Diploma Thesis

Security policies in Nizza on top of L4.sec

Stefan Kalkowski

September 29, 2006

University of Technology Dresden
Faculty of Computer Science
Institute for System Architecture
Operating Systems Group

Professor: Prof. Dr. rer. nat. Hermann Härtig
Assistant: Dipl.-Inf. Christian Helmuth

Declaration

I declare to have written this work independently and without using unmentioned sources.

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 29. September 2006

Stefan Kalkowski

Task formulation

Within the scope of the diploma thesis, the management of different security policies on top of the microkernel L4.sec has to be evaluated.

The resulting work will be part of a *Trusted Computing Base* software, which is under development. Therefore, it has to be compatible to this software. The following problems are of particularly interest for the exploration: How can a capability-based system be designed, so that it supports all imaginable security policies? In such a system, how can users and resources be mapped to names and identities of the platform?

The work has to include a prototypical implementation, as well as an evaluation of the design, regarding complexity and generality.

Aufgabenstellung

Im Rahmen der Diplomarbeit soll das Management verschiedener Sicherheitspolitiken, aufbauend auf dem capability-basierten Mikrokern L4.sec, untersucht werden.

Das Ergebnis der Arbeit soll Teil einer derzeit in der Entwicklung befindlichen *Trusted Computing Base* Software sein und muss sich entsprechend in diese einfügen lassen. Wichtige Fragestellungen für die Untersuchung sind: Welche Möglichkeiten gibt es ein capability-basiertes System generisch für alle möglichen Sicherheitspolitiken zu gestalten? Wie können NutzerInnen und Ressourcen in einem solchem System auf Namen bzw. Identitäten der Plattform abbgebildet werden?

Bestandteile der Arbeit sollen außerdem eine prototypische Implementierung und Evaluation des Entwurfs bzgl. Komplexität und Generalität sein.

Acknowledgment

First and foremost, I want to thank my supervisor Christian Helmuth, who helped me with answering a lot of questions, in finding design weaknesses and by giving me the freedom to develop my own ideas. Thanks goes to Norman Feske for his tirelessly work on Bastei and the support he spend to me. Furthermore, I'm thankful to Marcus Völp, for discussions about different policy models and Thomas Kriegelstein, who gave me hints on existing, theoretical work. The practical part of my work wouldn't exist without the patient answers of Christian Oschwald to my C++ questions. Also, I'd like to thank Sandro Merbd and the Conni e.V., who inspired the scenario in the design part.

Finally, I want to thank Hermann Härtig for facilitating the whole work, by establishing a research group with such a creative and pleasant working-atmosphere.

Contents

List of Figures

1 Introduction

Every day one can read several news about flaws in software, exploits and DoS-attacks, that were carried out by already infiltrated computer systems. Often, one adepts by these messages, that a buffer overflow or something similar was the reason for the successful attack, but hardly ever anybody asks, why a simple programming fault in an user application or system daemon is sufficient to compromise the whole system. Thereby, its no secret, that the principle of least privilege, consequently applied in an operating system, would diminish the vulnerability of the system drastically. But today's common systems are still far away from fulfilling this principle.

So, what are the reasons for this? One important prerequisite to fulfill the principle of least privilege are fine-grained Mandatory Access Control mechanisms. But common operating systems, that get enhanced by such mechanisms, like SELinux for instance, get refused by users, because they are overextended by its complexity. Of course, it's not the mechanism or policy language, which is responsible for the increased complexity, but the whole functionality of the system controlled by a monolithic reference monitor, that leads to a complex policy. Moreover, due to its monolithic nature, the TCB of such systems is oversized and therefore error-prone. To sum up, the monolithic structure leads to a complex, oversized security kernel, which moreover is difficult to control, as we have only one policy, containing all access control rules.

In contrast to traditional, monolithic operating systems, microkernel based systems promise to be more robust, to provide a tiny TCB, and to support different policies upon them. Moreover, they can be used to enforce different policies and even policy classes at the same time, which is essential for multilateral security. The security kernel L4.sec, a new member in the L4 microkernel family, that supports capabilities for controlling IPC, can be used to build systems upon it, which fulfill the addressed expectations. However, the arrangement of different policy models, the procedure of capability propagation and revocation, as well as proper abstractions for identity have not been designed yet for L4.sec. This is the focus of this work.

Goals: In my thesis I want to evaluate, how different policy classes can be mapped into a multi-server environment with more than one reference monitor. Thereby, generality has highest priority, that means it should be possible to use every conceivable policy class. I want to show the advantages of using several reference monitors in a nested configuration, in contrast to one monolithic reference monitor. Besides, the representation of users in the system, and the procedure of authentication will be subject of this work.

Outline: This document is structured as follows. In section 2 I will introduce you to today's variety of security policy models and mechanisms. Additionally, you will find a basic description of the overall architecture Nizza and the security kernel L4.sec, this work is integrated in respectively based on. Section 3 contains explanations to the access control framework Monitor, I have designed, and the overall framework Bastei, Monitor is a part of. Some implementation aspects of Monitor, concerning policy data transfer and persistence, service side policy enforcement, and reference monitor logic, I will discuss in section 4. An evaluation, regarding generality and complexity of the new framework constructs, as well as open issues, I present in section 5. At last, I sum up in short the whole work in section 6.

2 State of the art

A **security policy** with regard to a computer system defines it's authorized states, respectively the allowed operations, that held the system in a secure state. Compared with this, **security mechanisms** are mechanisms, that enforce security policies.

Major goals of security engineering are: firstly to analyze the security needs, from that to develop a security model, as far as possible of formal nature to simplify verification, and at last to construct mechanisms, that can enforce the policy, respectively a set of policies. Not surprisingly the security policy is closely coupled with it's security mechanisms. Actually policies are often implicitly implemented within the security mechanisms, rather than explicitly formulated; although the clear distinction of policy and mechanism is an important condition, to allow rearrangement of policies, and to support a wide range of distinct policies.

In this chapter, at first we shall examine some explicitly formulated and well-defined classes of policies; after that we will have a look at some common security mechanisms. At last I will present the current state of the actual platform, this work is designed for. Of course, hereby the focus lies on the currently available security mechanisms.

2.1 Policies

There is no general accepted taxonomy with regard to security policy models, but we can distinguish them, either by means of their primary protection-goals: as confidentiality, integrity, and availability, or by means of their historical arising, or based on the mechanisms they need. As the first and the second criteria somehow fits together and facilitates the understanding, we shall go that way.

Before going into deep and looking at specific security policies, I want to introduce a common, abstract model, that allows in general the definition of authorized and unauthorized operations in a system, and therefore can be used to express nearby all policies.

2.1.1 Access Control Matrix

The **A**ccess **C**ontrol **M**atrix (ACM) was firstly formulated by Lampson[Lam71] and later refined by Graham and Denning[Den71]. It had a great impact on later researches in the field of computer security.

Lampson abstracted a system as an aggregation of subjects or domains D and objects O whereby $D \subset O$. The subjects denote the rows of a matrix, whereas the objects are the columns. Every item in the matrix is a set of operations, respectively access rights the subject of the item's row is allowed to perform, with regard to the object of the item's column. Figure 2.1 shows an example of an ACM.

Some special operations allow a subject to change items of the matrix. How this is done, depends on the policy type. In the original model a special owner-right was introduced, that

	Domain1	Domain2	File1	File2	TCP-Port 80
Domain1	*owner control		*owner *read *write		write
Domain2		*owner control	read	*owner *read *write	read write

Figure 2.1: Example of an Access Control Matrix

allowed a single subject obtaining that right, to alter every item of the related object's column. Additional, a copy-flag (*) can be set for a single right, that indicates the possibility to transfer that right to another subject.

Apart from these special operations for altering the matrix, in the static case the ACM model is very general and can be used to express every kind of access control at a specific time. Some dynamic properties of certain models, like Chinese Wall (see section 2.1.2.5) or Usage Control (see section 2.1.2.7), cannot be easily expressed by the ACM. Nevertheless it provides a good abstraction, that helps to imagine some of the following models.

2.1.2 Formal models

The ACM provides the possibility to define access control in a system at a fine-granularly level, but it doesn't assists us to define specific policies with regard to special security needs. Furthermore it doesn't provide a definition of security at all. So it's fully up to the administrator of a policy to define a secure system. In the following sections I will introduce certain models, that provide some more support related to defined security goals. Outgoing from confidentiality policies, which were the earliest one, we will look at integrity concerned ones, up to hybrid and more complex models.

2.1.2.1 Bell-LaPadula Model

The main goal of confidentiality policies is to prevent the unauthorized disclosure of information. These first formal policy models in computer security, of course derived for military purposes, focused on the protection of classified information, that's why they are also known as military or governmental policies. The best known one is the Bell-LaPadula Model, that formed the basis for the **T**rusted **C**omputer **S**ystem **E**valuation **C**riteria[1] (TCSEC).

The Bell-LaPadula Model, first introduced 1973[BL73] and later refined[BL76] by Bell and LaPadula, assumes, that a system contains subjects, which are the active entities and objects the passive ones, although an entity can be both, dependent on the context. Objects can be accessed, either by read or write operations; accordingly every operation where information flows from object to subject is a read operation and vice versa a write operation[2]. Each subject and object

[1] The TCSEC is often called Orange Book in literature, because of the color of it's binding

[2] Bell and LaPadula actually used four access methods: execute, read, append and write, where execute meant no flow of information and write a flow in both directions. In this text write means the append operation of the original text.

has a security level. A security level consists of a classification or clearance (e.g., TOP-SECRET or UNCLASSIFIED) and a set of formal categories (e.g., BILL or CONTRACT). The classifications denote the information's level of protection and are linear ordered, whereas the category indicates the type of information. The security level of the entities is managed by the system mechanisms and should be tamper-proof (compare with MAC in section 2.1.3.2).

A subject should only read information from objects, if the subjects clearance is higher or equal to the objects one, and if the categories of the object forms a subset of the subjects ones. For example a General with security level: (TOP-SECRET, {NUCLEAR, CRYPTO}) should be able to read a file with level: (SECRET, {NUCLEAR}).

More formally, we can define an relation $dominates : (L,C) \times (L,C) \rightarrow \{true, false\}$, where L is the set of clearances and C the set of categories and $(l_1,C_1) dominates (l_2,C_2)$ is true, if and only if $l_1 \geq l_2$ and $C_2 \subseteq C_1$. Based on that, Bell and LaPadula defined the:

- **Simple Security Property**: A subject with security level S_1 can read an object with level S_2 if and only if $S_1 dominates S_2$.

This rule prevents direct access of unauthorized subjects. Now, suppose a subject, with read access on some highly classified data, copies the information to another object, with somehow lower restrictions. This would be a legal action, according to the access rights, but would conflict with the intended information flow rules, because now a subject, that formerly couldn't access the data, potentially can do so. To prevent this, another property was formulated:

- ***-Property**:[3] A subject with security level S_1 can write to an object with level S_2 if and only if $S_2 dominates S_1$.

These rules are widely known as **no read up** and **no write down** rule. As Bell and LaPadula have showed[BL76, p.75 et. seq.], the following theorem holds:

- **Basic Security Theorem**: A system with a secure initial state, that means a state, where every current access of subjects to objects don't break the Simple Security- or *-Property, will only enter secure states, if all possible operations of the system preserve both properties.

This sounds simple and evident. In practice, it is more difficult to proof, that the chosen mechanism will preserve both properties. Especially, if you consider that in this context read and write are used in an abstract manner.

The complete model includes another rule, which simply states, additional to the already mentioned constraints, that individuals may add further restrictions. Bell and LaPadula used therefore the notion of the ACM (see section 2.1.1), and called it **Discretionary Security Property** (compare with section 2.1.3.1).

One of the earliest systems, that supported the MLS requirements of Bell-LaPadula's Model was MULTICS. Better to say one version of MULTICS containing the so called **A**ccess **I**solation **M**echanism (AIM), that was added by project Guardian[Vle06]. After this early efforts a lot of different, mostly academic systems followed, that supported MLS (e.g., Scomp, Trusted Mach or SELinux).

[3] This is spoken star-property.

2.1.2.2 Biba's Model

The Bell-LaPadula Model (see section 2.1.2.1) concentrates solely on confidentiality. Integrity protection goals are not in the focus of the model. In contrast to that, Biba presented a model[Bib77], that only considered integrity concerns. Strictly speaking Biba's Model consists of three different models, whereas one titled **Strict Integrity Policy** is the fully dual of Bell and LaPadula's one.

Biba introduces the notion of integrity levels, which are very similar to the security clearances of Bell and LaPadula. The levels are ordered and a relation is defined, that determines if a level dominates another one.

The important difference in comparison to the Bell-LaPadula Model is, that a subject may read only data from an object, that possesses the same or a higher integrity level, and can only write data to objects, that have a lower or the same integrity level. So it can be seen as the inversion of the Simple Security Property and *-Property. But note: integrity levels and security labels are assigned and managed separately, and have to be distinguished! A third rule states, that subjects can execute other subjects only, if the integrity level of the second is dominated by the first one.

Somewhat more formally Biba's Model can be summarized as follows. Assumptive, a system consists of a set of subjects S and objects O and a set of integrity levels I. The levels are ordered and there exist some relation $\geq \subseteq I \times I$, that holds, if the first level dominates or is the same as the second one. Further, a function $i : S \cup O \rightarrow I$ exists, that returns the integrity level of either a subject or object. The rules of the Strict Integrity Policy are:

1. $s \in S$ can read $o \in O$ if and only if $i(o) \geq i(s)$
2. $s \in S$ can write to $o \in O$ if and only if $i(s) \geq i(o)$
3. $s_1 \in S$ can execute $s_2 \in S$ if and only if $i(s_1) \geq i(s_2)$

By replacing integrity levels with a combination of integrity clearance and category, one obtains the fully analogue of Bell-LaPadula's Model. As mentioned earlier, Biba presented two other models which are very similar. The one is less restrictive and ignores the problem of indirect modifications; only direct modifications are prevented. Therefore every subject can read every object, even objects with a lower integrity level. The other model, called **Low-Water Mark Policy**, changes the integrity levels of subjects according to the information it accesses. So the first rule stated above has to be changed to:

1. if $s \in S$ reads $o \in O$, then the new integrity level of s is $min(i(s), i(o))$

whereas $min : I \times I \rightarrow I$ returns the lesser of the two integrity levels.

2.1.2.3 Type Enforcement

Boebert and Kain criticized Biba's Model (see section 2.1.2.2) as impractical to meet integrity aspects. As result of their work on the SAT project[4], they proposed a framework for integrity policies[BK85], that don't have to be hierarchical structured, as with integrity labels.

[4] The NCSC as part of the NSA funded the Secure ADA Target project. It's goal was the development of a secure computer platform, that meets the A1 requirements of the TCSEC. It builds up on the results of the PSOS project and was later resumed by the LOCK project.[Say02]

Their argumentation was, that to implement hierarchical integrity policies, you need a tall TCB. This is necessary, because a lot of modules have to handle data of different integrity sensitivity, and therefore the integrity level of data often needs to be raised. A more flexible labeling and enforcement system is needed, that Boebert and Kain called **Type Enforcement**[5]. They saw their approach as supplement to the TCSEC demands.

As other security models, Type Enforcement characterizes a system as composition of subjects and objects. The subjects are grouped in domains, whereas the objects have associated types. The access rights, that domains possess with regard to types, are arranged in a **D**omain **D**efinition **T**able (DDT), that is similar to the ACM (compare to section 2.1.1). In addition, there exists a **D**omain **T**ransition **T**able (DTT), that determines which domains can execute which other domains. Type Enforcement is structured like the ACM, but achieves a drastic reduction of needed rules and table space, because subjects are grouped to domains and objects to types. Needless to say, that you can use Type Enforcement not only for integrity concerns, but also for confidentiality ones. In contrast to the Bell-LaPadula (see section 2.1.2.1) and Biba Model (see section 2.1.2.2) this model doesn't provide a simple solution to define security. While being capable to express sophisticated policies, a policy designer risks to construct flawed policies.

In later efforts, Badger and associates[BSS$^+$95a] applied this approach to UNIX-like systems and named it Domain and Type Enforcement (DTE). One variant of Type Enforcement is implemented in SELinux[Nat06].

2.1.2.4 Clark-Wilson Integrity Model

Clark and Wilson[CW87] investigated the commercial needs, in contrast to the so far focused military ones and determined, that integrity should be the main protection goal. Therefore, they stated two mechanisms, which are desired: **separation of duty** and **well-formed transactions**. The first one means, that operations should be organized into subparts and be only accessible by different people to prevent fraud. For example, in many companies the responsibility of ordering and paying is often divided into different sections, that's why fraud is less probable. Of course, this aim could also be achieved with the formerly known Biba Model (see section 2.1.2.2) by choosing proper integrity levels and categories.

The well-formed transaction was in a sense a new approach, that implied the notion of consistent states and checks. Such a transaction is a sequence of operations, that has to preserve consistency of the system. A very simple example would be the transfer of money from one account to another within a bank. A single operation, for instance if the sum is added on the target-account, will bring the bank's accounting in an inconsistent state, but in conjunction with a subtraction from the originator-account, it forms a well-transformed transaction.

The main point of the model is, that only certain transactions can touch certain data sets or objects, and users are constrained in their access to these transactions.

More formalized, the model of Clark and Wilson requires, that data, whose integrity has to be protected, must be labeled by the system. These data items are called **C**onstrained **D**ata **I**tems (CDI); other data is called UDI. An integrity policy describes two different types of procedures: **I**ntegrity **V**erification **P**rocedures (IVP) and **T**ransformation **P**rocedures (TP). An IVP is needed to check consistency of the CDIs at the time it is executed, whereby the TPs are the well-formed transactions. The following rules have to be fulfilled:

[5] Meanwhile Type Enforcement is patented by the Secure Computing Corporation.

- **C1 (Certification):** All IVPs must properly ensure, that all CDIs are in a valid state at the time the IVP is run.
- **C2:** All TPs must be certified to be valid. That means, they must take a CDI into a valid final state, given that it is in a valid state to begin with. Further on, it has to be defined, which CDIs a TP is designed for.
- **E1 (Enforcement):** The system must ensure, that only TPs, certified to run on a CDI, manipulate that CDI.
- **E2:** The system must associate a user with the TPs, it is allowed to run. Furthermore, it has to maintain which CDIs a specific TP may access on behalf of that user. Such a relation forms a triple: (user, TP, {CDI1, ...}). The system has to enforce, that access by users to CDIs through a TP follows the defined relations.
- **C3:** The defined relations must be certified to meet the separation of duty requirements.
- **E3:** The system must authenticate the identity of each user attempting to execute a TP.
- **C4:** All TPs must be certified to write to an append-only CDI all information neccessary to reconstruct it's operations.
- **C5:** Any TP, that takes an UDI as input value, must be certified to perform only valid or no transformations, for all possible values of the UDI. The transformation either rejects the UDI or transforms it into a CDI.
- **E4:** Only the certifier of an entity may change the list of the entities, it is associated with (e.g., a TP associated with CDIs). The certifier of an entity, must not have execute permissions with respect to it.

The first three rules ensure consistency of the CDIs. Rule E2, C3 and E3 provide mechanisms to fulfil the principle of separation of duty. Certification rule 4 defines how illegitimate transactions can be reverted. The remarkable feature of the Clark-Wilson Model is, that it defines how entities can be upgraded (compare with rule C5). A feature, that Biba's Model (see section 2.1.2.2) misses. In Biba's Model a security administrator would have to certify each unclassified data, that gets part of the system, which seems to be very unrealistic. The last rule is another example for the principle of separation of duty, and prevents a user from adding TPs, which can later be executed by him or her.

An implementation approach can be found in the thesis of Karger[Kar88]. He shows how to establish the model on top of SCAP - the secure Cambridge capability system.

2.1.2.5 Chinese Wall Model

The Chinese Wall Model was introduced by Brewer and Nash[BN] and is derived from the legal demands of the UK's Stock Exchange. Clearly, in such environments analysts, that work on companies data, have to be prevented from accessing data of competitive companies. Otherwise, insider information could be used for fraud.

Brewer and Nash proposed a three level hierarchical organization of data. The lowest level represents a single object. Objects related to a single company are grouped together to a **company dataset**. The highest level consists of **conflict of interests class**, which are sets of competitive company datasets. Figure 2.2 shows some exemplary dataset.

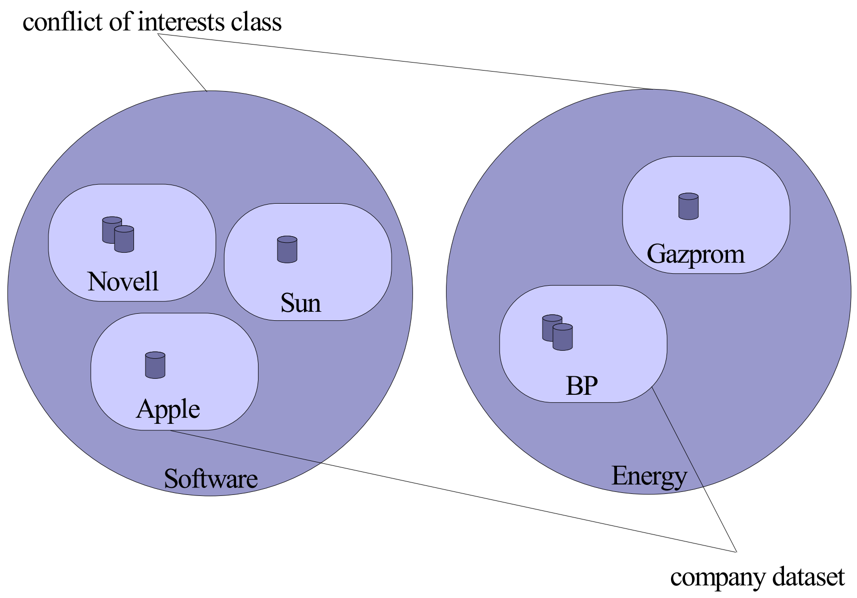

Figure 2.2: Organization of the data in the Chinese Wall Model.

The Chinese Wall Model assumes, in addition to the already mentioned objects, a set of subjects, whereby a subject is a clearly defined user[6]. It defines the following rule in analogy to the Bell-LaPadula's Model (see section 2.1.2.1):

- **Simple Security Rule:** Access to an object is only granted to a subject, if one of the following is true:

  1. The requested object is in the same company dataset as an object already accessed by that subject.
  2. The requested object belongs to an entirely different conflict of interest class than all objects previously accessed by that subject.
  3. The requested object is a sanitized one.

Hereby, sanitized means, that all sensitive information is removed from the object. To prevent indirect flow of information, Brewer and Nash defined a second rule analogue to Bell and LaPadula:

- ***-Property:** Write-access to an object is only granted to a subject, if the following is true:

[6] In contrast to previous definitions, where subject and process were interchangeably used, it is necessary for the following description, that we think of a subject as a single user

1. The access is permitted by the Simple Security Rule.
2. No object can be read by the subject, that belongs to a different company dataset than the requested one, unless the other company dataset contains only sanitized objects.

The really important point of view with regard to the Chinese Wall Model is, that the decision to grant or deny access is based on the access history of the subject. That part of the model can't be emulated by previously introduced ones. On the other hand, the Chinese Wall Model cannot emulate the Clark-Wilson Integrity Model as it has no equivalent to the verification rules in that model. As Brewer and Nash showed[BN], the two models are consistent and therefore can be used in conjunction.

2.1.2.6 Role-Based Access Control

The notion of **R**ole-**B**ased **A**ccess **C**ontrol (RBAC) was firstly named by Ferraiolo and Kuhn[FK92]. They argued, that in companies access rights are mostly linked rather with jobs than with certain users. Therefore, they proposed an abstraction, where subjects are associated with roles, and access rights are assigned to that roles.

In RBAC a subject s has one or more roles, it is authorized for. The function $authorized(s)$ returns that set of roles. The subject has one active role at a given time, which we will denote with $active(s)$. A role r is authorized to perform one or more transactions, namely $trans(r)$. In RBAC the following rules apply:

1. **Role assignment:** $\forall s \in S, s$ can execute **any** transaction only if $active(s)$ is not empty.
2. **Role authorization:** $\forall s \in S, active(s) \subset authorized(s)$.
3. **Transaction authorization:** $\forall s \in S$, $\forall t \in T$, s can execute t only if $t \in trans(active(s))$.

An important property of RBAC is that rights, needed for a job, can easily be assigned to users, that are new, or which terms of reference changed. This can be done by assigning predefined roles to them. One further aspect of the model is the composition of roles to new ones. An example can be seen in figure 2.3.

RBAC has been adopted as a standard by ANSI. There exist several implementations and variations of it, for example in Microsoft's .NET framework. In the area of operating systems, Solaris OS[7] provide RBAC[Cha03], another example is SELinux[Nat06].

By combining the notions of RBAC and Type Enforcement, one obtains an anymore flexible model. We can simply take the RBAC model, and additionally associate objects with types, or even organize them in a type hierarchy, analogue to the role hierarchy. Such a combination can be found in SELinux[Nat06].

2.1.2.7 Usage Control

Usage **CON**trol (UCON) is a relatively new approach, that tries do combine prior models on a high abstraction level. It is mostly related to DRM endeavors, because the authors explicitly

[7] Starting with version-number 8.

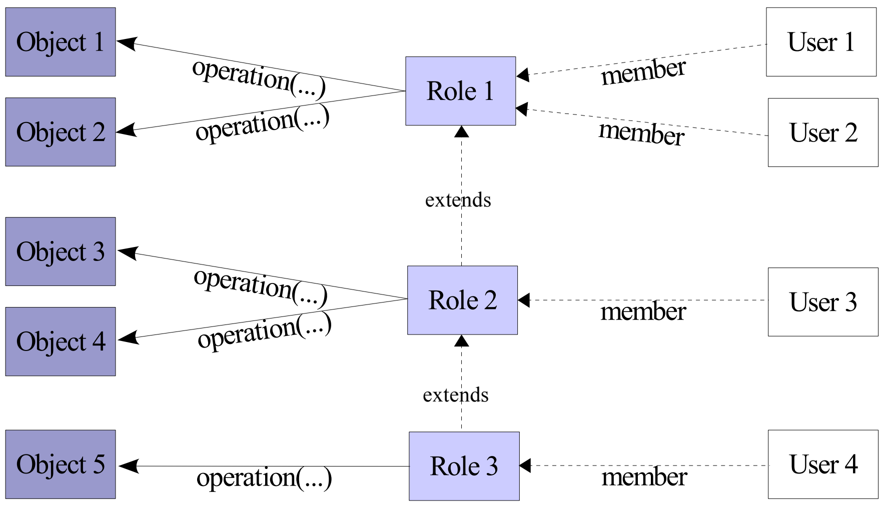

Figure 2.3: An example of a role hierarchy.

covered that field. Nevertheless, it generalizes certain models and provides lesser a concrete policy class, but rather an abstract description of what is needed to formulate sophisticated policies. Therefore, it is of interest for this work and will be introduced shortly.

The authors of UCON, Park and Sandhu describe it[SP02b][SP02a] as a mixture of Trust Management, DRM, ORCON (compare with section 2.1.3.3) and traditional access control. The UCON model abstracts a system, as former models did, to subjects, objects, and rights. Additional, it obtains authorization rules, conditions, and obligations. Figure 2.4 illustrates the model.

Rights are described as privileges, that subjects hold on objects, so it can be thought of as capabilities (see section 2.2.1.2). The authorization rules are requirements, that should be satisfied before rights are granted. They can be thought of as traditional access control rules (e.g., the label of the subject has to dominate the object's label for read access). In addition, the rules contain obligation-related rules, meaning the control of the fulfillment of obligations, a subject agreed on. An obligation might be an auditing function, performed while a subject modifies an object. While authorization rules are checked before access is granted, including obligation-agreements, the fulfillment of obligations has to be traced after access was established.

Further rights related components are conditions. One can distinguish stateless and state-dependent conditions. An example for a state-dependent condition might be the previous access to a company's dataset. That condition denies access to other datasets, as in the Chinese Wall Model (see section 2.1.2.5). A given time period, in which access is granted solely, is an example of a stateless condition.

While covering a lot of aspects of former models, like access-history tracking or auditing, UCON doesn't provide a detailed framework how the rules, obligations, and conditions can be expressed, not to mention a definition of how certain security goals can be achieved. However, it is a powerful, abstract model, that covers some aspects, that are not covered by other framework-

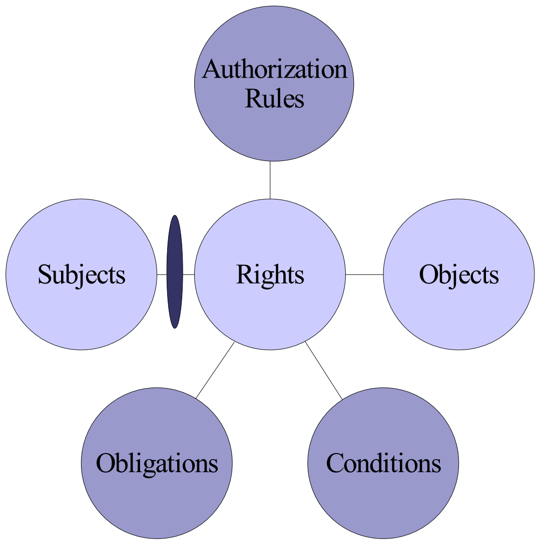

Figure 2.4: The UCON model.

like models, as ACM (see section 2.1.1), Type Enforcement (see section 2.1.2.3) or RBAC (see section 2.1.2.6).

2.1.3 Types of Access Control

Beside additional mechanisms like auditing[8], the main tool to enforce the different security policies is to confine the flow of information by controlling access, subjects have to objects. The three different forms, that are needed by the previously presented policies, will be summarized and sharpened in the next three sections.

2.1.3.1 Discretionary Access Control

The term **D**iscretionary **A**ccess **C**ontrol (DAC) was introduced by the TCSEC[Cen83] and is derived from the Discretionary Security Property of the Bell-LaPadula Model (see section 2.1.2.1). All systems with evaluation class from C up to A support this feature. DAC is defined as an enforcement mechanism, provided by the TCB[9], that allows named users to specify and control sharing of named objects. In particular, this means, that a subject, that possesses certain permissions, is capable of passing that permissions away to other objects.

Typical examples for DAC are legacy file-systems as found in UNIX-like systems. These allow users to control access to files based on the identity of the requesting subject, that can be distinguished by user- or group-ID.

[8] Strictly speaking, auditing can also be reduced back to information flow and access control.

[9] TCB is used with varying semantic. In this document we use the definition of the TCSEC. According to that, the TCB is the "totality of protection mechanisms – including hardware, firmware and software – the combination of which is responsible for enforcing a security policy"[Cen83, p.115]

2.1.3.2 Mandatory Access Control

As same as DAC the definition of **M**andatory **A**ccess **C**ontrol (MAC) goes back to the TCSEC, and the notion is derived from the Simple Security Property and *-Property of the Bell-LaPadula Model (see section 2.1.2.1). Systems, that shall be evaluated as class B or A, have to provide MAC mechanisms. Originally, it was defined as mechanism of the TCB, that on the one hand labels every subject and object, according to the form of label presented by Bell and LaPadula, and on the other hand restricts all access methods in such a way, that the Simple Security Property and *-Property won't be hurt. The important property of MAC is hereby, that it can't be broken by DAC mechanisms.

Today, the term is almost used with a widened meaning, that is: all system mechanisms, that control access to objects and cannot be altered by an individual user, provide MAC[Bis03, p.103].

Normally, MAC is used in combination with DAC. Of course, hereby MAC rules have to dominate the other ones. Accordingly, the existence of DAC helps the user to add further restrictions, but not to grant access that is otherwise forbidden.

2.1.3.3 Originator Controlled Access Control

ORiginator **CON**trolled Access Control (ORCON), first introduced by Graubart[Gra89], is a mixture of DAC (see section 2.1.3.1) and MAC (see section 2.1.3.2). Like in MAC, the access control relationship between subject and object cannot be altered by the owner of the object and as a consequence of this, if an object is copied, it's imposed restrictions have to be copied too. In contrast to MAC, there aren't centralized rules, that handle access to entities. The control of who may access an object is at the complete discretion of it's originator. Originator hereby means originator of the information, that lies in the object and has to be distinguished from the owner of an object.

ORCON defines the following rules:

1. The creator (originator) of an object is the **only** subject that can alter the access control restrictions of that object on a per-subject basis.

2. When an object is copied, the access control restrictions of the source are bound to the target of the copy.

The first rule is somehow a DAC rule, whereas the second is a MAC rule. A problem arises due to the second rule. Strictly speaking, every object o_1 a subject newly creates, while it has access to another originator-controlled object o_2, has to be controlled in the same fashion like o_2. Graubart therefore proposed **P**ropagated **A**ccess **C**ontrol **L**ists (PACL), which are associated with originator-controlled objects.

A PACL similar to traditional ACLs (see section 2.2.1.1) contains subjects, that obtain rights to access the concerning object. When a subject accesses an ORCON-object, the PACL binds to that subject. If that subject creates an object, while associated with some PACLs, these PACLs get associated with the newly created object too. In this respect, PACLs label the flow of information in the system. Whereby Graubart restricts the labeling to newly created objects only, already existing objects, information could flow in, are neglected by him.

Remarkable in ORCON surely is the decentralized approach coupled with MAC mechanisms. Similar, extended approaches are followed by the Decentralized Label Model[ML97] and the AsbestOS[EKV$^+$05].

2.1.4 Security policy languages

When security policies aren't implicitly implemented in the security mechanisms of a system, but explicitly formulated, a form of representation is needed. There exist low-level languages, closely coupled to certain security kernel implementations and high-level languages, that can express policy classes. For example, Badger and associates presented a high-level language[BSS$^+$95b] with respect to their Type Enforcement (see section 2.1.2.3) approach for UNIX-like systems.

Additionally, companies and research institutions made efforts to develop more general languages, that are able to express as possible very different policy types. The e**X**tended **A**ccess **C**ontrol **M**arkup **L**anguage (XACML)[OAS06] is one result of these efforts. It is an XML language, that can be used, both to formulate sophisticated access control rules, and to construct and analyze requests and responses. XACML is standardized by the OASIS.

But instead of considering all available security policy languages now, we shall have a look at the fundamental aspects of them. An overview of language requirements can be found among others in the paper of Vimercati, Samarati and Jajodia[VSJ]. With the following, I will present only some short points:

- There is a need to denote entities, whether distinguished by subjects and objects or not. Furthermore a language must be able to express relations between them, which are the controlled operations.
- A flexible and also simple language should provide the possibility to structure subjects, objects and even operations hierarchical, to minimize the needful rules.
- It should be possible to associate logical statements (conditions) with triples of the form (subject, object, relation), that have to be evaluated, when an access decision is made.
- The language should provide the possibility to define default rules. With other words, if no rule is available related to a specific request, access will be allowed (open policy) or denied (closed policy)
- Another main aspect in policy languages is: how can you express rules to resolve conflicting rules? For example, you can decide, that more general rules have to dominate specific ones or the other way around.

2.2 Mechanisms

The required security mechanisms, that derive from the policies and models, described in the previous sections, are the following. There is a need for an access control mechanism, that provides the different access control strategies like DAC, MAC and ORCON. For that, support for labels of different system objects is needed. Both, access control and labels are desired with flexible granularity. Fine-grained access control is truly the foundation for flexible policy support, as stated by Spencer and associates[SSL$^+$99]. Imagine, one is able to control every single operation, that is done by the system, then one obtains the ability to enforce every security policy. Of course, this is unrealistic and even not wanted. We only want to control the operations or accesses defined by a given policy.

Realizing access control involves implementing propagation and revocation of rights. The control of rights propagation and revocation can be a difficult aspect in modeling an operating system.

One thing beside access control, that is often ignored, is the control of resource usage. Furthermore the question, how identity is brought into system, is still open. We will discuss the state of the art of all these aspects in the following sections.

2.2.1 Mechanisms for access control

Basis for the control of access in computer systems is to prevent or allow processes to touch certain locations in memory. On the lowest level, this is normally done by a combination of protection rings, segmentation and/or paging. For example, a program, that is associated with the lowest used ring, is therewith in the position to organize some distinct address spaces, used by programs running on higher rings, and to provide some kind of controlled sharing of information between these processes.

Such an example of a combination of hardware and software is called **security kernel**. That is an implementation of the **reference monitor** concept, an idea that was introduced by Anderson[And72]. A reference monitor mediates all accesses of objects by subjects. That means in the previous case, it mediates every memory access.

An interesting aspect in the implementation of a reference monitor is, how the access rights are arranged respectively located. One possibility is to hold all access rights in a table analogue to the ACM (see section 2.1.1). But such a table would waste a lot of memory needlessly. Two different approaches exist to overcome this space problem. One can, whether associate every object with a list of subjects and their particular access rights, or the other way round, bind a list of objects and the corresponding rights to every subject. Both solutions eliminate empty items of the ACM. The first implementation approach is called **A**ccess **C**ontrol **L**ists (ACL), the second capabilities or capability lists.

The interpretation of the terms ACL and capability is controversial. Often they are erroneously associated with other concepts, like DAC and MAC. Before comparing both concepts with each other, we shall investigate some examples, whereby focus lies on capability systems, as the security kernel, this diploma thesis bases on, provides capabilities as primary protection mechanism.

2.2.1.1 Access control lists

ACLs often get equated with the traditional POSIX file system object permission model, where every file is associated with a bit-field, that indicates for owner, group and others, whether they have read, write or execute permission on that file. This is is clearly an abbreviation of ACLs, that can handle much more fine-grained access control.

For instance, the memory protection model of MULTICS[BCD69][Sal74] included some kind of ACLs, as one of the first systems at all. In MULTICS segments are protected by an ACL that contains all principals that are allowed to access that segment. Access can be defined as read, write, execute and append or a combination of that three rights. Every process has an identifier, that consists of a user, project, and compartment field. So this combination provides the facility of fine-grained memory sharing between different processes in the system by using ACLs. The single identifier fields can be used in conjunction with a wild-card (*) to limit the number of needed entries in an ACL.[10]

A more recent example is the FLASK[SSL$^+$99] architecture, that reaches access controls by a combination of object managers (e.g., a file-server), which enforce access control and separated security servers, that decide if access is granted or not.[11] Because decision is made alone by the security server, it must know the requesting subject and the target object and has to handle some kind of ACM or something similar. To enhance performance, the access control decisions are cached by an **A**ccess **V**ector **C**ache module (AVC) within the object manager. Because the AVC is located at the object and access decision is made on basis of it, it can be seen as a kind of ACL.

As can be seen ACL models differ with regard to the objects they protect, the subject's representation they contain, and the operations, that can be protected. So UNIX file-system strictly abbreviates what subjects can be differenced, namely owner, owner-group, and others. In most today's operating systems access control is bounded to file-systems only, beside primary memory protection. Therefore in practice ACLs are often adjusted solely to directories and files and only protect read-, write- or execute- operations.

A clear different approach, that was already described in section 2.1.3.3, are propagated ACLs or short PACLs. Hereby the ACL is more associated with the information in an object instead of the object itself.

2.2.1.2 Capability lists

Capabilities, first introduced by Dennis, locate "by means of a pointer some computing object, and indicates the actions that the computation may perform with respect to that object."[DH66, p.145] In other words they are a composition of reference and authority. In comparison with ACLs, capabilities are often described as rows of an access control matrix.

In practice they are an unique string of bits, owned and unforgeable by some subject. If that subject wants to access the object referenced by the capability, it is proven by means of the capability, if that kind of access is allowed. Capabilities suit well to the object-oriented world. There are in general two possibilities to held them unforgeable. One is to use some protected memory segment in the process's domain, that can be modified only by the TCB. The other approach uses cryptography to make it difficult to guess a correct capability. This is particularly interesting for distributed systems. The Amoeba system for instance uses 128-bit capabilities,

[10] Strictly speaking, the segment model of MULTICS includes furthermore a ring-number pair, that is associated with each segment. If a procedure, currently running in ring r, wants to access a segment with the ring-number pair (r_1, r_2), then full access is granted, as long as $r \leq r_1$; when $r_1 \leq r \leq r_2$ then only read and execute are permitted; if $r_2 \leq r$ all access is denied. Here the ring-number pair can be seen as an additional ACL.

[11] The separation of policy and enforcement is an important design principle, that stood at the head in the FLASK project. This principle is needful to guarantee real flexibility in policy support.

where the last 48 bits are randomly selected at creation time, and validity of that bits is checked at access time by the server, that created the capability.

A lot of different capability models were developed in the last four decades. In the original model it is necessary and sufficient to possess a capability, to perform all operations, which that capability allows. Furthermore, capabilities can be copied unrestricted between different subjects, as far as they own the capabilities to transfer data between each other. Such a model fits perfectly the DAC requirements, but if some MAC has to be established problems arise, because propagation of authority cannot be limited. In other words, that model isn't able to solve the confinement problem[Lam73].

Several adaptations were made to the original model to overcome this weakness. Kain and Landwehr proposed a taxonomy[KL86] for capability-based systems including two possible solutions. One is to assign at creation time each capability list the security level of the subject, that list is assigned to, and to set the proper access rights, related to the referred object's security level. Every time a capability gets copied it's permissions are restricted on the basis of the target's security label and the security label of the object, that capability refers to. A similar approach is followed by the ICAP system[Gon89], whereby here not only a security level is assigned to a capability list, respectively a segment the capabilities are stored in, but full identity is coded into each single capability. This approach might be useful for distributed systems.

The second solution is to copy capabilities freely, and to determine the correct access rights not before access is prepared. Karger chose that way in SCAP[Kar88]. To reduce the policy decision overhead here, access decisions have to be cached. Outgoing from the presumption, that access to an object occurs more often than passing it's reference away, the first variant of control at propagation time promises a better performance. Moreover, we can use special capabilities solely for capability passing; that don't need additional checks. These special capabilities can be used between processes of one security level to reduce decision overhead.

Another problem in capability-based systems is the revocation[12] of a formerly propagated capability, for instance when the related object changes it's security level. One possibility is to pass through all processes and check their capability lists, which self-evidently implicates a great performance overhead. An alternative approach uses indirection, for example a global table, where the objects locations are stored in and indexed by capabilities. If the corresponding index of an object is changed, no previously granted capability can access that object anymore. Shapiro and associates[SSF99] intended to use indirection as revocation mechanism in EROS, whereby here indirection can be done by inserting indirection objects and by destroying such an object if access to the real object gets revoked.

A delayed form of revocation can be reached by including an expire-field in the capability, that determines how long a capability is valid. Such an approach is particularly interesting in distributed systems, where centrally administrated revocation is more difficult.

In MULTICS immediate revocation is reached by using back pointers[Kar89][BCD69][Sal74]. These pointers refer from the segments (compare to section 2.2.1.1) to segment descriptors, that are stored local in the process's descriptor segment. Segment descriptors not only support the translation of names to segment numbers, but also cache the access rights of the concerning process with respect to the segment. As a combination of reference and access rights, they can

[12] When speaking about revocation, I mean particularly immediate revocation.

be seen as a kind of capabilities. When an segment's ACL is changed, the back pointers are used to invalidate the corresponding segment descriptors.

2.2.1.3 Comparison between ACLs and capabilities

Now, we will discuss strengths and weaknesses of ACLs and capability lists in comparison. If you want to know what objects can be accessed by a given subject, of course capabilities are more helpful. This is is the case for instance, if a subject gets removed from the system. In ACL-based systems all objects have to be passed to lookup their ACLs and possibly remove the subject from it. The other way around, if you want to know what subjects can access a given object, ACLs fit best. For example, the security clearance of an object has to be changed and for the particular subjects, that might have access to the object, the rights have to be revalidated. In capability systems this might be a difficult task (see section 2.2.1.2).

These obvious differences are results of the direction of the references between subjects and objects, that can also be seen in Figure 2.5.

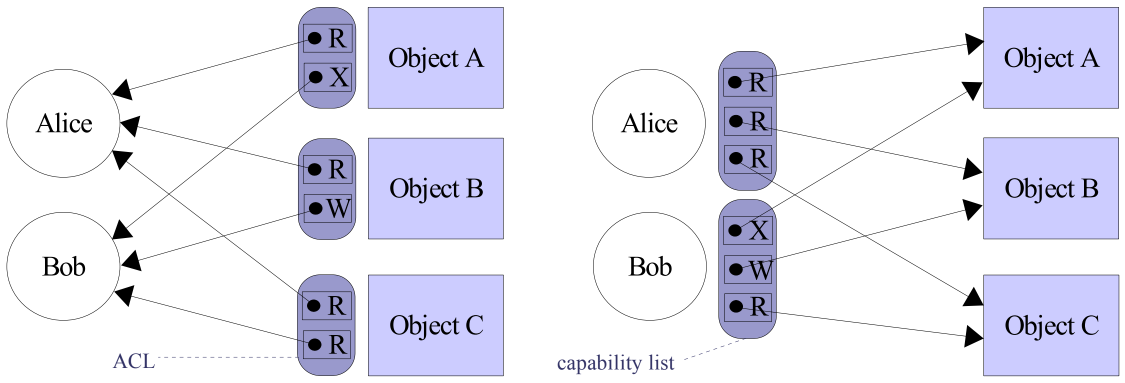

Figure 2.5: Comparison of reference direction in ACL and capability model.

The direction of reference is the main point, as Miller, Yee and Shapiro recognized[MYS03]. Because access goes off from subject to object, clearly a name-space, containing object references, is needed on the subject's side. In addition, an ACL needs a name-space in which to refer to subjects to fulfil it's task. This implies additional overhead, when using fine-grained subject identities. In the ACL model maintaining consistency is difficult, if subjects are frequently appearing and disappearing. To circumvent the overhead of either big ACLs or ever modifying of ACLs, typically less fine-grained subjects are used. For instance only the user-name is applied to all processes, that execute on behalf of a user. This tactic conflicts with the principle of least privilege. In contrast, capability-based systems doesn't need a name-space of subject references on behalf of the object.

Another strength of capabilities is there property, that designation on the subject's side is always coupled with authority. This implies the ability to solve the confused deputy problem[Har88].

2.2.2 Resource control

Until now, we only focused protection in relation to confidentiality and integrity. The question arises, whether availability concerns can be expressed with the so far described models or not, and if the formerly stated mechanisms are sufficient for that. Obviously, models like the ACM are not adequate to express rules for preventing denial-of-service. There has been little efforts to develop corresponding frameworks or policy classes. One attempt was made by Millen with his Resource Allocation Model[Mil92].

He stated the necessity of a resource monitor, analogue to the reference monitor notion (see section 2.2.1). The resource monitor has to be invoked, every time a shared resource has to be allocated. Furthermore, he noticed, that revocation of resources must be possible, and all access has to be bound to time. The resource monitor is somehow joined with the TCB's reference monitor, because both are invoked while resource access.

Without going into deep, one can summarize, that the need for fine-grained access control to all shared resources of the system, as well as the possibility of revocation of formerly granted access, goes together with the needs, that arise out of the confidentiality and integrity protection goals. Apart from that, the logic of such a general resource monitor is much more complex, including among others scheduling and logic to detect and solve deadlocks.

2.2.3 Identity and authentication

We have dealt a lot with subject's identities so far, but didn't considered how identity is brought into the system. This will be done now. The process of assigning identity to a subject is called authentication. Therefore a principal has to provide some knowledge, possession or properties. This can be a password, an electronic badge, or biometric properties. I don't want to present the different known authentication mechanisms now; it is out of the scope of my work. We shall only examine some aspects often ignored.

At first, in most systems a user has to authenticate him- or herself only one time, till he or she explicitly logs out, or gets logged out by the system, due to inactivity. But some authentication techniques can enduringly proof identity, for instance by analyzing key strokes. Another point is, that normally authentication mechanisms are defined implicitly by the system mechanisms and hold for all users indifferently, but like other security mechanisms it is needful, that the authentication policy is distinct from it's mechanisms and can be expressed on a per-principal basis.

Another important issue is, when a user or remote process is going to provide authentication information, the principal has to trust the authentication mechanism and therefore the underlying system. Trust in the running system can be established by authenticated booting[Gro91][Kau04]. Furthermore, to prevent tampering of a login screen or eavesdropping of keyboard events, a minimal, trusted window manager, like for instance nitpicker[NFCH05], is needed.

Beside the user, there are other entities providing authentication information. Every process, that isn't executed on behalf of some user, potentially needs to be associated with some form of security label, for example a domain in Type Enforcement (see section 2.1.2.3). Therefore, something like a secure file-system is needed, that protects integrity of stored programs, for instance by using hash values. Of course, hereby trusted computing techniques are needed to hold keys and hash values.

2.3 The security architecture - Nizza

The overall architecture, this work is designed for, is called Nizza. The most important design principle of Nizza is isolating all security sensitive code to obtain a minimal TCB[Här02][HHF$^+$05]. To achieve that goal, Nizza uses a multiserver-system approach. Upon a microkernel are running several trusted core components, for instance servers, that manage physical memory, devices, graphical user interface and so on. You can see the architecture in figure 2.6.

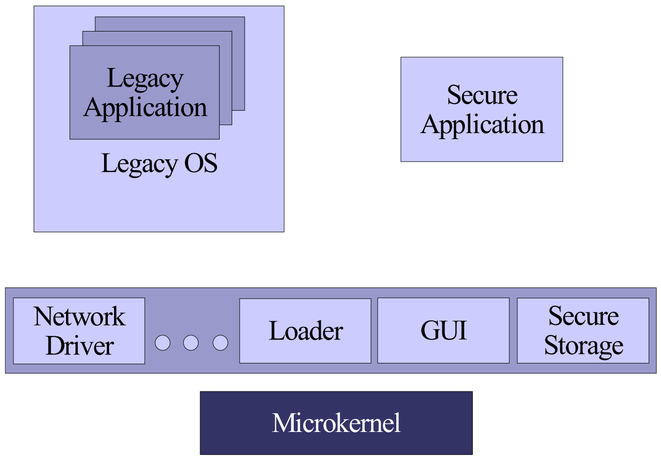

Figure 2.6: The Nizza architecture.

The core components are running in distinct address-spaces and have small interfaces. To achieve minimal size of components, you can use technologies like trusted wrappers, that means reusing untrusted components by protecting information on higher protocol layers. For example, imagine you want to save sensitive information to disk. Instead of constructing a complete, new file-system server, you can use an existing file-system implementation, as long as your information gets encrypted and hashed by a trusted wrapper, before it gets committed. A detailed explanation, how trusted wrappers help to reduce the TCB, can be found here[HPHS04].

For running legacy applications, for instance used in conjunction with trusted wrappers, legacy operating systems are integrated in Nizza by para-virtualization. This means the operating systems gets modified to accept the underlying abstractions (the core components), as its actual machine. This strategy allows you to use a variety of existing applications, without the need to create a complex ABI-compatible environment. At this time, there exist L^{4}Linux as legacy operating system in the current implementation of Nizza. As the name suggests L^{4}Linux is modified to fit the L4 microkernel interface. Therefore, it is not surprising, that an L4 microkernel is used as security kernel in the current Nizza implementation.

We shall now have a look at a new L4 microkernel, especially redesigned to better fit security issues.

2.3.1 The security kernel - L4.sec

The L4 microkernel family has in common, that the kernel doesn't implement any policies and only provides a few abstractions needed to achieve separation of programs upon it. It provides distinct virtual address spaces, also denoted as tasks, in which threads execute user-level code. Therewith, threads can communicate with each other, a third abstraction beside tasks and threads is implemented by the kernel, namely **I**nter **P**rocess **C**ommunication (IPC).

Currently available L4 specifications, like L4v2[Lie96], L4X0[Lie99] and L4X2[DLSU04], don't provide mechanisms to effectively restrict IPC between two processes. Therefore, the operating system research group of the TU-Dresden concurrently developed an extended L4 specification L4.sec, that is still in progress. It's current experimental implementation[Kau05] is called Florence. The main new features of L4.sec are endpoints and capabilities. Endpoints are associated with tasks. IPC is done between endpoints, not directly between two threads. That implies: Various threads can listen to an endpoint or send to it within a task, which simplifies the construction of multi-thread servers.

The additional new feature, the capability is a reference to a kernel object with assigned permissions. Kernel objects, referenced by capabilities are endpoints, tasks, threads, kernel-memory objects, and CPU-reservations. Capability IDs are task local and build an own name-space, beside the memory name-space. The most interesting capabilities for our meditations are endpoint capabilities, because controlling them means: control of IPC. An endpoint capability has the following permissions: send, receive, and map. The map permission is used to allow or deny the propagation of pages and capabilities over that endpoint. Moreover, a badge is attached to each endpoint capability, that is transfered to the receiver, each time that capability is used for sending. The interpretation of the badge can be used for different purposes and it's up to the receiver to interpret it. Because badge transfer is enforced by the kernel and is tamper-proof, it can be used to represent the sender's identity or permissions.

An experimental feature of L4.sec is directive unmap, that supports selective revocation. By specifying a mask you can unmap those mappings, whose badge corresponds to the mask.

2.3.2 Related work

Projects that are closest to Nizza are Perseus[PRS$^+$01] and Microsoft's NGSCB (previously known as Palladium). As far as i know, there exists no further descriptions how these projects integrate different security policies in a flexible fashion.

Another project, that explicitly determined how to achieve policy flexibility, is FLASK[SSL$^+$99] (see section 2.2.1.1). The original FLASK architecture was implemented by using a Mach-based microkernel named Fluke. Later the NSA integrated FLASK in Linux[Nat06] and created SELinux. In contrast to most other operating systems (except for MULTICS) the FLASK people realized the importance of permission revocation. A short-coming of the design is the central point of the security server, that determines every security decision. To support dynamically a wide range of policies, that component might become to great in size.

EROS[Sha99] is another example of a capability system built upon a microkernel. It has inspired the work on the L4.sec specification a lot. It's successor project is the Coyotos microkernel, which is in fact very similar to L4.sec. Nevertheless, the overall system architecture, that puts different security policy in practice, is missing here.

3 Design

In this chapter I will introduce you to a new operating system architecture called **Bastei**, that was developed in parallel to this work[1]. To be more illustrative, we will study the new features of the architecture on the basis of a real-world scenario. Derived from the needs of the scenario and the existing knowledge in the field of security policies and mechanisms (compare to section 2), I have designed an access control framework, that can be used within Bastei, which will be presented at the end of this chapter.

3.1 Bastei Architecture

On the one hand, Bastei[2] is the attempt of a complete clean re-design of the L4Env components[l4e03] and their interfaces, which became more and more interweaved. On the other hand, it was necessary to integrate new access control primitives like that of L4.sec (see section 2.3.1) in the user-level components.

The main design characteristic of Bastei, beside the general principles of Nizza[3] (see section 2.3), is a tree structure. Hereby, every node is a task. A node can create a new task, which will be its **child**[4]. A parent node or short **parent** is the only communication partner of a newly created task. Therefore, it is a kind of reference monitor for all access requests, that are related to their children. Every request of a node *A* to other nodes, that aren't children of *A*, are mediated by its parent. Hereby, the parent additionally acts as a name service for its children.

To access resources, a node asks its parent to initialize a **session** to a named **service**. In fact, a session abstracts from the process of requesting a capability, that is associated with an communication endpoint to the service. Dependent on the child's identity and the requested service, the parent delegates the request, either to its own parent, it constructs a capability to a service of its own, or it invokes one of its other children, that implements that service. In any case, the parent delegates the resulting capability to the client. The procedure of a session request is shown in figure 3.1.

Of course, to establish sessions to other children, the parent has to know of their services, therefore children might "announce" them at their parents. While announcing a service, the parent gets a so called **root-capability**, which allows to request for new session capabilities,

[1] I present Bastei in this chapter, although it is not designed by me, but by different people of the operating system research group of the University of Technology Dresden. Nevertheless currently it is not 'state of the art', but of highly experimental nature. Both working processes (Bastei and this thesis) started nearly the same time and were closely coupled.

[2] Bastei is the German word for a kind of bastion. It is derived from the Italian word bastia. Beside that, it denotes a touristic sight in Saxony.

[3] Lately, Bastei is a part respectively further development of the Nizza architecture

[4] Please, do not confuse children and parents with their UNIX analogues.

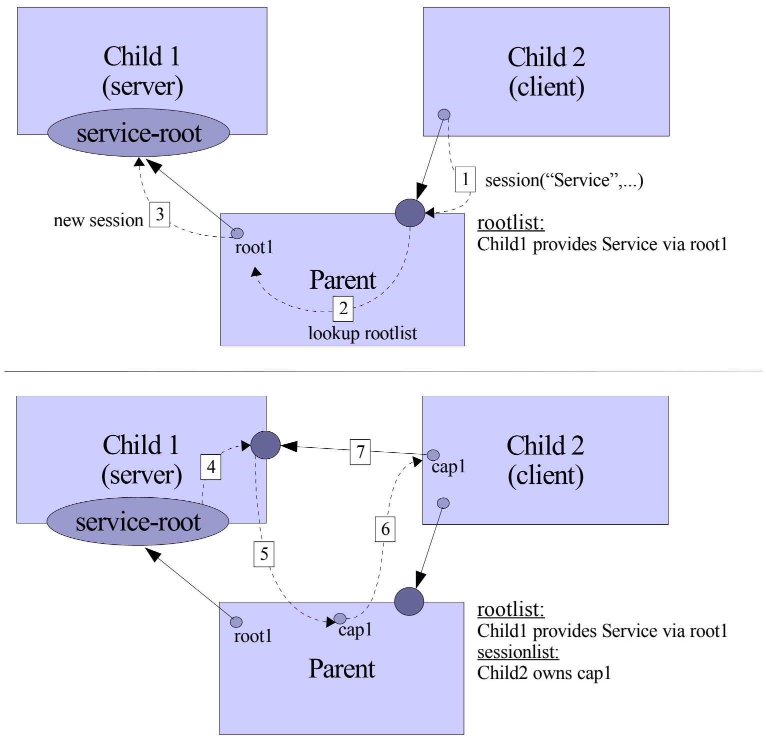

Figure 3.1: Service request of a child.

that reference that service. You can see the whole process of service announcement in figure 3.2.

After a successful session establishment, a child can use the resulting capability directly to reference the requested service, without a further indirection over its parent. On the service's side at the time of session establishment, a `Server_object` is created, that is addressed by the newly constructed capability. Every time the capability gets invoked, the `Server_object` will be invoked on the service's side. Of course, it must provide the service's interface. One example of a `Server_object` is the child representative on the parent side, which is simply called `Child`. This object gets invoked, every time a child uses its parent capability, for instance to request further sessions. `Child` implements the parent interface, which is defined as follows[5]:

```
interface Parent {
    exit([in] int exit_value);

    announce([in] String     service_name,
             [in] Capability service_root);
```

[5] This IDL-like notation is used to derive a client- and a server-side interface from it. Thereby, if a client invokes a function, it transfers by means of IPC all arguments, that are marked with the `[in]` tag, to server. The arguments, that are marked with the `[out]` tag, are transfered back to client, when the server responds.

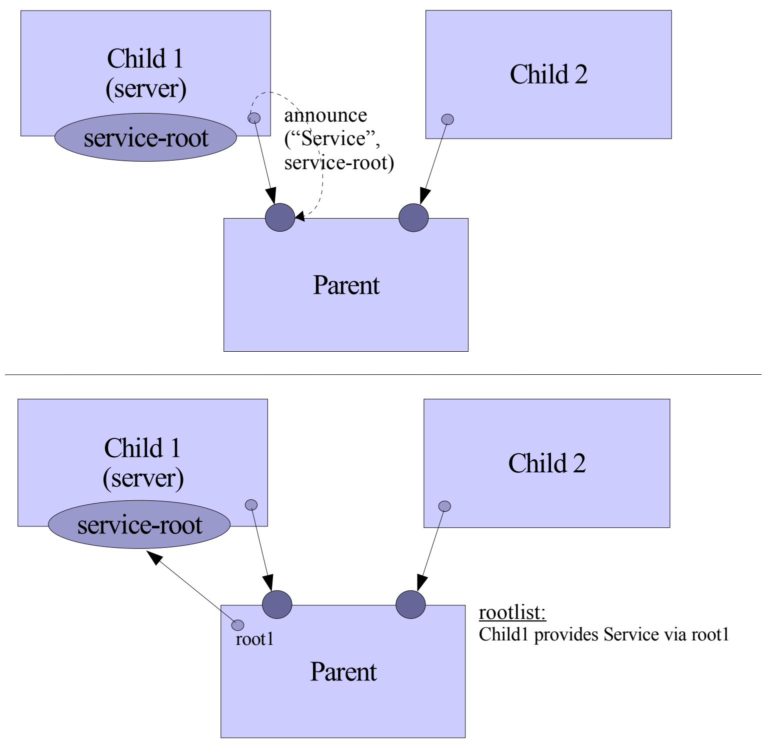

Figure 3.2: Announcement of a service by a child.

```
    session([in]  String     service_name,
            [in]  String     arguments,
            [out] Capability session_cap);

    transfer_quota([in] Capability session_cap,
                   [in] size_t     amount);
};
```

As you can see in the parent interface, when a child wants to establish a session, it can provide additional arguments, for instance if it needs a capability, that allows only limited functionality. Such session arguments can be evaluated and potentially changed by every node, which the session request traverses; it might help the node to find and further propagate a policy decision.

Equally, when creating a new session-capability by the root-capability, one might send additional information to the service, as you can see in the root interface:

```
interface Root {
    session([in]  String     arguments,
            [out] Capability session_cap);
};
```

The session argument string should contain sequences of tag-value pairs, like: *ram_quota=4K*. The previous argument example is used by parents, that handle memory quota for their children, and need to transfer quota from the requesting child to the target service, so that the service is able to perform the task at the expense of the client. Such a resource donation might help to prevent DoS attacks (compare to section 2.2.2).

The session request migrates through the Bastei tree, until the right service is found. Thereby, every node of the path, might change the session arguments, dependent on its policy. Figure 3.3 illustrates a migrating session request through a subtree and how the session arguments get modified, until they reach the target service. In this example, an application requests a new window widget from the service *GUI*. According to the application, the new window should be titled as *terminal* and should forward keystrokes, it receives. However, the parent of the application modifies the session request and demands, that the label should be the name of the user, on who's behave the application is running, followed by the programs name. After that, the request is send to the grandparent, that really controls the *GUI* service. This node contains a policy, that denies for that user to create listening windows, and therefore sets the *input* argument to *none*.

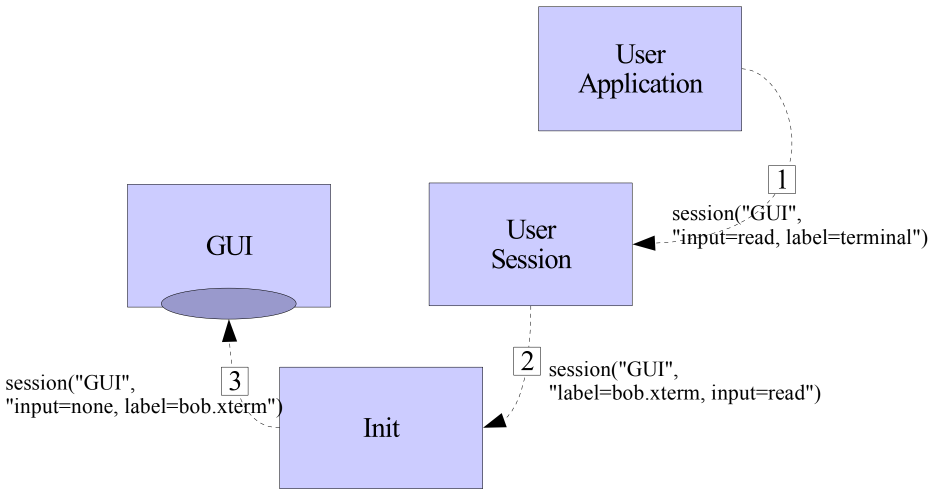
User Application
1
session("GUI", "input=read, label=terminal")
GUI
User Session
2
session("GUI", "label=bob.xterm, input=read")
3
session("GUI", "input=none, label=bob.xterm")
Init

Figure 3.3: Session request gets modified by parent nodes.

After this detailed introduction to the parent-client relationship and the whole process of session establishment, which is quite important for the following, because it builds the ground mechanism for the access control framework, we will focus our attention once more on the overall architecture of Bastei.

The root of the tree node is the **Core** component. Core controls every physical resource and provides a first abstraction of them. Core itself contains no policy interpretation logic and only provides services for all its resources to its only child: **Init**. From the point of view of access control Init is rather the root-node, because it's the first node, that handles policies. It starts all components, that are needed for the system to work and controls the communication relations between them. Furthermore, it controls session requests to the physical resources provided by Core, for instance it handles a RAM quota of all its children. Beside Init and Core, there exists the **Device Manager** as a third component, which runs on top of Init and handles, as the name suggests, device accesses. Beside these first components, there are some base session-protocols defined in Bastei, but we will not deal with them.

If you are interested in further information about Bastei, please have a look at[tud06], where soon you will find a detailed explanation. Bastei is written fully in C++. Until now, it runs on top of the L4v2 microkernel Fiasco, as well Fiasco UX and Linux. A port to L4.sec is work in progress.

3.2 Real-World Scenario

To demonstrate the advantages of recursive adjusted, small software components, that run on top of a microkernel, we want to look at a real-world example. Especially, we will see the positive effects one obtains, when using stacked security policies in contrast to one overall policy.

Imagine a company, society, project, or something similar[6] with a small LAN, that is connected to the Internet. The LAN connects the PC workplaces with an internal Web server, that is used for a Content Management System (CMS). In the CMS act different persons in different roles, for example the administrators or the bookkeepers. Of course, configuration data should be only available to administrators and bills only to bookkeepers; a typical example of a RBAC policy (see section 2.1.2.6). Furthermore, the persons want to set access rights for their personal data, so DAC (see section 2.1.3.1) needs to be handled too.

In addition, some data of the internal CMS Web presentation have to be available to the external Web server, that can be connected from outside of the LAN. Also, there might exist a mail server and some other Internet services. For security reasons, there should be at least a packet filter between the Internet and the public services and one in front of the LAN, to build a Demilitarized Zone (DMZ). Figure 3.4 illustrates the network topology.

With traditional, monolithic and complex operating systems a similar topology, particular the separation of components in hardware, might be necessary to minimize the risk of successful attacks from the Internet and the inner network, with respect to the public services. This results not only in a need for a lot of computer systems, but means a high administrative effort too, because you not only have to configure policies for Web server or packet filter, but also for each underlying system.

If you take systems like SELinux for instance, with thousands of rules in one monolithic policy, it is clear, that complexity gets difficult to handle and new sources of error arise. Even if you use virtual machines, you might reduce hardware complexity and costs, but the software complexity will still be high. Moreover, beside the policy complexity, the monolithic nature of traditional operating systems results in one complex security monitor with one complex decision engine, as long as it has to handle different kinds of policy types.

Now, lets have a look how this topology can be mapped into the Bastei architecture and what enhancements arise. Figure 3.5 shows a possible design.

You can see Init as the first component[7], which automatically initializes the Device Manager, an auditing task, the initial process of the DMZ and the CMS. We assume, that somewhere in

[6] I took inspiration by a small, alternative youth project in Dresden called Conni e.V., which is in the well known dilemma of finding a policy, that allows to be open to its clientel on the one hand, and secure the privacy of the individuals on the other hand.

[7] Core is left out for simplicity reasons and because it doesn't matter.

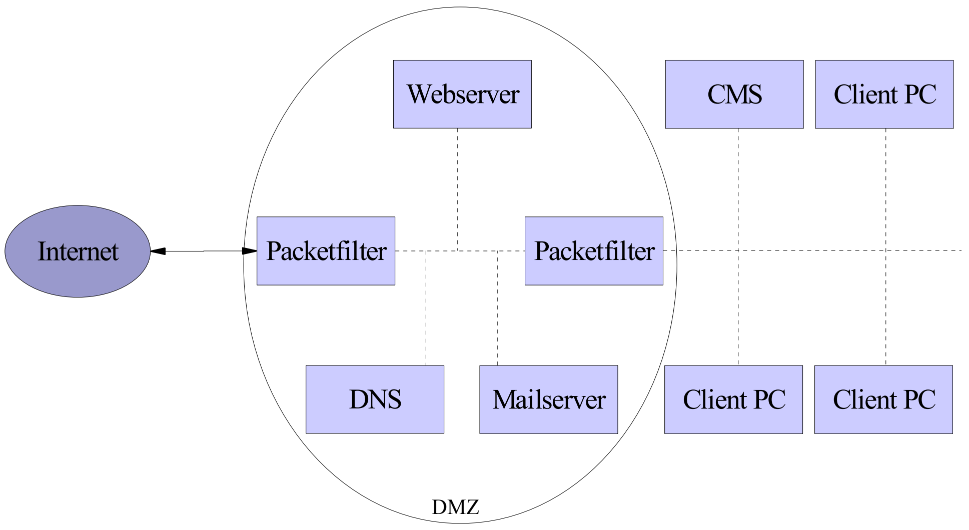

Figure 3.4: Demilitarized Zone example.

the Device Manager subtree there exist two Ethernet card driver components, one that is used as connection to the LAN's switch or hub, and one that is used as connection to a DSL modem or something similar, which represents the connection to the Internet. Now, Init contains a policy, which grants access from the CMS subtree to the first Ethernet card driver and from the DMZ subtree to the second driver. This is the equivalent to the hardware cabling of the former topology. Furthermore, access from the CMS to the DMZ and vice versa has to be allowed, and all components might use the auditing component, whereby only the CMS is allowed to access it with read operations, to allow administrators to analyze it.

The DMZ initial process acts as the former outer packet filter (meaning the one towards the Internet), because it allows some listening and connecting operations of its children with respect to its TCP/IP stack component. For instance, the Web server is allowed to listen to port 80. Moreover, it decides which connections of its parent Init, and therefore which connections from the CMS side are allowed. Of course, the TCP/IP stack is the only child, that has direct access to the DMZ's Ethernet card driver component.

The CMS initial process acts as the former inner packet filter, the one towards the LAN and in addition as the RBAC and DAC security kernel. Upon it runs a TCP/IP stack, a Web-getty, a document store respectively file-server, and two authentication components. The TCP/IP stack alone is able to access the CMS's Ethernet card driver, analogue to the DMZ subtree. Web-getty is a thin HTTP-server, that waits for new HTTP 1.1 connections, identifies users, and asks the CMS initial process to establish a new HTTP component, which from now on represents the user. For the authentication purposes two components might be used: one to identify the computer system (e.g.: with SSL) and one for the actual user. Of course, which components have to be used, depends on the actual policy. A user might authenticate itself for one of its roles. In this case, the user component request a new role component from its parent. Dependent on

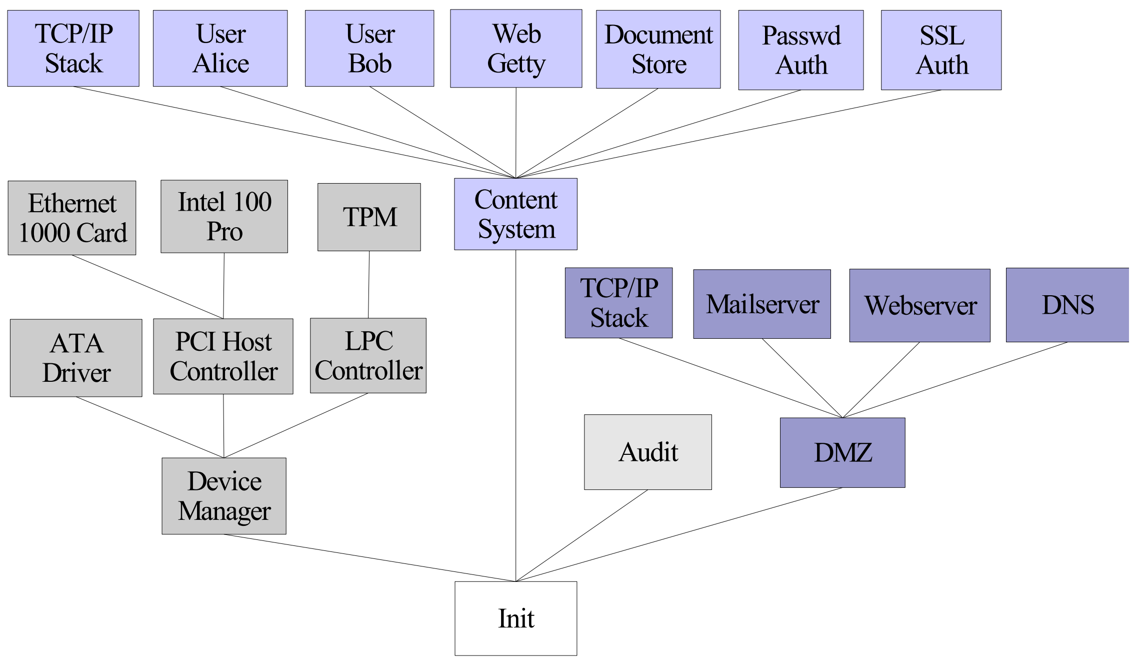

Figure 3.5: Scenario design in Bastei.

the user or role identity, a component is associated with, it can access documents from the document store. All these access rules are formulated in the CMS initial processes policy.

So, we don't need to look at this example in more depth. The explanations are sufficient to draw some conclusions. As one can see, beside the drastically reduction of the code basis this design implies, with respect to the traditional solution shown in figure 3.4, a nested structure of small components leads to a functionality and policy complexity of a size, the actual subsystems really need. This simplifies to model systems, that puts the principle of least privilege into practice. For example, when implementing the Init component, we don't need a policy engine, that knows of TE, RBAC or even authentication. A simple form of an Access Control Matrix[8] would be sufficient. Therefore, complexity of the components, closely to the root of the tree, can be kept low. On the other hand, a subsystem like the CMS, doesn't depend on the rules, that are defined in another subtree (e.g., the DMZ). In all centralized systems with one reference monitor and one access control policy, all components depend on this one policy. On the other hand the complexity of the policy depends on the number of components.

However, if you really want to build systems and components, which are highly configurable and only contain the functionality they need, its necessary to have components, which are interchangeable, and a framework, which simplifies the construction of new components, that fits the requirements. Bastei is such a framework, but it doesn't provide an abstraction to easy build up reference monitors and their client's and server's stubs. Therefore I have integrated a security

[8] Hereby, ACM doesn't mean, that it has to be implemented as a matrix, but we only need the simple abstractions subjects, objects and operations, no hierarchies of them and no further conditions.

framework into Bastei, on the basis of the todays requirements of security policies, which I want you to introduce to in the next section.

3.3 Monitor Framework

When designing the Reference Monitor or short **Monitor** framework, I followed up two approaches. The one was to build a structure of generic nature, one easily can map security policies to. On the other side, I had to search for appropriated constructs in Bastei, where decision engine and enforcement can take place. In this section I want to present the general policy abstractions and where they took place in Bastei. A detailed explanation of some protocols and procedures you will find in section 4.

3.3.1 Goals

The primary goal of Monitor is to provide a simple, generic framework with a tiny code basis, that simplifies the construction of different types of reference monitors within Bastei. To minimize complexity, the framework has to be organized in such a way, that a reference monitor uses only abstractions it needs to support the desired policy class. From this it follows that dependencies between framework components have to be minimized. It is desired, that components are most independent of security policies as possible, so a separation of policy and enforcement is aimed. To support a wide range of policy classes, the following properties are useful:

- **Naming:** A form of representation for subject, resource[9] and operation names or labels is necessary.
- **Rules:** We need a simple form of access rules, that define operations, which a nameable member of the policy is allowed to process with respect to another one.
- **Nested structures:** Subjects, resources or even operations might build a hierarchy, like for instance TE and RBAC require it.
- **Authentication:** A principal might has to be authenticated, before we bind a nameable policy member to it. Therefore, authentication methods have to be specified for certain subjects.
- **Adaptability:** Some policy classes, like Chinese Wall, ORCON and all policies, that require DAC, make it necessary, that the policy changes dynamically.
- **Information flow:** Subjects may have to be marked, what access they already performed, so that no illicit, indirect information flow occurs.
- **Rule conditions:** Further conditions might be useful, like allowed access time, allowed access duration or allowed access count.

These abstractions have to be organized in a package or module structure, with few as possible dependencies between them. In the following, we want to look at the basic package, all others depend on.

[9] From now on, we avoid the (in the security community) commonly used term object, to not confuse it with the object orientated constructs, that will follow.

3.3.2 Basic Abstractions

The base of the framework builds the `Security_context`, that represents a subject, resource or operation within a security policy. A `Security_context` possesses a label, which can be compared with each other. Furthermore, we have a `Rule` class. It contains a target context and a set of operation contexts. In addition, we can distinguish rules, whether they grant or deny access. Several rules form a `Rule_set`, which can be evaluated with respect to a requested target and operation set, whether access is granted, or not. A context, that owns a `Rule_set` is called `Subject`. Security contexts can be collected in a `Policy_database`, which can be used by a reference monitor to find access decisions.

Now, lets have a look, how the reference monitor concept gets integrated into Bastei. As we have already seen in section 3.1 and 3.2, every node in the components tree of Bastei potentially implements a reference monitor, with regard to its subtrees. Additionally, it is the name-server for its children. So, if a child firstly requests a service, this is done indirectly over its parent. In contrast to architectures like FLASK (compare to section 2.3.2), the reference monitor gets invoked first. At this point a policy decision has to be done and then further propagated to the requested service. The service constructs a capability for the client, which addresses a `Server_object` in the service's name-space. Now, it is straightforward to stick policy information to this object, meaning the information, what operations the holder of the capability is allowed to perform, with respect to the `Server_object`. This is somehow the analogue of the Access Vector Cache in FLASK (compare to section 2.2.1.1). The request procedure in Bastei in comparison to FLASK is shown in figure 3.6.

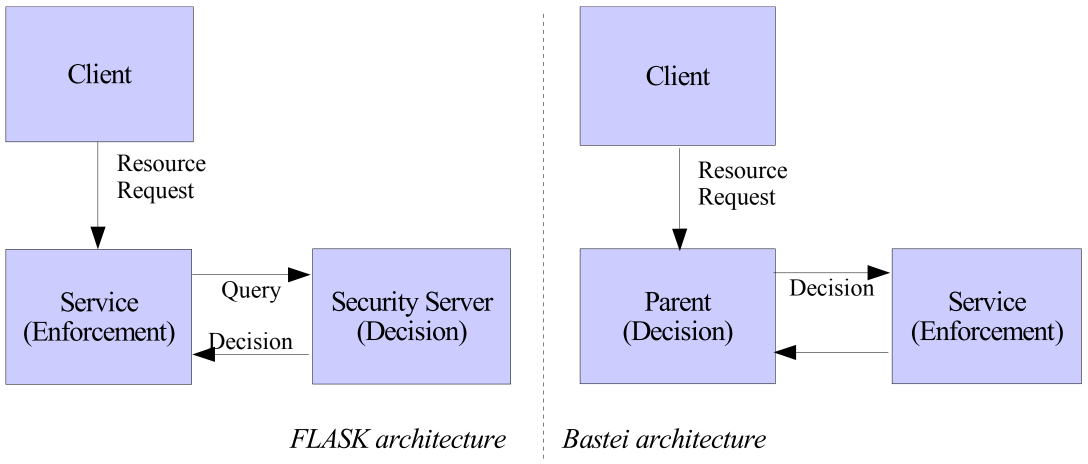

Figure 3.6: Resource request procedure in FLASK and Bastei.

Next we want to see, what particularly happens at the parent side. In Bastei, when a child requests a session-capability for a service, a the parent side the corresponding `Child` object gets invoked. So, it's straightforward to associate each `Child` with the appropriated `Subject` of the policy. If the session method (remember the parent interface in section 3.1) of the `Child` object gets invoked, we can use its `Subject` and its corresponding rules to decide, if access has to be granted or not.

Therefore, the requested resource within the service has to be identified and mapped to its appropriated security context. Furthermore, the child might have specified, which operations it would like to use with respect to the resource. That means, it wants only that operations to be attached to the resulting capability. For example, an user's shell might start a graphical viewer, that only needs read access to some document, then only the read operation should be associated with the requested session capability, also if the requesting user has more access rights, with regard to the target file. So, if operations are specified within a session request, that operations have to be mapped to their appropriated security contexts too, to make a decision possible.

Because resource and operation names are local to the specific services, the mapping of them to security contexts have to be specified by the service designer. For instance, if you would like to use an information flow policy, like Bell-LaPadula's one, first of all you have to categorize each service-operation as read, write or read-write operation; this can be done only by the service programmers, as they know the server logic. An Interface Definition Language (IDL), where developers mark operations appropriately, together with a compiler, that generates mapping policies, which can be used by the parent, might help in that case. Anyway, the mapping is local to the service and it's therefore logical to implement the mapping of resources and operations to security contexts in service representatives of the parent. Such objects already exists in Bastei and are simply called `Service`.

The Monitor framework provides an inherited `Service` class, which has two functions, in addition to its base class, with that you can map session arguments to resource and operation contexts. The whole process of access decision, when a child object gets invoked is shown in the sequence diagram in figure 3.7. For a clear arrangement, the invocation of the several rules within a `Rule_set` is left out.

Now, we have seen the procedure of a session request and the following access decision. If access is granted, the parent has to propagate its decision: what operations have to be attached to the session-capability. It propagates the operations, either to its own parent, or to the requested service, dependent on where the service is located. Therefore, again the session argument can be used.

The former explanations, described the general process of access decision and propagation. In the next sections, we will see how to construct reference monitors, that support more sophisticated policy classes.

3.3.3 Nested Security contexts

Most policy classes support hierarchies of subject labels, like roles and groups and also nested resources, to reduce the policy complexity. For instance, all subjects, with administrative competence within a policy, might inherited from role *admin*. Then, only for that subject the appropriated access rights needs to be set. The Monitor framework defines an additional package, that supports nested subjects called `Domains` and nested resources called `Types`, in analogy to Type Enforcement.

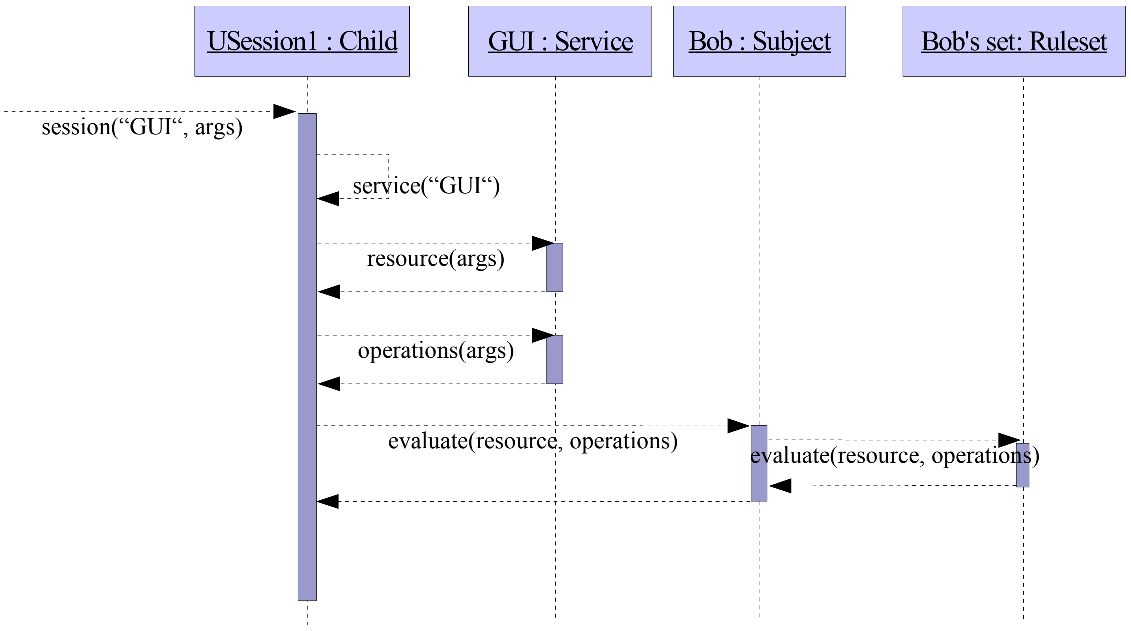

Figure 3.7: Decision making after a session request.

Both classes encapsulate a set with security contexts, from which they are derived, and a set with derivatives of it. Now, if a resource is requested by a child, the rules of its corresponding domain are checked, as well as the domains, this domain is derived from. When evaluating, which rule matches the requested resource, we compare not only the target type of the rule, but its derivatives too.

Take the security policy in figure 3.8 for example. We have the user Bob, who is an administrator. Administrators are allowed to read configuration data, whereby all policy data is classified as configuration data. If a component requests policy data on behalf of Bob, first the rules

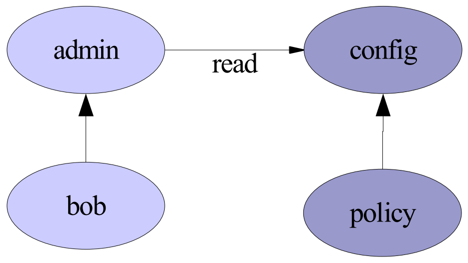

Figure 3.8: Example policy with nested security contexts.

of the *bob* domain are checked. No rules are defined, so the generalized domains are checked, in this case only *admin*. This domain defines the read rule for the *config* type, so the requested *policy* type is checked against it. Because the labels are not the same, the inherited types of *config* are checked and this time it matches and the access is granted for read operations.

3.3.4 Authentication

Every node, that allows the creation of new children at runtime and as well implements a policy, have to bind a security context to newly created children. In addition, some nodes have to support execution requests from other nodes, for instance if a child requests a new user session. Therefore, an extra package is provided by Monitor, for authentication purposes.

In general, when a child *C* wants to authenticate an identity, there is no difference to other requests. When *C* initiates an authentication session request, its parent *P* invokes the concerning authentication service *S*, after evaluating, that the child is allowed to use *S* in connection with the given identity. The service returns a capability for a newly constructed authentication object, which holds the requested identity. With that capability, the client is able to authenticate the identity.

After successful authentication, service *S* requests for a session to a so called offspring service of node *P*. This service allows to start new children with a specified identity. Therewith, *S* can request for the offspring service, it has to know of the service's name. Therefore, *P* transmitted that service name previously, when requesting for the authentication capability. If *S* is allowed to use the offspring service with the specified target identity, *P* returns a capability to an `Offspring_server` object, that encapsulates the identity. The authentication service returns that capability to its client *C* and closes the session to *C*.

Now, the client is able to send an execution request to the `Offspring_server` object, which firstly evaluates, if the encapsulated identity is allowed to execute the specified binary. You can see the whole process of authentication exemplary in figure 3.9.

This design of the authentication process enables us to specify, which instance is able to bring which identity into the system, thereby which authentication mechanism has to be used and lately what binaries this identity is able to execute. Moreover, using this protocol the parent node on who's behalf the new task is started, is able to construct the `Offspring_server` object at the expense of the requested identity's memory quota.

You may believe, this protocol is needlessly complex. Why not transferring identity and binary, that has to be started, within the initial session request? Then the parent is able to do the authentication and if that succeeds, it can execute the binary in a new child with the corresponding identity. But this second solution implies a potential DoS attack, as the parent always has to wait for the authentication to complete. In the introduced solution, the client itself has to perform the authentication; the parent only controls and mediates session requests.

3.3.5 Dynamic policies

Until now, we only considered static policies, that are defined off-line, but policies might change at runtime. For ionstance, if we want to support Discretionary Access Control or Chinese Wall like policies, we might add, remove or change rules. Hereby, the addition of more access rights is something simple, because it doesn't influence existing capabilities. But if access gets restricted, we possibly have to revoke capabilities or change their semantic.

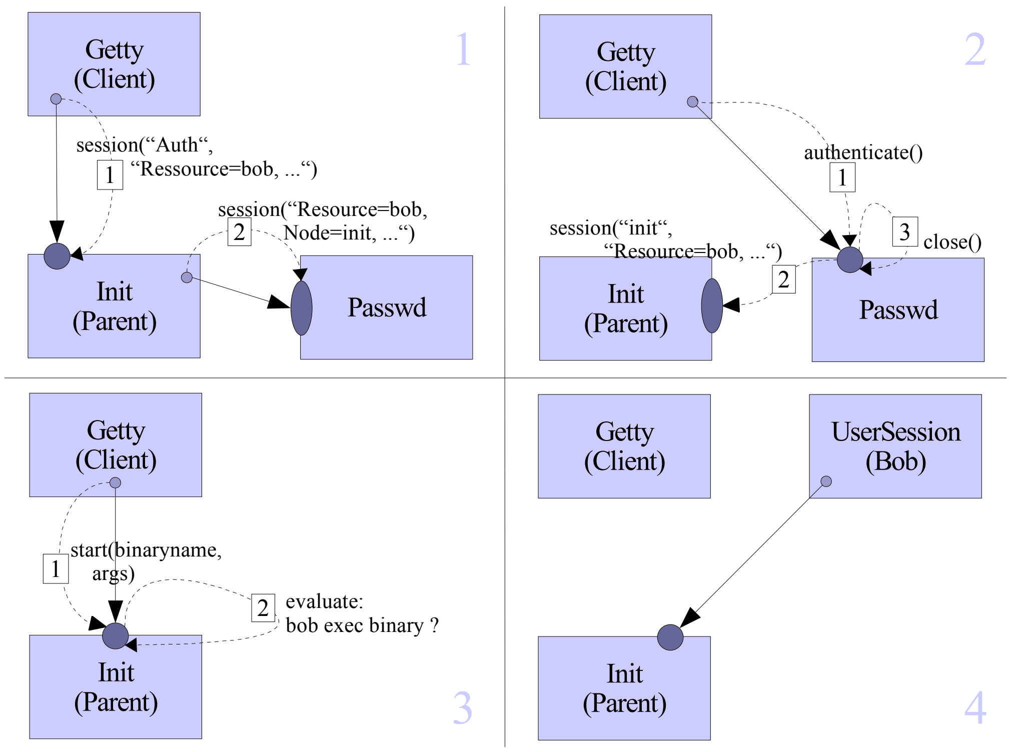
Getty (Client)
Init (Parent)
Passwd
session("Auth", "Ressource=bob, ...")
session("Resource=bob, Node=init, ...")
1
2
authenticate()
session("init", "Resource=bob, ...")
close()
3
start(binaryname, args)
evaluate: bob exec binary ?
UserSession (Bob)
4

Figure 3.9: Getty authenticates user Bob and starts a binary on behalf of Bob.

In general, there exists two possibilities. One is to simply close sessions, that provide access facilities, which are forbidden by the modified policy. The other possibility is to contact the corresponding services and to propagate to them, which rights have to be restricted. Of course, in this case the service respectively its `Server_objects` must provide an additional interface, the parent can use to send modifications. In addition, the parent has to save the capability, addressing the resulting `Server_object`, together with the information, what access rights it has propagated to that object. With other words, the parent has to track sessions, it established. As we have already seen in figure 3.1 (see section 3.1), there exists already a data structure for tracking sessions in Bastei: the session list. Every `Child` object associates such a list, which contains so called `Session` objects. For our purposes, these objects only need an additional attribute, representing the policy data, that was send to the service side. When a policy gets modified, the parent traverses its children's session lists and checks the decisions, it has made with respect to the modified policy. It invokes all `Service_objects`, that cache invalid access rights and transmit the appropriated rights.

Both presented solutions may cause an exception, if the client tries to access the `Server_object`, after its access rights were restricted. The only difference is, that if the session was closed, an exception gets thrown for sure. Whereby, if the access rights are modified on the server side, an exception only occurs, when the requested operation is forbidden by the modified policy. Both solutions might be used. The first one reduces server and parent complexity a little bit, with the overhead of requesting a new capability, each time a policy changes and a session gets closed. In both cases, the client needs to handle invalid calls and the parent has to track, which access rights it has attached to which sessions.

3.3.6 Sophisticated information flow policies

Policy classes, that need information flow tracking, like the ORCON model (compare to section 2.1.3.3), complicate the process of decision making. Such models require, that we check not only the access rights of the requesting subject, but consider in addition, to what resources the subject already has access to. If the requested target isn't permitted to access the resources, the requesting subject is already connected to, access should be denied. The other way round, if resources respectively other subjects, the requesting subject keeps sessions to, aren't allowed to access the requested resource, access has to be denied too.

Obviously, the process of access evaluation in the `Subject` class needs to be redesigned in a inherited class. Also a further attribute is needed: the set of security contexts the subject owns sessions to. It appears to be easy to enhance the framework to support ORCON, but we won't go into more depth here, because this policy class isn't the main focus of my work and therefore I didn't implement it.

3.3.7 Further conditions

At last I want to outline, how additional conditions of rules can be integrated into Monitor. Imagine, we want to specify, that access to a resource can be made only a confined number of times, or we want to limit the validity of capabilities to a specified period of time. In general, there is no problem to extend the policy constructs by such aspects. For instance, we can add a time-stamp and time period attribute to the operation's context, or we add a counter to it. Problems may arise, because the services might have to support these new constructs, as they enforce the access control. As long as we want to evaluate such conditions, each time a client invokes the service by its capability, the service needs to control the conditions. For instance, it has to increment the counter, or to check the time-stamp. This is somehow a dilemma, because we wanted to make the server independent from its policy and these conditions are part of the policy. A solution would be to only check such attributes, when a session gets established, or, if this is not sufficient, to use a proxy, that is addressed by the client's capability, which checks all conditions. After a successful checking, it propagates the client's calls to the service.

Applications in higher levels of the Bastei tree, that are constructed with regard to such conditional access rules, like DRM products, time aware applications, and so on, may use more sophisticated constructs of the Monitor framework at the outset. In general, it is possible to use more sophisticated components on top of a simple reference monitor and vice versa, to use a simple component on top of a reference monitor, that supports additional policy features. An important feature of the framework is, that the derived, sophisticated constructs in Bastei interact correctly with the basic forms of them. How this is done, we will see in the next section, where some implementation details will be explored.

4 Implementation

In this section, I will explain some details of the implementation, especially concerning the encoding of policy data. Furthermore, you will learn how to use the framework classes to implement reference monitors and services, that are under control. I will discuss dependencies, that has to be noted, when you want to enhance the framework. Therefore, we will look into the implementation of DTE. At the end of this section, the construction of authentication services is explained. As the whole Monitor framework is written in C++, a basic understanding of this language is helpful to understand the following.

4.1 Serializing policy objects

Section 3.3.2 gives a quite detailed overview to the controlled process of session establishment. But it does not cover, how the required policy information is encoded in the session argument string and decoded at the receiver's side. Of course, we need some general mechanism and not a service specific implementation, so that each node, the session request traverses, is able to interpret that information. As already mentioned, we need to transfer the target resource and potentially operations, the client wants to use with regard to the resource, within each session request. Therefore, we use the suggested scheme of tag-value pairs (compare to section 3.1), with the tags: *Resource* and *Operation*. The value part consists of an XML representation of the particular resource's `Security_context` and `Operation_set`.

In general, you can serialize each class of the Monitor framework, that represents a policy part, to an XML string. To encode policy data in a human readable manner has several advantages. First of all, you can easily express security policies, but it helps to interpret auditing entries, or the actual used policy at run-time. The XML representation of a Monitor object contains for each relevant attribute a corresponding tag. You can see the XML representation of the basic Monitor classes in figure 4.1.

Monitor construct	**Related XML scheme**
Security_context	`<ID>`*String*`</ID>`
Operation_set	`<Operation>`*Security_context*`</Operation><Operation>...`
Rule	`<Resource>`*Security_context*`</Resource>` *Operation_set*
Rule_set	`<Rule>`*Rule*`</Rule><Rule>...`
Subject	*Security_context Rule_set*
Policy	`<Subject>`*Subject*`</Subject><Subject>...`

Figure 4.1: Mapping of policy objects to their XML representation.

To emphasize the simplicity of this scheme, look at following sample policy, which defines access rights for *Alice*. She is allowed to read the document *great_plan* and further to read and execute the binary *shell*.

```
<Subject>
   <ID>alice</ID>
   <Rule>
      <Resource>
         <ID>great_plan</ID>
      </Resource>
      <Operation>
         <ID>read</ID>
      </Operation>
   </Rule>
   <Rule>
      <Resource>
         <ID>shell</ID>
      </Resource>
      <Operation>
         <ID>read</ID>
      </Operation>
      <Operation>
         <ID>exec</ID>
      </Operation>
   </Rule>
</Subject>
```

It is quite easy to serialize policy information for persistence reasons or to transmit it by IPC to a service. Every class, whose objects needs to be serialized, implements the following interface, respectively inherits from the following class:

```
class Serializable {
public:
   virtual int to_string(char *buf) = 0;

   virtual int length(void) = 0;
}
```

Hereby, the `length` method delivers the resulting string length and `to_string` writes the XML data to the given buffer. To construct objects by their XML representation, we have an additional class called `Context_factory`. Such a factory has to be defined for each relevant combination of framework classes, that represents a part of the policy. For instance, we have a factory defined for the basic framework classes (see section 3.3.2) and one for using domains and types (see section 3.3.3). These factories are used by parent nodes and services to analyze session requests. Parents also use it to parse their initial security policy.

For the basic classes of the framework there are even two different factories defined: one for parents, that act as reference monitors and one for services. The first factory uses a

Policy_database, when it analyzes a security policy or a part of it. When parsing the security context of a resource, operation, or subject, it firstly looks up in the database, if the context is already in there. Otherwise, it puts the parsed context in the database. A central database here is useful, to prevent duplicates, because contexts reference other ones within their rules, or when they are nested (compare to section 3.3.3). On the service side, mostly a database is not needed. Therefore the second factory simply constructs Security_contexts and Operation_sets from XML strings.

By transmitting a whole object in its serialized XML form, we are able to transmit more relevant policy data, than just the name of resources and operations. For instance, we are able to associate each operation context with a time-stamp and a validity time range. This time information will be transfered within the session string, when the operation context gets serialized. A service, that receives the session request and is able to interpret the additional information, can enforce more fine grained access control.

By using an XML scheme, one obtains the advantage of compatibility between different framework constructs in different nodes. Every node, only analyzes the XML tags of a resource context or operation set it understands. Take the example with the time-stamps in operation contexts. A parent node, that implements such a policy model, will transfer the time-stamp information within the session argument string each time it requests a service, regardless of, whether the service is able to interpret the information or not. But, if there are services without time dependent access control enforcement, because it's not implemented nor needed, that doesn't matter, because they will ignore a time-stamp tag in the operation contexts. The other way round, extensions of the framework need to be carefully designed, so that they will interpret simple policies and policy parts, that miss the new extensions, further on correctly.

4.2 Implementing controlled services

When implementing a service, one has to define an interface, from which the client and server side will be derived, like the Root or Parent interface in section 3.1 for instance,. On the client side, to use that interface, the client firstly needs a session to the server. Therefore, Monitor contains a class called Service_client_executive, that encapsulates the process of session establishment in it's constructor. The client stub of a service should inherit from it. This class provides the client side of an IPC channel, after a successful session establishment. A derived class can use this so called IPC_client object, which contains the session capability, to transmit an operation code, that represents the desired function, and further arguments to the server, as well as receiving results from the server. To facilitate the understanding, I want to give an example. A service, that controls a terminal has the following simple interface:

```
class Console {
protected:
   enum Opcode {MESSAGE,WARNING};
public:
   virtual void print_message (char *str) = 0;
   virtual void print_warning(char *str) = 0;
};
```

The client stub is defined as follows:

```
class Console_client : public Console,
                       public Service_client_executive {
public:
   Console_client() :
      Service_client_executive("Console", "tty1", "4k") {}

   virtual void print_message (char *str) {
      *server() << MESSAGE << Buffer(buf) << IPC_CALL;
   }

   virtual void print_warning(char *str) {
      *server() << WARNING << Buffer(buf) << IPC_CALL;
   }
};
```

Hereby, the function `server`, which is inherited by `Service_client_executive`, returns a pointer to the `IPC_client`, the channel to the server. As you can see, first we put the operation code into the channel, after that the string argument, and lastly the code for an IPC call.

To be able to establish a session at all, the constructor of the `Service_client_executive` class expects the service's name (*Console*), the resource's name (*tty1*) and the memory quota (*4K*), we want to donate to the server. Optionally, one could provide an array of operation names and a further string argument to the constructor. Within the constructor, the names are used to construct an appropriated session argument string and to request for the session at the client's parent. The session call in this example would look like this:

```
session("Console","Resource=<ID>tty1</ID>, ram_quota=4K");
```

When no valid capability is returned and therefore no communication can be established an exception gets thrown. Otherwise, the capability is used to construct the `IPC_client` object.

Now, we want to look at the server side of a service implementation, that enforces access control. As already mentioned in section 3, for each session there is a corresponding `Server_object` at the server side, that has to support the service's interface. In general, a class, that inherits from `Server_object`, has to implement a `dispatch` function, which "unmarshalls" arguments from the IPC channel and calls a function of the service's interface, based on a given operation code.

The Monitor framework provides a so called `Server_object_executive` class, which is derived from `Server_object` and has as additional attributes a resource and operation set. In its `dispatch` implementation, first it checks if the client is able to proceed the requested operation, that is known by the given operation code. The definition of this method is as follows:

```
int dispatch(int op, Ipc_istream &is, Ipc_ostream &os) {
   if(!_valid_ops[op])) {
      PWRN("unallowed access request detected");
      return 1;
   }
   return _hook_dispatch(op,is,os);
}
```

Hereby, `_hook_dispatch` is a *hook*-function[1], that need to be defined by derived classes. It has to do the “unmarshalling” work and invokes the appropriated function based on the given operation code. But before this function is called, the server object evaluates by an array of Boolean values called `_valid_ops`, if the desired operation is allowed. This array gets initialized in the constructor, on the basis of the transmitted operation set. As we have no previous knowledge about the count of operations and their names, these information must be provided by derived classes. The count of operations one specifies by a template argument, as the names are defined by a string array, one has to provide, when invoking the constructor.

The constructor of the `Server_object_executive` class is defined as follows:

```
template <int OP_COUNT, typename RESOURCE, typename OPERATION>
class Server_object_executive : public Server_object {
public:
   Server_object_executive (RESOURCE                *res,
                            Operation_set<OPERATION> *ops,
                            const char              *op_names[])
         : _res(res), _ops(ops) {
      for(int i = 0; i < OP_COUNT; i++) {
         if(_ops->find(op_names[i]))
            _valid_ops[i] = true;
         else
            _valid_ops[i] = false;
      }
   }
   ...
};
```

As one can see, the constructor looks up each operation name in the given operation set and if it gets found, the concerning entry in the array is set to *true*, otherwise to *false*. The encapsulated resource can be used by a service to determine, what resource itself represents with regard to its client.

So, a service designer only needs to define the `_hook_dispatch` function and doesn’t need to worry about access control. Again let’s consider the terminal example:

```
static const char *opcode[] = {"print","warn"};

class Console_server : public Console, Service_object_executive<2> {
protected:
   int _hook_dispatch(int op, Ipc_istream &is, Ipc_ostream &os) {
      Buffer args;
      is >> args;
```

[1] Such functions are also often denoted as template methods, derived from the design pattern: template method pattern[wik06]

```
      switch(op) {
      MESSAGE:
         print_message(args.addr());
         return 0;
      WARNING:
         print_warning(args.addr());
         return 0;
      }
      return 1;
   }
};
```

This is everything, that needs to be defined at the server side, beside the actual functionality. Of course, by using a concerning IDL and IDL compiler, this work could be done automatically.

4.3 Implementing reference monitors

Next, we will see, how parent nodes need to be modified, to take advantage of the framework. I have already mentioned, that nodes, which enforce a specific policy, need a database and an appropriated `Context_factory` to read in their initial policy and to analyze session requests from their children. Moreover, they need to associate each of their children with a subject of their policy. The Monitor framework provides a class called `Child_executive`, that encapsulates such a subject. Moreover, it already implements the process of evaluating if a session is valid and the following construction of the session argument string, that has to be send. You can see the implementation of the `session` function in the `Child_executive` class here:

```
virtual Capability session(const char *service_name,
                           const char *args) {
   ...
   if(_controlled_service(service_name)) {
      ...
      if(!_session_valid(service_name, args, buf, len))
         return Capability();  //return invalid capability
      return _hook_session(service_name, buf);
   }
   return _hook_session(service_name, args);
}
```

As you can see, when the function gets invoked, firstly a *hook*-method is called, which delivers if the desired service is one, that is under control of the policy. By default, this method returns true for all services, accept the services of the Core node. Of course, this method can be overwritten by derived classes. If the service is under control of the policy, the function `_session_valid` is called, which evaluates by the encapsulated subject's rules, if the session request is valid or not. The process of evaluation is the same as in the diagram in figure 3.7. When the request is valid, the function fills the given buffer with the correct session arguments, that has to be used to contact the service. If the session request is valid or the requested service is not under control of the policy, the `_hook_session`-method gets invoked, which has to do the actual work of session establishment. Because this process is specific to the node, this function is pure virtual and has to be implemented by a specific, derived class.

4.4 Domain and Type Enforcement

Next, I would like to introduce you to the modifications, that have to be done, if we want to enhance the defined security contexts, exemplary by the implementation of domains and types (see section 3.3.3). Of course, the modifications depend on what kind of attribute and behavioral changes are needed, but they all have some rules of thumb in common.

Firstly, if you add additional attributes to a security context class, you need to redefine the `to_string` and `length` methods, therewith the attributes can be made persistent and transferable. In addition, you will have to implement a new context factory (compare to section 4.1). Secondly, when defining a new class of resource contexts you will need to define a new class of subject contexts too, as subjects always have to be derived from the resource contexts. That is because subjects can also be referenced as resources. For an example, let's look at the implementation of nested security contexts.

The class diagram in figure 4.2 shows the coherence of the basic security contexts for resources and subjects and their derivations for hierarchical contexts. As you can see, the simple `Subject` class is derived from `Security_context` and the `Decidable_context` class, which encapsulates the rules. The DTE package within the Monitor framework provides in addition a `Nested` class, that contains a set of specific contexts and a set of generalized contexts. This class already implements the `Serializable` interface. The class `Type` represents nested contexts for resources. Nested subject's contexts are provided by class `Domain`, whereby this class is derived from `Type` and from `Decidable_context`. The classes `Type` and `Domain` have to overwrite their *to_string* and *length* methods[2], but that's no problem, because they only need to combine the implementations of their base classes, as the following example shows:

```
virtual int length (void) {
   return Security_context::length() + Nested::length();
}
```

This is the definition of the `length` function in the `Type` class.

I don't want to go into more depth in how nested contexts are serialized, but you need to know, that a DTE policy can define two more attributes of resources and subjects in its XML representation by the tags: *GeneralizedContext* and *InheritedContext*. Within these tags one has to specify an ID of an already defined resource or subject. When the parser finds such a tag, it searches for the specified ID in the database and if it finds a corresponding context, it puts that one into the set of generalized or specific contexts.

Remember the sample policy in figure 3.8. The XML form of it, looks like this:

```
<Resource>
   <ID>policy</ID>
</Resource>
```

[2] Not least because of problems with multiple inheritance, as both base classes provide partially the same interface

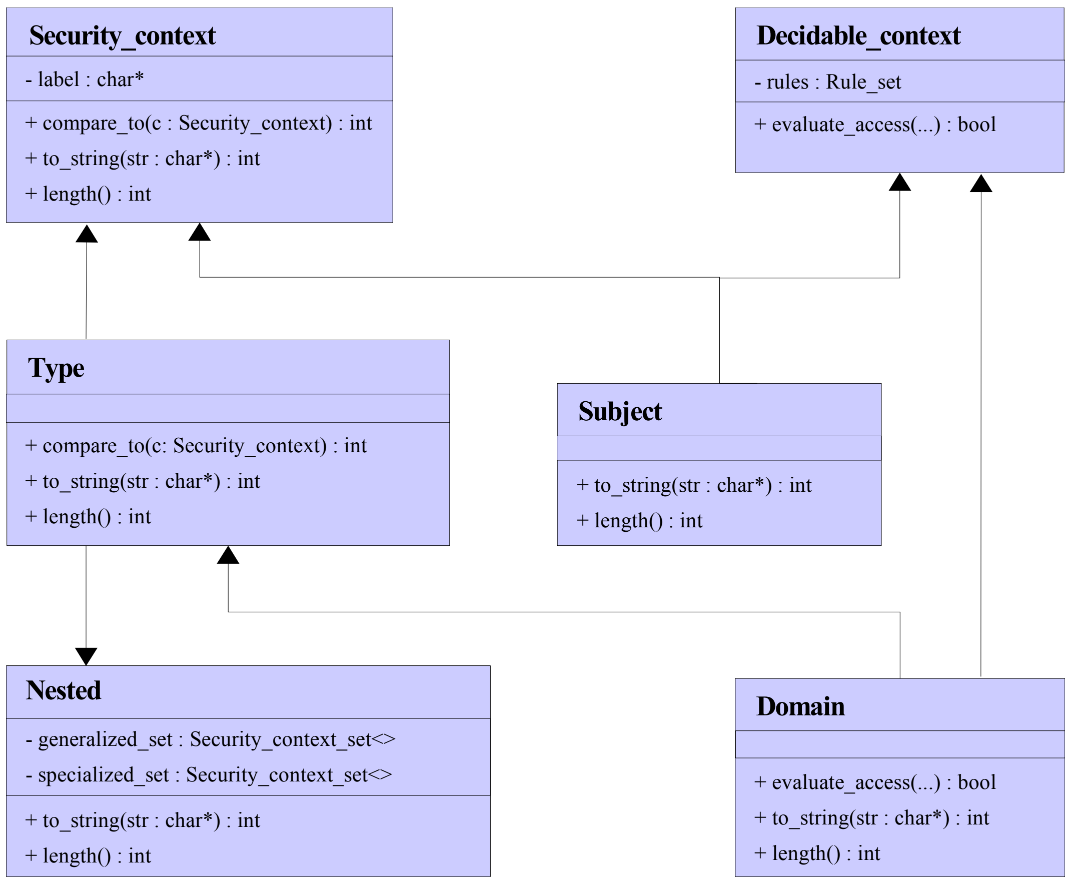
Security_context
- label : char*
+ compare_to(c : Security_context) : int
+ to_string(str : char*) : int
+ length() : int
Decidable_context
- rules : Rule_set
+ evaluate_access(...) : bool
Type
+ compare_to(c: Security_context) : int
+ to_string(str : char*) : int
+ length() : int
Subject
+ to_string(str : char*) : int
+ length() : int
Nested
- generalized_set : Security_context_set<>
- specialized_set : Security_context_set<>
+ to_string(str : char*) : int
+ length() : int
Domain
+ evaluate_access(...) : bool
+ to_string(str : char*) : int
+ length() : int

Figure 4.2: Class dependencies of security contexts.

```
<Resource>
   <ID>config</ID>
   <InheritedContext>policy</InheritedContext>
</Resource>
<Subject>
   <ID>admin</ID>
   <Rule>
      <Resource><ID>config</ID></Resource>
      <Operation><ID>read</ID></Operation>
   </Rule>
</Subject>
<Subject>
   <ID>bob</ID>
   <GeneralizedContext>admin</GeneralizedContext>
</Subject>
```

To achieve a correct decision making with domains and types, the `compare_to` method is redefined. Now, if a type gets compared against another one, its inherited contexts are included in the decision. The method is defined as follows:

```
virtual int compare_to(const Security_context &c) {
   int ret = strcmp(this->name(), c.name());
   if(!ret)
      return ret;
   return (Nested::compare_to(c)) ? ret : 0;
}
```

As you can see, only the specific contexts of the type, we apply the compare function to, are considered. This is done, because when checking if a rule has to be applied to a request, we always have a specific resource, that is requested and a potentially more general resource, as part of the rule. So, we only need to check the specific contexts of the resource within the rule.

The process of deciding, whether a requested resource can be accessed, or not, is redefined in the `Domain` class. The overwritten `evaluate_access` function traverses the rules of the domain, as well as the rules of the generalized domains. Thereby, it collects every operation, granting rules define in a set, and all operations, that denying rules define in a separate set. At the end, it returns the complement of the denying rule set in the granting rule set.

4.5 Authentication and Offspring service

When implementing authentication services, the scheme, suggested in section 4.2 and 4.3, cannot be used. If a parent node requests an authentication service for a session, on behalf of one of its children, it has to transmit not only the desired identity and allowed authentication operations, but also the name of its own offspring service (compare to section 3.3.4). This is done, to enable the authentication service to create a new offspring at the requesting node's side, when authentication was successful.

The needed functionality is provided by a derivation of the `Child_executive` class called `Child_authenticator`. It redefines the `_session_valid` method as follows:

```
virtual bool _session_valid(const char *service_name,
                            const char *args,
                            char *out_args,
                            int len) throw(Buffer_length_exception) {
   if(!Child_executive<R,O,S>::_session_valid(service_name, args,
                                              out_args, len))
      return false;
   if(_hook_auth_service(service_name)) {
      if(!Arg_string::set_arg(out_args, len, "Node", _offspring_service))
         throw Buffer_length_exception();
   }
   return true;
}
```

This implementation maintains the functionality of its base class, but adds a *Node* argument to the session string, when an authentication service is requested. The value of this argument has to be the offspring service name of the parent and it must be given to the constructor of the `Child_authenticator` class.

The `_hook_auth_service` function again is a *hook*-function, a derived class has to implement. It decides, whether the given service name is the name of an authentication service or not.

As an authentication service has a somehow foreknown interface, the Monitor framework provides a ready to use server object for such services called `Authentication_server`, which is derived from the `Server_object_executive` class and the following class:

```
class Authentication {
protected:
   enum Opcode {AUTH,REMOTE};

public:
   virtual Capability authentication(void) = 0;

   virtual Capability remote_authentication(Capability channel) = 0;
};
```

The first function should be used, when the necessary authentication data can be accessed directly by the authentication service. For instance, when the authentication service provides a password field in a window of its own behalf, where the user can type in his or her password. The second function has to be used, when the authentication data cannot be accessed directly, but is provided by a communication channel. For example, when a user tries to authenticate itself from another system out over a TCP connection. In this case a capability for the communication channel, for example a capability for the corresponding socket, has to be given as parameter. Regardless of which method is used, the client gets a capability to an offspring server object of its parent as return value, as long as the authentication was successful.

The whole process of “unmarshalling” the IPC message and invoking the right function, as well as the session request to the offspring service, who’s name was transmitted prior to the service, is already implemented by the `Authentication_server` class. A service designer only has to fill the following two *hook*-methods with functionality:

```
virtual bool _hook_authentication(const char *identity) = 0;

virtual bool _hook_remote_authentication(const char *identity,
                                         Capability channel) = 0;
```

The interface of an offspring service is also known in advance, because we only need one function `start`, that takes a binary name and further arguments and starts the binary on behalf of the prior requested subject, as long as the subject is allowed to execute that binary. Similar to the `Authentication_server` class, there exists a ready to use `Offspring_server`, which does the whole work including the check, if the binary is accessible by the subject. The actual procedure of starting a new child is encapsulated by a *hook*-method, as this process depends on what child class the parent uses. The `_hook_start` method is invoked together with the subject's security context, the binary name, and further invocation arguments as parameters.

5 Evaluation

In this section I evaluate the Monitor framework with respect to generality and complexity. Moreover, I consider some aspects related to the resource consumption, caused the framework. At last, open issues and enhancements of the framework, that need to be done, are discussed.

5.1 Generality

In the context of this work, generality of the framework means the ability to put any existing policy model into practice. Also, if this goal is not reached by the existing implementation, it will be reached by some further enhancements. The current version supports with its base package an implementation of the Access Control Matrix, as well as the DTE model (compare to section 2.1.2.3) in an extra package. The rules of both models are mandatory and of static nature and cannot be modified by subjects. Regardless of the actual policy model, the framework supports an authentication mechanism to link identities to instances of the system.

Although, the Access Control Matrix and DTE model are in fact very powerful and cover a wide range of different policies, some properties are still missing. That are: dynamic rule modifications, DAC and history tracking.

In fact, these three properties are partially mutually dependent: The ability of changing rules at runtime is required to implement policy models, like Chinese Wall. In addition it's a prerequisite for DAC. To support this feature, the framework has to provide session tracking and an capability revocation or modification mechanism, as already outlined in section 3.3.5. DAC would require a second policy database held by the reference monitor node, containing the discretionary rules and an additional service, that facilitates authorized clients to modify these rules. The third missing feature is the tracking of "who has or had access to what resource", as it is required for models like Chinese Wall and complex information flow policies. The necessary modifications to support history tracking are outlined in section 3.3.6. The broached additional features, should be organized in separate packages, so that we can combine them to new packages, supporting actual policy models. For instance, we could combine a DAC package with the base package to design the Bell-LaPadula model. In general, it seems to be relatively easy to complete the framework and probably requires only minor or no changes to the existing part.

From the present point of view, there are no clues, that the design of the framework restricts us to certain policy models.

5.2 Complexity

When evaluating security relevant software, one of the main quality attributes is complexity, because the rate of vulnerabilities in software is correlated with its complexity. To measure the complexity of the framework we will take Source Lines of Code (SLOC) as scale. All data is measured with *sloccount*, a tool written by David A. Wheeler[Whe06].

In figure 5.1 you see the current framework packages and their sizes in SLOC, whereby each package is split in a part specific to the parent node, that represents the reference monitor, and a service part, that enforces the access control. As one can see, the total amount of source lines

Monitor package	**SLOC**
Base	1184
Parent	1032
Service	992
DTE	365
Parent	365
Service	0
Authentication	319
Service	180
Parent	165
Total framework	**1868**

Figure 5.1: SLOC of the Monitor framework.

is fewer than 2000, which is quite less. Of course, we have to count the parts of the Bastei framework, these packages depend on, too. Furthermore, these values aren't expressive as long as we have no size measuring of an example, that compounds the framework classes to a working application. Therefore, I've implemented two reference monitors. One of them can be used as a simple *Init* process, like that in the scenario of section 3.2. It supports a simple Access Control Matrix model by using the base package of the Monitor framework. The other one can be used as the *CMS* component of the same scenario, whereby this application supports Domain Type Enforcement and authentication. The source line numbers of both applications are listed in the table in figure 5.2. You see, the necessary "glue", which compounds the framework

Application component	**SLOC**
Bastei	≈ 4600
Parent (Base)	1032
Parent (DTE + Authentication)	1562
Init	143
CMS	216
Init total	≈ 5780
CMS total	≈ 6380

Figure 5.2: SLOC of example reference monitors.

constructs to an usable application, is negligible with regard to complexity measurement. Both applications consists of approximately 6000 SLOC. For these two examples the difference with regard to complexity is still small. The *CMS* node has about 600 SLOC more than *Init*. If we add additional packages to the *CMS* node, as needed for DAC for example, the difference in size becomes more significant. This fact reinforces the design decision to organize different policy properties in mutually independent packages, which can be combined in new packages to build a specific policy model. In contrast, using only one library for all nodes has the advantage of full compatibility, but by the price of, whether missing features and therefore no generality, or by an increased complexity.

The size of the source code is only one aspect of complexity. Another one, specific to this work, is complexity of the policy. Of course, the complexity of a policy depends on the complexity of the whole system, meaning how many components and identities have to interact and how open or restrictive the system has to be. Nevertheless, by structuring the system in a way, like Bastei does it, one obtains a tree of components, with the applications most crucial to security nearest to the root. As the policy of one node only depends on its ancestors, the complexity for the most security sensitive policy parts, which relies near to the root of the policy tree, is reduced in contrast to one monolithic policy.

Moreover, such a design enables a developer to configure base systems, that can be used for a variety of operational areas with proven security properties. At the same time, these systems can be used to run component subtrees with individual policies upon them, that don't interfere with the base system. For instance, it will be easy for users to define their own mandatory policies, without the need of understanding a sophisticated overall system policy, as for instance in SELinux.

Let's have a look on the scenario of section 3.2 once again. There, the *CMS* and *DMZ* subsystem are initialized by the *Init* node. We would implement such a system with fixed and proven policies for the *Init* and *DMZ* node, that simply can be used by organizations, which only need to define a policy for the *CMS* node with regard to their organizational structure and needs.

To sum up, one can state without proving it by numbers, that complexity of more security sensitive components can be reduced, by using a nested structure, which represents the security sensitivity of the components, as in the Bastei architecture. To give a factor of reduction, we will have to implement a fully working scenario first, that can be compared against an appropriated SELinux configuration, for instance.

5.3 Overhead

Next, I examine the overhead, that is caused by using the framework. Although, it is probably the most interesting point, I will not present any performance measurements, because it wouldn't be expressive currently. Firstly, the Bastei architecture isn't ported to the L4.sec security kernel yet. This lack of kernel-supported capabilities has a great impact on performance. Furthermore, a simple measurement of one capability allocation or usage wouldn't be expressive. Again, we will need a fully working scenario, that can be compared to the same scenario running on existing systems. Moreover, some kind of macro-benchmark has to be defined.

However, I expect only little overhead by the access control decision process of one node, but more loss of performance by the several indirections, that need to be done when a client requests for a capability for the first time. Hereby, the overhead is dependent on the distance of the client to the service in the components tree.

5.3.1 Memory

Beside the computing effort, the additional memory usage is of particular interest. Figure 5.3 shows the object sizes of some important framework classes in bytes. For the child and service class you see the difference with regard to their base classes, which are defined by the Bastei framework. The memory overhead for children and services can be neglected, as long as a node needn't to handle thousands of them, which won't be the case normally. Now, let's have a look how much memory a policy occupies.

Monitor class	**Memory consumption in bytes**
Security_context	28
Rule	36
Subject	44
Type	44
Domain	60
Child_executive	(+)14
Service_executive	(+)4

Figure 5.3: Memory consumption of some Monitor objects.

Assumptive, a policy consists of the set of subjects S, the set of resources R, the set of operations O, and the set of rules P. The average length of the labels in S is denoted with s, the average length of the resource labels is r, and the average length of the operation names is o. Furthermore, the function $count(x)$ delivers the number of elements in x. The memory consumption of a policy, using only the simple abstractions of the Monitor base package, is calculated by the following formula:

$$count(S) * (4s + 44) + count(R) * (4r + 28) + count(O) * (4o + 28) + count(P) * 36.$$

A policy comprising of 6000 rules, 100 different subjects, 5000 resources and 500 operations with an average label and name length of 6 characters, consumes 496 kilobytes of the reference monitor's memory. However, this is an extreme example for nodes, that control a file-system or something similar.

The memory consumption of a DTE policy is calculated by the formula:

$$count(S) * (4s + 53) + count(R) * (4r + 44) + count(O) * (4o + 28) + count(P) * 36.$$

The above example would result in 550 kilobytes, when a DTE policy is involved.

5.4 Open issues

At last, I want to discuss open issues, that exist beside those, already discussed in section 5.1. One important enhancement is an additional language or a language extension, that helps to define the mapping of service specific resource and operation names to members of the policy database. Currently, there exists no real mapping and a policy must contain every resource and operation name, which increases policy complexity and is inacceptable. Furthermore, it should be possible to define, whether a service itself is a resource, or if it administers various resources. During the process of service announcement, the concerning policy part has to be parsed and an appropriated service object has to be constructed.

It is imaginable, that these mappings are partially generated by an IDL compiler, when using an IDL language for the service's interface definition. For instance, one marks all defined methods of the service interface as read, write, or read-write. The compiler constructs a policy sample, containing the mappings of operation names to these abstract operation contexts. A reference monitor, that implements some information flow policy now can use the policy sample to map operations to these abstract one.

However, the resource and operation mapping is an open issue, that needs to be further investigated. Another feature, that is desirable, is the integration of the memory quota a subject owns into the policy language. As I mentioned briefly in section 3.1, in Bastei a parent handles a memory quota for each of its children. Consequently a parent not only acts as a reference monitor for its children, but also as resource monitor (compare to section 2.2.2), at least for memory. A policy language enhancement and potentially a migration of the quota mechanism out of the Bastei framework into the Monitor framework needs to be discussed. Currently, memory quota is associated with child objects, but it seems to be more sensible to attach it to subjects of the policy.

6 Conclusion

In my thesis I have determined existing security policy models and their different properties and commonalities, as well as arrangements of security mechanisms in existing operating systems. Also, I have investigated the capability mechanism of the microkernel L4.sec, a new experimental security kernel, for which it was my task to evaluate the management of different policies on top of it.

In parallel, members of the operating system research group of the University of Technology Dresden developed a framework, that is called Bastei, and which defines among others a protocol of capability propagation between clients and services. Moreover, it foregoes global names in a system and introduces a nested program tree, where children nodes are controlled by parent nodes, with respect to their access facilities to other nodes in the tree.

I designed an own framework called Monitor, that takes advantage of the Bastei architecture, and which supports software developers creating new applications, that act as parent nodes within Bastei, and which implement a specific security policy model. Until now, this framework supports a model, which is equivalent to the Access Control Matrix model, and an implementation of the Domain Type Enforcement model, as well as an authentication mechanism. In addition, I have outlined solutions for missing features, that serve as prerequisite for other policy models. In a prototypical solution I have implemented two reference monitors respectively parent nodes, that show the usage of the framework and served as evaluation base.

One of the main results of my work is the discovery, that the usage of several, different, and nested reference monitors promises better security properties, than one complex reference monitor. Moreover, we can use several different security policies in one system without interferences between them. Also, the presented architecture simplifies the definition of individual policies, that work upon basic and common policies, which fit for a lot of operational areas.

A detailed performance measurement of the new Bastei architecture, as well as an overhead measurement of the Monitor framework logic is left out in this work.

Glossary

ACL Access Control List

ACM Access Control Matrix

AIM Access Isolation Mechanism (in MULTICS)

AVC Access Vector Cache

CDI Constrained Data Item

CMS Content Management System

DAC Discretionary Access Control

DDT Domain Definition Table

DMZ Demilitarized Zone

DoS Denial of Service

DRM Digital Rights Management

DTE Domain Type Enforcement

DTT Domain Transition Table

EROS Extremely Reliable Operating System

FLASK Fluke Advanced Security Kernel

ICAP Identity-based Capability Protection System

IDL Interface Definition Language

IPC Inter Process Communication

IVP Integrity Verification Procedures

LOCK Logical Coprocessing Kernel

MAC Mandatory Access Control

MLS Multilevel Security

MULTICS Multiplexed Information and Computing Service

NCSC	National Computer Security Center
NGSCB	Next-Generation Secure Computing Base
NSA	National Security Agency
OASIS	Organization for the Advancement of Structured Information Standards
ORCON	Originator Controlled Access Control
PACL	Propagated Access Control List
PSOS	Provable Secure Operating System
RBAC	Role Based Access Control
SAT	Secure ADA Target
SCAP	Secure Capability Architecture
SELinux	Security Enhanced Linux
SLOC	Source Lines Of Code
TCB	Trusted Computing Base
TCSEC	Trusted Computer System Evaluation Criteria
TE	Type Enforcement
TP	Transformation Procedure
UCON	Usage Control
UDI	Unconstrained Data Item
XACML	eXtensible Access Control Markup Language

Bibliography

[And72] James P. Anderson. *Computer Security Technology Planning Study Volume II*, volume 2 of *MA 01730*. Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, esd-tr-73-51 edition, October 1972. 15

[BCD69] A. Bensoussan, C. T. Clingen, and R. C. Daley. The multics virtual memory. In *SOSP '69: Proceedings of the second Symposium on Operating Systems Principles*, pages 30–42, New York, NY, USA, 1969. ACM Press. 16, 17

[Bib77] K.J. Biba. Integrity considerations for secure computer systems. Technical Report Report TR-3153, MITRE Corporation, Bedford, Mass., 1977. 6

[Bis03] Matt Bishop. *Computer security: art and science*. Addison-Wesley, 2003. 13

[BK85] W. Boebert and R. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, page 18, October 1985. 6

[BL73] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: mathematical foundations. Technical report, MITRE Corporation, 1973. 4

[BL76] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: unified exposition and multics interpretation. Technical report, MITRE Corporation, 1976. 4, 5

[BN] Dr. David F.C. Brewer and Dr. Micheal J. Nash. The chinese wall security policy. In *Proceedings of IEEE Symposium on Security and Privacy*. 8, 10

[BSS$^+$95a] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A domain and type enforcement unix prototype. In *Proceedings of the fifth USENIX UNIX Security Symposium*, pages 127–140, Salt Lake City, Utah, USA, June 1995. USENIX. 7

[BSS$^+$95b] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical domain and type enforcement for Unix. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, page 66, 1995. 14

[Cen83] DoD Computer Security Center. *Trusted Computer Evaluation Criteria*. US Department of Defense, Washington, DC, August 1983. 12

[Cha03] T. M. Chalfant. Role based access control and secure shell - a closer look at two solaris operating environment security features. Technical report, Sun Microsystems, June 2003. 10

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194, April 1987. 7

[Den71] Peter J. Denning. Third generation computer systems. *ACM Comput. Surv.*, 3(4):175–216, 1971. 3

[DH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966. 16

[DLSU04] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, 2004. Latest version available from: `http://l4hq.org/docs/manuals/`. 21

[EKV+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th Symposium on Operating Systems Principles*, Brighton, UK, October 2005. 14

[FK92] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992. 10

[Gon89] Li Gong. A secure identity-based capability system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 56–65, 1989. 17

[Gra89] R. Graubert. On the need for a third form of access control. In *Proceedings of the 12th National Computer Security Conference*, pages 296–304, October 1989. 13

[Gro91] Michael Groß. Vertrauenswürdiges booten als grundlage authentischer basissysteme. In *Vis '91: Verlässliche Informationssysteme, GI-Fachtagung*, pages 190–207, London, UK, 1991. Springer-Verlag. 19

[Har88] Norman Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review*, 22(4):36–38, 1988. 18

[Här02] Hermann Härtig. Security Architectures Revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002. 20

[HHF+05] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The nizza secure-system architecture. Presented at The First International Conference on Collaborative Computing: Networking, Applications and Worksharing., December 2005. 20

[HPHS04] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004. 20

[Kar88] Paul Ashley Karger. *Improving security and performance for capability systems.* PhD thesis, University of Cambridge, 30 March 1988. 8, 17

[Kar89] Paul A. Karger. New methods for immediate revocation. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 48–55, 1989. 17

[Kau04] Bernhard Kauer. Authenticated booting for L4. URL: `http://os.inf.tu-dresden.de/papers_ps/kauer-beleg.pdf`, November 2004. 19

[Kau05] Bernhard Kauer. L4.sec implementation - kernel memory management. Master's thesis, TU Dresden, 2005. 21

[KL86] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 66–77, May 1986. 17

[l4e03] L4env concept paper. Technical report, Technical University of Dresden, 2003. 23

[Lam71] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, page 437, Princeton, 1971. 3

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973. 17

[Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996. 21

[Lie99] J. Liedtke. L4 nucleus version x reference manual (x86). Technical report, September 1999. 21

[Mil92] J.K. Millen. A resource allocation model for denial of service. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 137–147. IEEE, May 1992. 19

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the Symposium on Operating Systems Principles*, pages 129–142, 1997. 14

[MYS03] M. Miller, K. Yee, and J. Shapiro. Capability myths demolished, 2003. 18

[Nat06] National Security Agency. Security-Enhanced Linux. URL: `http://www.nsa.gov/selinux/`, 28 May 2006. 7, 10, 21

[NFCH05] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005. 19

[OAS06] OASIS. eXtensible Access Control Markup Language (XACML) Version 2.0. URL: `http://docs.oasis-open.org/xacml/2.0/`, May 2006. 14

[PRS$^+$01] B. Pfitzmann, J. Riordan, Ch. Stüble, M. Waidner, and A. Weber. Die PERSEUS System-Architektur. In D. Fox, M. Köhntopp, and A. Pfitzmann, editors, *Verlässliche IT-Systeme (VIS)*. GI, Vieweg, September 2001. 21

[Sal74] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, 1974. 16, 17

[Say02] O. Sami Saydjari. Lock : An historical perspective. *acsac*, 00:96, 2002. 6

[Sha99] Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, 1999. 21

[SP02a] R. Sandhu and J. Park. Originator control in usage control. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, page 60, Washington, DC, USA, 2002. IEEE Computer Society. 11

[SP02b] Ravi Sandhu and Jaehong Park. Towards usage control models: beyond traditional access control. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 57–64, New York, NY, USA, 2002. ACM Press. 11

[SSF99] J.S. Shapiro, J.M. Smith, and D.J. Farber. Eros: A fast capability system. In *Symposium operating system principles*, pages 170–185, 1999. 17

[SSL$^+$99] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The FLASK security architecture: system support for diverse security policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, August 1999. 15, 16, 21

[tud06] Drops - the dresden real-time operating system project - publications. URL: `http://os.inf.tu-dresden.de/drops/doc.html`, September 2006. 27

[Vle06] Tom Van Vleck. Multics. URL: `http://www.multicians.org`, June 2006. 5

[VSJ] Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sushil Jajodia. Policies, models, and languages for access control. In *Databases in Networked Information Systems: 4th International Workshop*. 14

[Whe06] David A. Wheeler. David a. wheeler - personal home page. URL: `http://www.dwheeler.com/sloccount/`, 24 September 2006. 50

[wik06] Wikipedia entry - template method pattern. URL: `http://en.wikipedia.org/wiki/Template_method_pattern`, 19 September 2006. 41