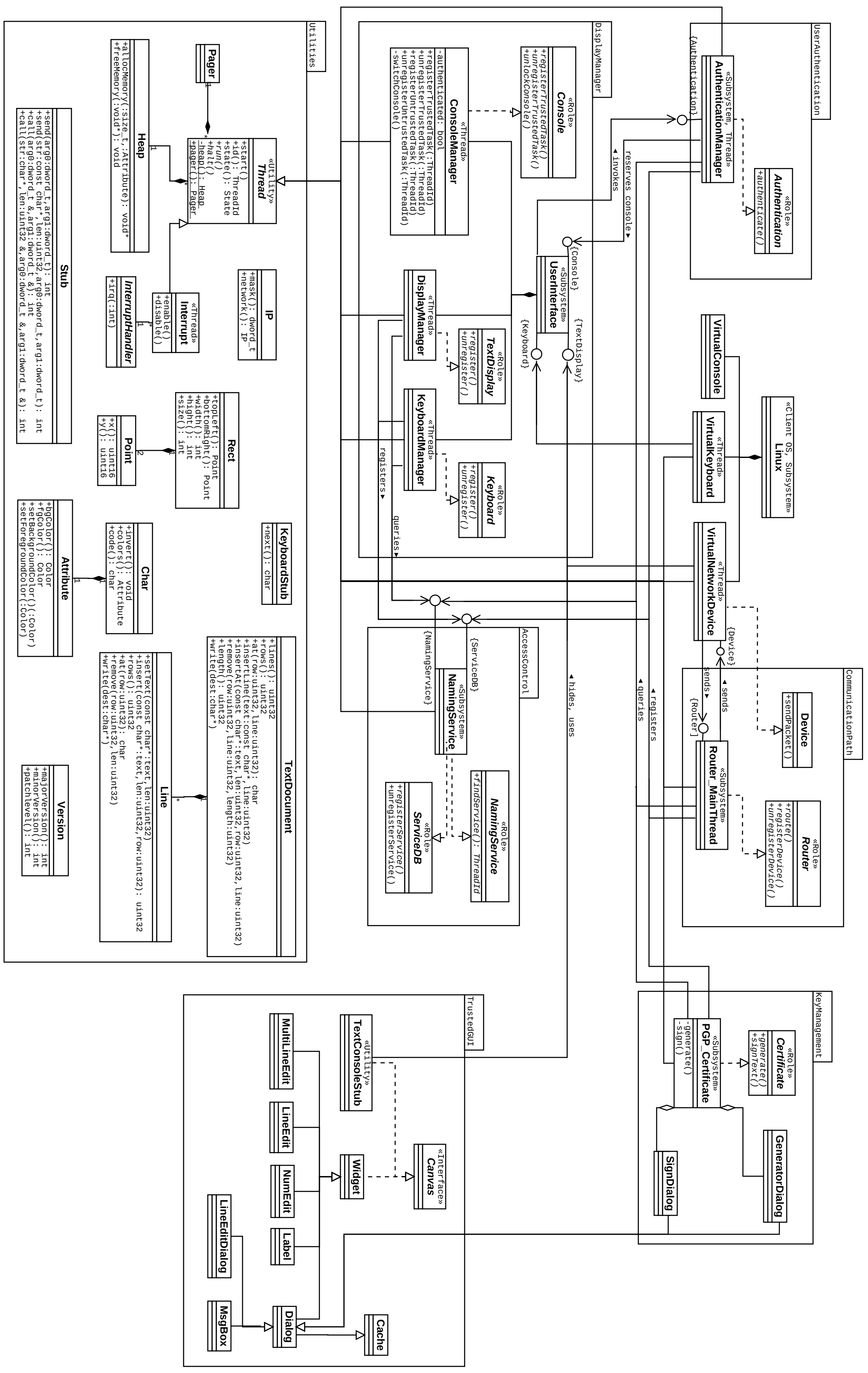

Figure 5.5: Components implemented so far.

| **Subsystem** | **Address** | **Length** (byte) |
|---|---|---|
| Sigma0 (Fiasco) | `0x0080000` | 16096 |
| NamingService | `0x0300000` | 6160 |
| Linux (gzipped) | `0x03ff000` | 435673 |
| Router | `0x0800000` | 254336 |
| Certificate | `0x0850000` | 271852 |
| Authentication | `0x0900000` | 263392 |
| UserInterface | `0x0950000` | 259484 |
| RMGR (Fiasco) | `0x1000000` | 154852 |

Table 5.2: Hardcoded start addresses and length of all implemented subsystems.

## 5.2 Instead of a User's Manual

This sections contains user-specific information which should later become a separated UserManual.

### 5.2.1 Installation

It is assumed that an existing Linux installation exists and that the GNU gcc compiler is installed.

The package `PERSEUS-<version>.tar.gz` contains all sources of this diploma thesis and additionally binary versions of components of third parties[6]. Unpack the package and go to the main directory `PERSEUS-<version>`. The file `s./INSTALL` and `./README` contain general configuration and installation instructions. Read them carefully!

Then start the configuration script by typing `'./configure [your options]'`. The installation directory given by the option "`--prefix=`" is hence called <PATH>. If configuration has finished successfully, you can start compilation by typing `'make'`. To copy all services to the defined installation directory type `'make install'`.

#### 5.2.1.1 Installation of the GRUB bootloader

To start Fiasco a multiboot compliant bootloader is required. This subsection describes required steps to install the GRUB bootloader which is contained in the distribution.

This installation process overwrites the existing boot sector of your harddisk, thus you should make a copy of it (better backup the whole installation) and/or generate a bootdisk to be able to reboot the system if something goes wrong!

To install the GRUB bootloader, change to the directory `src/Boot` and edit the file `menu.lst` to adapt appropriate directory paths to your installation path (given by the `--prefix` option of the configure script). Don't forget to adapt the entries which boot your existing Linux installation. Then copy the files `menu.lst, stage0` and `stage1` to the directory `/boot/grub`. Now start GRUB by typing `'./grub'`. The following lines explain an example installation, but keep in mind that your partition table may look different:

[6]The L4 port of Linux, the GRUB bootloader, the Fiasco $\mu$-kernel, the resourcemanager and the sigma0 server.

```
    grub> root (hd0,2)
    grub> setup (hd0)
    grub> install <PATH>/boot/grub/stage1 (hd0) <PATH>/boot/grub/stage2
 p <PATH>/boot/grub/menu.lst
    grub> quit
```

This example expects that the first disk (`hd0`) is your boot disk and that LINUX's root filesystem is on partition `/dev/hda3` (!). If you're unsure, download the GRUB package or look for additional information at `www.gnu.org/grub`.

Now reboot the system and hope that it works.

### 5.2.2 Using

#### 5.2.2.1 Booting

Use the GRUB bootloader to load the following components using the following GRUB configuration

```
kernel = <PATH>/services/rmgr -configfile -sigma0
module = <PATH>/services/fiasco -nokdb -nojdb
module = <PATH>/services/sigma0
module = <PATH>/services/rmgr.cfg.sure
module = <PATH>/services/NamingService
module = <PATH>/services/Authentication
module = <PATH>/services/Certificate
module = <PATH>/services/UserInterface
module = <PATH>/services/glinux.gz root=/dev/hda3
```

and ensure that the RMGR configuration file contains a line

```
memory in [0x0, 0x01000000]
```

to force LINUX to use only lower memory regions. If the RMGR configuration file contains an entry bootwait you have to hit <Return> to start the RMGR. After a little while you should be able to see LINUX's boot output and a red line at the top of the window, hence called headline, which tells you which task you actually see. The color red indicates that the task is untrusted. If the color is green you are using a trusted task.

#### 5.2.2.2 Authentication

If LINUX has finished booting and you press a key, the headline should become green and you are requested by a dialog to enter a login name. Currently only "`stueble`" is a valid login name. If you hit <Return> you have to enter a password. Type "`sure`" and hit <Return> again. If authentication was successfull, a green message box is shown for a short time interval and LINUX gets the keyboard and display focus (the headline becomes red).

If authentication failed a red alert box is shown for a longer (than the green message box) time interval. Then you can retry authentication.

#### 5.2.2.3 Using Linux and a trusted application

If authentication was successful you can use LINUX as you are used to (graphic mode does currently not work). Try to use the virtual network device as explained in Section 5.1.10.

To create a new certificate securely you have to compile and install the program `seccert` which can be found in the directory `KeyManagement/Client`.

If you enter "`seccert -k`" a trusted application should get the focus which shows a small dialog to select attributes and a password of the certificate to be created.

To sign a text document type "`seccert -s <filename>`"! The trusted certificate application gets the focus, shows the task-id of the task which invokes it and starts a simple editor to view/edit the document to be signed. <ESC> aborts the trusted program. If you type <TAB> to continue you have to enter the password selected while key creation. If the password is correct, the file `<filename>.signed` should contain the signed document.

## 5.3 Conclusion and Future Work

The first part of this work has examined basic requirements of a secure kernel which acts in parallel to and protected from an existing operating system. Results of these analysis phase have been used to illustrate existing problems to demand security related properties of underlying components and to design a first model of services which have to be provided by the secure environment. A first working implementation based on the FIASCO $\mu$-kernel and the operating system LINUX is presented which contains all required components to securely sign documents which have been created under LINUX.

The next development steps should aim to modify the init file `crt0.S` in such a way that static data is initialized, because non-initialization of static data is error prone and makes the code more complicated than necessary. Further improvements should provide a system-wide memory pager which supports virtual memory and has an interface which considers memory attributes like integrity or confidentiality. Additionally development of subsystems would be easier if the RMGR would be able to load them to virtual addresses to prevent overlappings.

To be able to develop an ACEF the SubsystemManagement package is required to make it possible for ACEF instances to start new subsystems on their own. The packages TrustedGUI and ResourceManagement should be extended by an event-handler which replaces the currently available Keyboard and Console interfaces. The EventHandler should provide a secure interface to services in need of non-deterministic data input, as for example random generators.

Before the secure environment is ported to a small-sized PDA, a secure shared library concept should be made available to save physical memory.

# Appendix A

# System Requirements Specification

## A.1 Purpose

A system has to be developed which provides a secure environment for security- and privacy-critical applications of e-business and related legal and social interaction. In parallel an existing operating system (*Client OS*) shall provide users and developers a usual environment. Functions and services of the secure environment shall be accessible through well-defined interfaces which guarantee enforcement of the security policy of the secure environment. Users shall be able to verify the correctness and security of the software and, if demanded, be able to exchange individual components.

## A.2 Functions

The secure environment shall contain services to securely generate, manage and use cryptographic keys of at least one common cryptographic data exchange format (e.g. OpenPGP or S/MIME). Additionally a trusted document viewer or a trusted document editor (optional) and related security-management functions shall be provided.

In order to be able to provide these abstract functions all security-related underlying components, including access control and resource management, should be designed and implemented in such a way that later evaluation is possible.

## A.3 Properties

The secure environment should be able to protect user and system data against malicious applications (viruses and trojan horses) and similar sorts of internal and external attacks by enforcing its own security policy, independent of security policies of the *Client OS*. Meaningful default values shall be provided which guarantee security without special user knowledge about security and cryptography, but experienced users shall be able to overwrite default values and/or define their own security policy. Later evaluations according to common criteria (e.g. ITSEC, Common Criteria) should be considered by the design model and the implementation.

### A.3.1 Protection Profile

[Not provided by this diploma thesis].

## A.4 Tests

The secure environment should be tested by developing some secure applications:

1. A signature application which provides functions to securely sign ASCII documents by *Client OS* applications.

2. A secure stock broking application (optional).

# Appendix B

# System Definition

## B.1 Purpose

Develop a secure environment for security- and privacy-critical applications of e-business and related legal and social interaction which provides confidentiality and integrity to protect user and system data against external and internal attacks.

## B.2 System Environment

### Software

- Fiasco microkernel
- L4-Linux as *Client OS*.
- EPOC as *Client OS* of a PDA (optional).
- GRUB boot loader, version 0.5.94

### Hardware

- IBM Thinkpad 600E, Intel Pentium II 366 MHz (default)
- Dell Latitude XPi, Intel Pentium I 133 MHz
- PDA (optional)

## B.3 Security Requirements

### B.3.1 Security Target(s)

To be derived from *Protection Profiles* listed in Appendix A.
[Not provided by this diploma thesis]

### B.3.2 Cryptographic Support

[SR1] The TSF shall be able to generate cryptographic keys in accordance with a specified cryptographic key generation algorithm and specified cryptographic key sizes that meet specified standards.

[SR10] The TSF shall distribute cryptographic keys in accordance with a specified cryptographic key distribution method that meets specified standards.

[SR15] The TSF shall be able to perform cryptographic key access in accordance with a specified cryptographic key access method that meets specified standards.

[SR20] The TSF shall destroy cryptographic keys in accordance with a specified cryptographic key destruction method that meets specified standards.

[SR40] The TSF shall be able to perform cryptographic operations in accordance with a specified cryptographic algorithm and cryptographic key size that meet specified standards.

### B.3.3 User data protection

[SR60] The reference monitor has to be able to cover all upcoming operations between all objects and subjects which are not fully trusted.

[SR65] The reference monitor has to be able to support several access control policies in parallel.

[SR80] The TSF shall enforce related SFPs when exporting user data , controlled under the SFPs, outside the TSC.

[SR81] The TSF shall be able to export user data without the user data's associated security attributes.

[SR85] The TSF shall be able to enforce all access and data flow control policies when importing user data, controlled under the SFP, from outside the TSC.

[SR86] The TSF shall be able to use security attributes associated with the imported user data.

[SR87] The TSF shall be able to ensure that the protocol used provides for the unambiguous association between the security attributes and the user data received.

[SR88] The TSF shall be able to ensure that interpretation of security attributes of imported user data is as intended by the source of the user data.

[SR100] The TSF shall be able to ensure that any previous information content of a resource is made unavailable upon the deallocation of the resource.

[SR105] The TSF shall be able to enforce all access and information flow control SFP(s) to permit the rollback of all operations on all objects.

[SR106] The TSF shall permit operations to be rolled back within a *configurable time* interval.

[SR110] The TSF shall be able to monitor user data within the TSC for *software bugs, memory errors* on all objects.

[SR111] Upon detection of a data integrity error, the TSF shall *rollback into the last coherent state.*

[SR115] The TSF shall be able to enforce all access and information flow control SFP(s) to be able to *transmit and receive* objects in a manner protected from unauthorized disclosure.

[SR120] The TSF shall be able to enforce all access and information flow control SFP(s) to be able to *transmit and receive* user data in a manner protected from *modification, deletion, insertion and replay* errors.

[SR121] The TSF shall be able to determine on receipt of user data, wether *modification, deletion, insertion and replay* has occurred.

### B.3.4 Identification and Authentication

[SR135] The TSF shall be able to maintain security attributes to individual users.

[SR143] The TSF shall allow *only user-authentication* on behalf of the user to be performed before the user is authenticated.

[SR150] The TSF shall provide *smartcard-based* and *password-based* authentication mechanisms to support user authentication.

[SR151] *The security admin has to be authenticated using the smartcard-based mechanism; the admin and the user can be authenticated using the password-based mechanism.*

[SR155] The TSF shall re-authenticate the user under the following conditions: *changes of assigned roles and after a configurable time-interval of user inactivity.*

[SR156] The TSF shall provide *only the number of characters typed or the characters typed* to the user while authentication is in progress.

[SR160] The TSF shall require each user to identify itself before allowing *any other* TSF-mediated actions on behalf of that user.

[SR165] The TSF shall associate the appropriate user security attributes with subjects acting on behalf of the user.

### B.3.5 Security management

[SR170] The TSF shall restrict the ability to invoke security-related functions to the *security-admin* role.

[SR171] The TSF shall restrict the ability to invoke non security-related functions which globally change the system behaviour to the *admin* role.

[SR180] The TSF shall enforce the access and information flow control SFP(s) to restrict the ability to modify critical security attributes to the *security-admin* role.

[SR181] The TSF shall enforce the access and information flow control SFP(s) to restrict the ability to modify non-critical security attributes to the *admin* role.

[SR185] The TSF shall ensure that only secure values are accepted for security attributes.

[SR190] The TSF shall restrict the ability to change security-related data to the *security-admin* role.

[SR191] The TSF shall restrict the ability to change system-wide data to the *admin* role.

[SR195] The TSF shall restrict the specification of the limits for security-related data to the *security-admin* role.

[SR196] The TSF shall restrict the specification of the limits for system-wide data to the *admin* role.

[SR198] The TSF shall be able to take actions, if the TSF data are at, or exceed indicated limits.

[SR200] The TSF shall ensure that only secure values are accepted for the TSF data.

[SR205] The TSF shall restrict the ability to revoke system wide security attributes within the TSC to the *security-admin* role.

[SR206] The TSF shall be able to enforce the revocation rules.

[SR208] The TSF shall be able to restrict the capability to specify an expiration time for security attributes to the *security-admin* role.

[SR209] For each of this security attributes, the TSF shall be able to take some actions after the expiration time for the indicated security attribute has passed.

[SR210] The TSF shall be able to maintain roles.

[SR211] The TSF shall be able to associate users with roles.

[SR215] The TSF shall be able to require an explicit request to assume roles.

## B.3.6 Protection of the TSF

[SR250] The TSF shall be able to ensure the operation of *critical functions* when *software bugs of non-core components* occur.

[SR280] After a failure or service discontinuity, the TSF shall enter a maintenace mode where the ability to return the TOE to a secure state is provided.

[SR281] The TSF shall be able to ensure that failure scenarios have the property to that the SF either completes successfully, or for the indicated failure scenarios, recovers to a consistent and secure state.

[SR285] The TSF shall be able to detect replay for a pre-defined list of entities.

[SR286] The TSF shall be able to perform actions when replay is detected.

[SR290] The TSF shall ensure that TSP enforcement functions are invoked and succeed before each function within the TSC is allowed to proceed.

[SR295] The unisolated portion of the TSF shall maintain a security domain for its own execution that protects it from interferences and tampering by untrusted subjects.

[SR296] The TSF shall enforce separation between the security domains of the subjects in the TSC.

[SR297] The TSF shall maintain the part of the TSF that enforces the access control and/or information flow control SFPs in a security domain for its own execution that protects them from interferences and tampering by the remainder of the TSF and by subjects untrusted with respect to the TSF.

[SR298] *Layers below the red line (Figure 1.1) have to protect security domains against all other accesses which are not covered by the access control mechanisms.*

[SR299] Subsystems should be able to refine system-wide or enforce local security policies.

[SR300] The TSF shall be able provide reliable time stamps for its own use.

[SR310] The TSF shall ensure that TSF data is consistent when replicated between parts of the TOE.

[SR311] When parts of the TOE containing replicated TSF data are disconnected, the TSF shall be able to ensure the consistency of the replicated TSF data upon reconnection before processing any requests.

### B.3.7 Resource allocation

[SR310] The TSF shall be able to ensure the operation of core components when the following errors occur: *software failure of non-core components.*

[SR315] The TSF shall mediate *all unshareable resources by trusted components.*

[SR320] The TSF shall assign a priority to each subject in the TSF.

[SR321] The TSF shall ensure that each access to *unshareable resources* is mediated on the basis of the subjects assigned priority.

[SR325] The TSF shall enforce maximum quotas of *all unshareable resources.*

### B.3.8 TOE access

[SR340] The TSF shall be able to lock an interactive session after a specified **time interval of user inactivity** by:

1. clearing or overwrite display devices, making the current contents unreadable.
2. disabling any activity of the user's data access/display devices other than unlocking the session.

[SR341] The TSF shall be able to require events (e.g. re-authentication) to occur prior to unlocking the sessions.

[SR345] The TSF shall allow user-initiated locking of the user's own interactive session, by:

1. clearing or overwrite display devices, making the current contents unreadable.
2. disabling any activity of the user's data access/display devices other than unlocking the session.

[SR346] The TSF shall be able to require events (e.g. re-authentication) to occur prior to unlocking the sessions.

### B.3.9 Trusted path/channels

[SR350] The TSF shall provide a communication channel between itself and a remote trusted IT product that is logically distinct from other communication channels and provides assured identification of its end points and protection of the channel data from modification or disclosure.

[SR351] The TSF shall be able to initiate a communication to another trusted IT product via the trusted channel.

[SR352] The TSF shall initiate communication via the trusted channel for *download of new content.*

[SR355] The TSF shall provide a communication path between itself and *local* users that is logically distinct from other communication paths and provides assured identification of its end points and protection of the communicated data from modification or disclosure.

[SR356] The TSF shall permit *the TSF and local users* to initiate communication via the trusted path.

[SR357] The TSF shall require the use of the trusted path for *initial user authentication and every communication between user and secure environment.*

### B.3.10 Miscellaneous

[SR400] Subsystems should only be able to access other subsystems indirectly.

[SR405] References to other subsystems should only be locally valid.

## B.4 System Functions

### External functions

[F3] Communication channel between Client OS applications and secure platform.

[F5] A sign function to the Client OS which accepts a plaintext and a key-id as arguments.

[F15] A key generation service with no arguments which provides abstract security levels and key identifiers in a user-readable format.

[F30] A function to extract a public key.

[F40] A function to start a new subsystem, which accepts a protocol definition and a destination as argument.

[F45] A function to remove an already installed subsystem. The argument should be a reference to the subsystem.

[F75] A naming service which maps external subsystem references into an internal data type and vice versa.

[F85] A function to open a new session which authenticates users before opening a session.

[F90] A function to close the current session.

[F95] A function to rollback the system state.

### Internal functions

| | |
|---|---|
| [F10] | A crypto-standard independent key management service. |
| [F20] | A random bit generator service which xors different random streams. |
| [F25] | A random generator which generates random bits from user input. |
| [F35] | A service to copy external data into the secure environment. |
| [F50] | A service which stores dependencies between installed subsystems. |
| [F52] | A function to parse data and then execute it as a new subsystem. |
| [F55] | A service helps users to define permissions of a new subsystem. |
| [F60] | A hash function which matches a defined crypto standard. |
| [F65] | A sign function which matches a defined crypto standard. |
| [F70] | A function to convert ASCII text into a binary format which matches a defined standard. |
| [F71] | A function to convert binary data into ASCII text which matches a defined standard. |
| [F78] | A version scheme to distinguish interface versions. |
| [F80] | A function to store the complete system state to a persistent storage. This function has to ensure that the state of the current session is stored. |
| [F81] | A function to restore the system state from data stored by [F80]. It has to ensure that after loading into the latest state a authentication mechanism is invoked. |
| [F95] | A function which returns the current time. |
| [F100] | The secure environment should provide its own trusted GUI toolkit. |
| [F105] | A document viewer which can display documents in a well-known and system-independent format. |
| [F110] | A function which makes security attributes assigned to subjects acting on behalf of users available to subsystems accessed by these subjects. Required to enforce local security policies or refine them. |
| [F115] | It should be possible for permitted subsystems to synchronize their local references, required by [SR 400]. |
| [F120] | Local references, required by [SR 405], of subsystems of incoming and outgoing messages should be synchronized. |

## B.5 System Data

| | |
|---|---|
| [D10] | A users secret key. |
| [D20] | A complete copy of the latest consistent system state (optional). |
| [D30] | An application-info describes the demanded permissions of an application. |
| [D40] | An application-certificate defines the permissions of an application. |
| [D30] | Dependencies between subsystems. |

## B.6 User friendlyness and flexibility

[P5] An existing operation system makes existing applications available to the user.

[P10] Emergency PIN (optional)

[P15] The user is able to choose between security and usability.

[P20] Meaningful default values.

## B.7 Quality Demands

| **Quality** | **very good** | **good** | **normal** | **not relevant** |
|---|---|---|---|---|
| Functionality | | | x | |
| Security | x | | | |
| Reliability | x | | | |
| Useability | | x | | |
| Efficiency | | | | x |
| Changeability | | x | | |

## B.8 Test scenarios

The secure environment should be tested by developing some secure applications:

1. A signature application which provides functions to securely sign ASCII documents by Client OS applications.
2. A secure stock broking application.

## B.9 Development Environment

Using vmware version 1.0 does not work because the GRUB boot loader does not run. Version 2.0 should be tested because of shorter development circles.

Table B.1 lists types and components of used development platforms. Software

| **PC** | **Processor** | **Operating System** | **Environment** |
|---|---|---|---|
| IBM Thinkpad 600E | Intel Pentium II 366 MHz | Linux 2.2.14 | SuSE 6.3 |
| Dell Latitude XPi | Intel Pentium 133 MHz | Linux 2.2.13 | SuSE 6.3 |
| No Name | Intel Celeron 466 MHz | Linux 2.2.14 | SuSE 6.3 |

Table B.1: Hardware used as develop environment.

components and versions of the development environment are listed by Table B.2.

| Software | Version |
|---|---:|
| GNU gcc | 2.7.2.3 |
| GNU automake | 1.4 |
| GNU autoconf | 2.13 |
| GRUB bootloader | 0.5.94 |
| doxygen | 0.49-991003 |

Table B.2: Software used for development.

# Appendix C

# Abbreviations

| | |
|---|---|
| ACDF | Access Control Decision Facility |
| ACEF | Access Control Enforcement Facility |
| ACL | Access Control List |
| API | Application Programming Interface |
| BIOS | Basic Input-/Output System |
| CC | Common Criteria |
| CPL | Current Privilege Level |
| DFD | Data Flow Diagram |
| DMA | Direct Memory Access |
| IOPBM | Input/Output Permission Bitmap |
| IOPL | Input/Output Privilege Level |
| IPC | Inter Process Communication |
| OO | Object Oriented |
| OOA | Object-Oriented Analysis |
| OOD | Object-Oriented Design |
| OOP | Object-Oriented Programming |
| OS | Operating System |
| PCS | Program Control Structure |
| PDA | Personal Digital Assistant |
| PP | Protection Profile |
| RBAC | Role-Based Access Control |
| RMGR | Resource Manager |
| RM | Reference Monitor |
| SE | Secure Environment |

| | |
|---|---|
| SF | Security Function |
| SFP | Security Function Policy |
| SRS | System Requirements Specification |
| ST | Security Target |
| STL | Standard Template Library |
| TID | Thread Identifier |
| TOE | Target of Evaluation |
| TSC | TOE Scope of Control |
| TSF | TOE Security Functions |
| TSP | TOE Security Policy |
| TSS | Task State Segment |
| UML | Unified Modeling Language |

# Appendix D

# Glossary

***Access Control Mechanism*** →*Reference Monitor*

***Access Control Model*** A definition of syntax of rules to express →*Access Control Policies.*

***Access Control Policy*** Rules which determine how accesses are controlled and access decisions determined.

***Address Space*** A mapping which associates each virtual page to a physical page frame or marks it non-accessible.

***Analysis Pattern*** A →*Pattern* which provides solutions of frequent analysis problems.

***Atomic Operation*** An operation which is finished successfully or aborted completely.

***Automatic Memory*** Memory which is allocated on the task's stack if the program counter enters its range of validity, and released if the program counter leaves it.

***Client OS*** An operating system which is running as a subsystem of the →*Fiasco* →*Microkernel* and provides a familiar environment to users and developers.

***Common Criteria*** Common Criteria for Information Technology Security Evaluation, ISO/IEC 15408.

***Design Pattern*** A →*Pattern* which provides solutions of frequent design problems.

***Dynamic Memory*** Memory which is allocated and released only by explicit functions.

***Fiasco*** A →*Microkernel* developed by the DROPS project of the University of Dresden.

***Global Memory*** Memory which is allocated and initialized before the `main()` function is entered. Can be accessed by all functions/classes of the same →*Address Space.*

***Heap Memory*** →*Dynamic Memory.*

***Implementation Pattern*** A →*Pattern* which provides solutions of frequent programming problems.

***Message Redirection Mechanism*** A concept of an abstract machine which redirects messages sent between →*Subsystems* to a defined Subsystem.

***Microkernel*** A minimized operating system kernel which only provides an abstract view of the hardware.

***Naming Service*** *A* service which maps →***Service*** descriptions to →***Subsystem*** identifiers.

***Object-Oriented Analysis*** The phase of object-oriented development which determines and describes requirements of the product to be developed and which forms the conceptual solution by an OOA model.

***Object-Oriented Design*** The phase of object-oriented development which realizes the OOA model with respect to constraints of the real world. The result is an OOD model.

***Object-Oriented Implementation*** The phase of object-oriented development which realizes the OOD model using a specific programming language.

***Object-Oriented Programming*** →*Object-Oriented Implementation.*

***Pattern*** Describes classes of problems and provides general solutions of the problem which can often be used.

***Persistence*** A persistent object stores its state, even if the environment (computer, operating system, program) is not active.

***Protected Domain*** The smallest execution unit that can be protected by the →*Microkernel* or the underlying hardware.

***Protection Profile*** An implementation-independent set of security requirements for a category of →*TargetsOf Evaluation* that meet specific consumer needs.

***Reference Monitor*** The concept of an abstract machine which enforces →*Access Control Policies.*

***Role*** A predefined set of rules establishing allowed interactions between users and →*Targets Of Evaluation.*

***Secure Environment*** All security-related layers which are required to provide a trusted path between secure application and user.

***Security Function*** A part or parts of the →*Target Of Evaluation* that have been relied upon for enforcing a closely related subset of the rules from the →*TOE Security Policy.*

***Security Function Policy*** The Security Policy enforced by a →*Security Function.*

***Security Target*** A set of security requirements and specifications to be used as the basis for evaluation of an identified →*Target Of Evaluation.*

***Session*** The time interval between user log-in (opening a session) and log-out (closing a session). Accessing subsystems is only possible if users have opened a session before.

***Service*** A →*Subsystem* which provides functions to other subsystems.

***Static Memory*** Memory which is allocated before the function containing the static data is entered. In C++ static data members are allocates like global memory, before the `main()` function is invoked.

***Subject*** An entity within the →*Target of Evaluation* that causes operations to be performed.

***Subsystem*** →*Protected Domain*

***System Definition*** Describes all required functions and a conceptual solution of an IT product.

***System Requirements Specification*** Describes required functions of an IT product and the constraints under which it must operate. It shall be written in such a way that it is understandable by customers without special knowledge.

***Target Of Evaluation*** An IT product or system and its associated administrator and user guidance documentation that is the subject of an evaluation.

***TOE Security Functions*** A set consisting of all hardware, software, and firmware of the →*Target Of Evaluation* that must be relied upon for the correct enforcement of the →*TOE Security Policy.*

***TOE Security Policy*** A set of rules that regulate how assets are managed, protected and distributed within a →*Target Of Evaluation.*

***TSF Scope Of Control*** The set of interactions that can occur with or within a →*Target Of Evaluation* and are subject to the rules of the →*TOE Security Policy.*

***UML*** →*Unified Modeling Language*

***Unified Modeling Language*** A language to analyse and design object-oriented systems. It unifies object-oriented methods of Booch, OMT and OOSE and has been accepted by the OMG (Object Management Group) in 1997.

***Use Case*** "Specifies the behaviour of a system or a part of the system and is a description of a set of actions, including variants, that a system performs to yield an observable result of value to an actor.", [36].

***XServer*** An application which provides an abstraction of the video hardware and can be accessed by XClients via a specific protocol.

# Appendix E

# Contents of the CD-Rom

The main directory of the CD-Rom distributed with this diploma thesis contains different versions of this work, e.g. a postscript (`Stue00.ps`) and a PDF version (`Stue00.pdf`). A html version, generated by `latex2html`, can be found in the subdirectory `Stue00_html`.

The subdirectory `Stue00_src` contains all sources of this diploma thesis, e.g. Latex files, postscript figures and UML diagrams generated by `dia`.

Subdirectory `Documents` contains postscript/pdf documents which have been referenced by this work (if available online) and some additional information/papers. Read `Documents/index.html` for a more precise overview.

The directory `PERSEUS-0.1` contains the source code of the first prototype developed by this diploma thesis. It also contains a postscript, PDF and html version of the Reference Manual, generated by `doxygen`, in the subdirectory `doc/refman`.

# Bibliography

[1] Cryptographic Message Syntax. Internet Draft draft-ietf-smime-cms-*.txt.

[2] OpenPGP Message Formats. Internet Draft draft-ietf-openpgp-formats-*.txt.

[3] James L. Antonakos. *The Pentium Microprocessor.* Prentice Hall Inc., 1997.

[4] Heide Balzert. *Lehrbuch der Objektmodellierung.* Spektrum Verlag, 1999.

[5] Helmut Balzert. *Software-Entwicklung.* Lehrbuch der Software-Technik, Band 1. Spektrum Verlag, 1996.

[6] Michael Barabanov. *An Introduction to RT-Linux.* 1998.

[7] D.E. Bell and L.J. La Padula. *Secure Computer Systems.* Mitre Corperation, 1974.

[8] Joachim Biskup. *Grundlagen von Informationssystemen.* Vieweg Verlag, 1995.

[9] Joachim Biskup. *Sicherheit in Rechnersystemen: Fragen und Lösungsansätze.* Fachbereich Informatik, Universität Dortmund, 1998.

[10] G. Booch, J.Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[11] Brey. *The Intel i386 Microprocessor family.* 1996.

[12] Alan Cox. Network buffers and memory management. *Linux Journal*, 1996.

[13] DeMarco. *Structured Analysis and System Specification.* Englewood Cliffs: Yourdon Press, 1979.

[14] D.E. Denning. A lattice model of secure information flow. In *Communications of the ACM 20,7*, pages 504–513, 1977.

[15] Kevin Elphistone, Stephen Russel, and Gernot Heiser. Supporting persistent object systems in a single address space. Technical report, School of Computer Science and Engineering. The University of New South Wales, 1996.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, 1995.

[17] G. Heiser, K. Elphinstone, S. Russel, and G.R. Hellestrand. A distributed single address-space operating system supporting persistence. Technical Report SCS&E Report 9302, Computer and Systems Technology Laboratory, School of Computer Science and Engineering; The Univerity of New South Wales, Kensington NSW, Australia 2033, March 1993.

[18] G. Heiser, K. Elphinstone, S. Russel, and J. Vochteloo. Mungi: A distributed single address-space operating system. Technical report, Computer and Systems Technology Laboratory, School of Computer Science and Engineering; The Univerity of New South Wales, Kensington NSW, Australia 2033, 1994.

[19] Michael Hohmuth. The Fiasco Kernel, Requirements Definition. Technical Report ISSN 1430-211X, Dresden University of Technology, Dept. of Computer Science, December 1998.

[20] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. Technical report, Dresden University of Technology, Dept. of Computer Science, 1998.

[21] Hermann Härtig, Birgit Pfitzmann, James Riordan, and Michael Waidner. Necessity and preliminary architecture of a security platform for mobile devices. Private communication, 1999.

[22] Intel Corporation. *The 386 DX Microprocessor Programmer´s Reference Manual*, 1990.

[23] T. Jaeger, A. Prakash, J. Liedke, and N. Islam. Flexible control of downloaded executable content. In *Transactions on Information and System Security*, pages 177–228. ACM, May 1999.

[24] Gerard Lacoste, Birgit Pfitzmann, Michael Steiner, and Michael Waidner. Semper final report. Technical report, to appear in LNCS, Springer-Verlag, Berlin, 1999.

[25] Jochen Liedke, editor. *Clans and Chiefs. Architektur von Rechnersystemen*, 12. GI/ITG Fachtagung, Kiel, 1992.

[26] Jochen Liedke, editor. *A Persistent System in Real Use. Experiences of the first 13 Years*, International Workshop on Object-Orientation in Operating Systems, Asheville, North Carolina, December 1993.

[27] Jochen Liedke. *L4 Reference Manual.* GMD, 1996.

[28] Norbert Luckhardt. Eigentor - Ersatzschlüssel hebelt Exportkontrolle aus. *c't Magazin für Computertechnik*, (19):68, 1999.

[29] Norbert Luckhardt. Send it: Programmierer lasen e-mails mit. *c't Magazin für Computertechnik*, (3):64, 2000.

[30] Matthias Schunter and Christian Stüble. Effiziente Implementierung von kryptografischen Datenaustauschformaten am Beispiel OpenPGP und S/MIME v.3. In *Sicherheitsinfrastrukturen*, DuD Fachbeiträge, pages 272–284. Vieweg Verlag, 1999.

[31] Frank Mehnert. Portierung des SCSI Geräteftreibers von Linux auf L3. Beleg, TU Dresden, November 1996.

[32] Frank Mehnert. Ein zusagefähiges SCSI Subsystem für DROPS. Diplomarbeit, TU Dresden, Januar 1998.

[33] Scott Meyers. *Effective C++.* Addison Wesley, 2nd edition, 1997.

[34] Jens Nerche. Dynamisches Nachladen von Komponenten in DROPS. Großer beleg, Technische Universität Dresden, Mai 1999.

[35] B. Oestereich. *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language.* Oldenbourg-Verlag, Wien, 1997.

[36] OMG. *Unified Modeling Language Specification (draft)*, version 1.3 edition, March 1999. www.rational.com/uml.

[37] Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation (Version 2.1).* Common Criteria Project Sponsoring Organisations, August 1999. adopted by ISO/IEC as Draft International Standard DIS 15408 1-3.

[38] Amun Ott. Regelsatz-basierte Zugriffskontrolle nach dem 'Generalised Framework for Access Control' Ansatz am Beispiel Linux. Diplomarbeit, Universität Hamburg, November 1997.

[39] Alessandro Rubini. *Linux Device Drivers.* O'Reilly Verlag, 1998.

[40] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. In *IEEE Computer 98,2*, pages pp. 38–47, 1996.

[41] A. Silberschatz. *Operating System Concepts.* Addison Wesley, 5th edition, 1998.

[42] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent storage. In *Persistent Object Systems, San Miniato 1992*, pages 11–33. Springer-Verlag, 1992.

[43] Rene Stange. Systematische Übertragung von Gerätetreibern von einem monolithischen Betriebssystem auf eine mikrokernbasierte Architektur. Diplomarbeit, TU Dresden, 1996.

[44] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, 2 edition, 1995.

[45] Sun Microsystems Inc. *The Java Wallet(tm) Architecture White Paper*, March 1998. obtained from <http://java.sun.com/products/commerce/docs/whitepapers/arch/architecture.pdf>.

[46] Michael Tischer. *PC intern 3.0.* Data Becker, 1992.

[47] Sven Wohlgemuth. Schlüsselverwaltung - Objektorientierter Entwurf und Implementierung. Master's thesis, University of Saarbrücken, 2000.

# Index